Hybrid Simulation of Packet-Level Networks and Functional-Level Routers

Mirko Stoffers
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: stoffers@gatech.edu

George Riley
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: riley@gatech.edu

***Abstract*—We discuss our approach to federating dissimilar discrete event simulations, leveraging the strengths and design goals of both, to produce a packet-level detailed network model federated with a component-level detailed input-queuing router model. All existing network simulation tools that we are aware of incorporate a very simplistic model for the flow of packets through a router. The simplistic model simply responds to a *packet receipt* event by performing a route look-up and adding the packet to the output queue of the next-hop output interface. This is often simulated to take place in zero time, or with rudimentary probabilistic models of delay within a router. However, modern high-end routers are designed using a complex *input-queuing* methodology and a sophisticated scheduling approach to move packets through a crossbar switch from the input queue to the output queue. We used the popular *ns–3* network simulator to create realistic packet-level models of network load, and the *Manifold* computer architecture simulator to create a realistic model of data movement through an input-queued router. We federated the two by means of two alternative approaches: First, two *POSIX threads* run within a single simulation process and utilize the shared memory for both time synchronization and packet exchange. Second, we used the well-known *MPI* message passing library for the federation. Our results show that the detailed router models can in fact produce somewhat different packet delay and loss characteristics than the simplistic router models at the expense of considerable computational complexity.**

I. Introduction

Modern network traffic loads demand high performance from the network, both in terms of throughput and delay. Naturally, network simulation tools are used heavily to analyze how a particular application of interest will perform on various networks with varying capabilities and configurations. Using the simulation results as a guide, network operators then design a network with given link bandwidths, queuing methods, queue sizes and other characteristics that will presumably support the applications and provide the performance required. With modern-day applications such as voice-over-IP, video streaming, and music streaming, the network performance in terms of loss, delay and jitter is paramount. The developers of network simulation tools such as *ns–3* [1] devote considerable effort into making their simulation models realistic so that simulation results can be used to guide design parameters. However, virtually all network simulators incorporate very simplistic models of the flow of information *within a router*. A typical router model simply responds to a *packet receipt* event by performing a route look-up, and then adding the received packet to the output queue of the next hop interface. This simplistic model is easy to implement and easy to understand. However, actual high-end Internet routers such as those developed by Cisco, utilize much more complicated approaches to responding to a newly arrived packet and routing the packet data through the router to the appropriate output interface. For example, the well-known *iSLIP* design [2] uses a complex set of *input-queues* to store incoming packets, a novel scheduling algorithm to request and reserve crossbar switch circuits, and a set of output queues to store the packet after successfully traversing the crossbar. It is clear to us that routers utilizing this approach will exhibit considerably different performance metrics with respect to delay and packet loss than will the simplistic router model discussed above.

To address this issue, we chose to develop a detailed model of the flow of information through a router using the *iSLIP* scheduling algorithm, input-queued arrival processing, and a crossbar approach for moving packets from inputs to outputs. While we likely could have developed this model using the *ns–3* network simulator, we decided that this approach would have meant using the network simulation in a way it was not designed to be used, namely modeling hardware components with clock-driven actions and low-level data flow between components. Instead, we chose to use the *Manifold* architecture simulator [3] which is in fact designed to do exactly what we needed. Further, this approach also simplifies the extension of the hardware model in a domain designed for this purpose. The construction of the *iSLIP* model for *Manifold* was relatively straightforward, requiring models for the input-queues, the crossbar switch (implemented using *Manifold* components with clock-driven actions), and the matching algorithm. Of course, once we had the detailed router model in *Manifold* and the detailed network model in *ns–3*, we then needed to create a federated simulation using both simulators in concert. Our approach to this federation is discussed in detail later.

The remainder of this paper is organized as follows. Sec-

This work is supported in part by the U. S. National Science Foundation under contract numbers CNS 0958015 and CNS 855110.

tion II gives some details about *ns–3*, *Manifold* and *iSLIP*. Section III discusses briefly some other works federating dissimilar simulations. Section IV explains our approach to federating the two simulators. Section V shows the results of our federated simulation tool both in terms of measured network performance and the run-time performance of the simulator itself. Finally, Section VI summarizes the work.

II. Background

This section gives some background information for *ns–3*, *Manifold* and *iSLIP*, and describes our *Manifold iSLIP* model.

A. The ns–3 *Network Simulator*

The *ns–3* network simulator is the result of a collaborative effort between researchers at Georgia Tech, University of Washington, INRIA France, and Bucknell University. This work was begun in summer of 2006 as a result of a grant from the U. S. National Science Foundation. The approach proposed and used by the team was a complete redesign of a new simulation tool, rather than any attempt to modify or adapt older simulators such as *ns–2* [4]. In addition, the design was such that the ability to utilize distributed simulation methods to increase scale and performance was inherent in the design. The *ns–3* simulator has seen continual development and enhancement since 2006, with more than 40 developers worldwide contributing models to the tool. Recent *ns–3* releases have been downloaded more than 5000 times per month.

The design of *ns–3* uses a very novel event scheduling mechanism that allows any member function with arbitrary parameters on any *C++* object to act as an event handler. This approach greatly eases model development, as compared to simple well-known event handler abstract classes. Further, the *ns–3* design makes considerable use of helper objects that ease the burden on a simulator user when creating complex topologies with applications and protocol stacks. Of particular importance for our approach to federating *ns–3* and *Manifold* is the way network packets are modeled and managed in *ns–3*. The underlying data representation for the model of a network packet is simply an array of bytes, exactly as packets are represented in real networks and real routers. The rationale for this design decision was originally to ease the integration of *ns–3* with actual networks using *network emulation*.

Additionally, *ns–3* was designed and implemented to support distributed simulation of larger network topologies. Detailed discussion of the design and implementation of the *ns–3* distributed simulation can be found in [5]. The approach used to implement the distributed execution of *ns–3* was to utilize the well-known *MPI* message passing library for both time synchronization and packet migration between logical processes. As we discuss later, this design decision was particularly beneficial to our *ns–3* and *Manifold* federation.

B. The Manifold *Architecture Simulator*

The *Manifold* architecture simulator is a work-in-progress by our research group designed to allow designers of large-scale multi-core processing environments to model and analyze various techniques for connecting the multiple cores to other elements in the computing environment. To date we have an instruction emulation front-end based on the open source *QEmu* emulator, a detailed model for on-chip interconnection networks such as torus and grid networks, and models for coherent caches and memory systems.

The design of *Manifold* supports various levels of abstraction in the hardware being modeled, as detailed as flip-flops and logic gates, or as abstract as a memory system with address selectors input, a read/write flag input and data output. All models in the *Manifold* simulator consist of one or more *components* (which might be a network interface or a logic gate) that are connected together with *Links* that move information from component outputs to the appropriate component inputs. Naturally, links introduce delays as information is moved from outputs to inputs. These delays can either be specified in units of *Clock Ticks* or in seconds. Components can be either *synchronous* and perform all state transitions on rising-edge or falling-edge clock ticks, or they can be *asynchronous* and perform state transitions whenever their input state changes, or they can perform state transitions on either of these events.

The *Manifold* simulator uses fairly standard discrete event simulation techniques to handle the advancement of time in the model, with the usual future event set (FES) and event handler objects for each event. Additionally, it maintains a set of clocks and regularly determines the next event by means of both the clocks and the FES. Further, the design supports distributed simulation, allowing components connected via links to be modeled in separate logical processes. Both the transmission of remote events and the time synchronization activities are handled by using *MPI* calls. All data moved between components is represented in an abstract, simple array of bytes approach. Only the sending component and the receiving component of any information are aware of the semantics of the information. As we discuss later, the use of *MPI* for time synchronization led to somewhat simple integration with *ns–3*.

C. The iSLIP *Router Design*

The router and switch model we created in *Manifold* is based on the *iSLIP* approach by McKeown [2]. This design utilizes a set of input links for receiving incoming packets, a set of *Input Queues* to store incoming packets temporarily (described later), a crossbar switch that moves packets from the input queues to the output interfaces, and a set of output queues to store packets awaiting transmission to the next hop. Without going into great detail about the *iSLIP* design, the input queues are designed in a way preventing *head-of-line blocking*, which results when a packet at the head of a single FIFO input queue is to be moved to an output that is presently busy, but packet(s) behind the head are to be moved to an available output. Using k input queues (called *Virtual Output Queues*) for each of k ports solves this problem.

In addition to the use of *Virtual Output Queues*, the *iSLIP* design incorporates a simple but novel matching algorithm for determining which inputs should be granted access to the

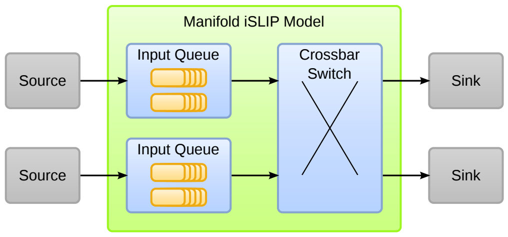


Fig. 1. The *iSLIP* Model for a 2 × 2 Router.

crossbar to move data to their desired outputs. This algorithm guarantees fairness and guarantees no starvation for any input.

In studying the design of an *iSLIP* router, it was clear to us that such a design will certainly affect delay and jitter for packets being routed, especially in high-load conditions. Equally clear was the fact that existing router models in *ns–3* are likely too simplistic for measuring delay and jitter in high-speed networks under heavy load conditions. These observations motivated our work towards modeling *iSLIP* routers in *Manifold* and network traffic generation in *ns–3*.

We implemented a *Manifold* model of this *iSLIP* router as shown in Figure 1. In general, this model consists of two components. The *Input Queue* component holds a configurable set of virtual output queues. On an incoming packet it determines the correct queue, stores the packet, and informs the *Crossbar Switch* component. The internal clock of this crossbar switch defines the time-stamps to compute a matching between input and output ports. A predefined time after the matching was computed, the incoming packets are forwarded to the outputs of the crossbar switch.

The model relies on packet sources and sinks. In the federation with *ns–3* the packet sources will be the *ns–3* MAC layer modules, and the sinks will be the output queues of *ns–3*.

III. Related Work

There have been some prior published works describing the integration of two or more independent simulation environments. Indeed, the entire focus of the well-known *High-Level Architecture* (*HLA*) is to facilitate exactly that goal. The number of works discussing the use of *HLA* are numerous and not enumerated here. Nevertheless, the *HLA* relies on the simulators to be *HLA* compliant. Unfortunately, neither *ns–3* nor *Manifold* maintains this feature.

In work by Yeung, Takai, Bagrodia, et al. [6], the *QualNet* network simulator was integrated with the detailed physical layer path loss models found in the *Matlab* tool-set. *Qualnet* and *Matlab* exchange packet transmission and reception data as packets move from senders to receivers, and the packet reception probability is determined based on computed path loss information in *Matlab*. This did in fact produce accurate results, but the overhead imposed by the *Matlab* environment resulted in an overall slowdown ratio of nearly 10,000.

In two separate works by Xu, Riley, Ammar, et al. [7], [8] the *ns–2* simulator was integrated with the *GloMoSim* simulator. The intent of this work was to utilize the strengths of each simulator, using *ns–2* for the wired part of the simulated network, and *GloMoSim* for the wireless part.

Although not directly related, in a similar work by Wu and Fujimoto [9] the authors report on difficulties integrating the commercial network simulator *OpNet* with *itself*. This required the integration of some form of message passing and time synchronization similar to what we needed in the work here. At the time of that work, the chosen approach was a locally developed *Run-time Infrastructure Kit* (*RTIKit*) for these actions.

Excepting the work by Yeung et al., all of the above works were attempting to integrate multiple simulators, but the simulators were all *network simulators*. In our work we are integrating a network simulator with a *component-level hardware simulator*.

A similar approach to model the effects of hard- and software detail into a network simulation is called *network emulation*. Here, a network simulator is federated with physical nodes or virtual machines. *NIST Net* by Carson and Santay [10] emulates a network in order to apply effects like delay or packet loss on traffic arriving at the physical network links of the simulating node. While this requires the simulator to work in real-time, the *SliceTime* approach by Weingärtner et al. [11] federates virtual machines with *ns–3*. The advantage of virtual machines is the possibility to slow down the machine in a way the software does not notice to avoid desynchronization. As opposed to our approach NIST Net relies on the existence of physical nodes, and SliceTime models details of network software rather than hardware.

IV. Federation of the Simulators

In this section we describe the federation of *ns–3* and *Manifold*. First, we discuss the goals of our design in Section IV-A. We took two different approaches to federate the simulators. In Section IV-B we introduce a federation based on POSIX threads. In Section IV-C we describe an MPI based federation scheme. Both schemes share the same component structure and modifications to the available *ns–3* components (see Section IV-D). Finally, we discuss selected issues on the design in Section IV-E.

A. Design Goals

The primary goal of our design is to federate *ns–3* and *Manifold* in a way that a wide variety of router and switch models can be applied on both sides. This means on the *ns–3* side that we do not require a certain network or MAC layer protocol, but instead allow reusing the available protocol models like IPv4 or IPv6. In *Manifold* we want to enable model developers to create router models in arbitrary degree of detail. Our example model implements the virtual output queues connected to a crossbar switch that computes the *iSLIP* matching in a predefined time interval. Nevertheless, we designed the federation in a way such that a detailed implementation of the arbiters, e. g., as FPGAs, would also be possible to predict the time needed to determine a decision in certain situations. Furthermore, our design enables easy implementation of different matching algorithms or ways to organize the input queues.

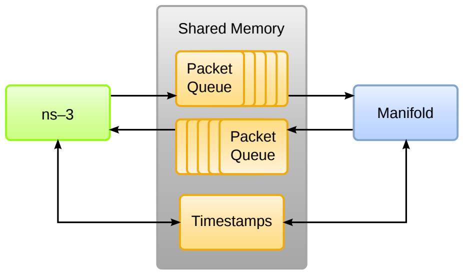


Fig. 2. The thread based scheme utilizes the shared memory for packet transmission and time synchronization.

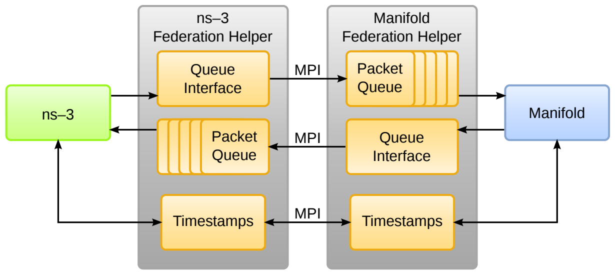


Fig. 3. The MPI based scheme utilizes two federation helpers communicating via MPI messages for packet transmission and time synchronization.

We note that simulator performance is a secondary goal of this work. We know that additional details cannot speed-up a simulation, but will require more run-time. Nevertheless, we tried to keep the synchronization overhead to a minimum.

B. *POSIX Threads Based Federation*

In general, there are two challenges in the federation of dissimilar discrete event simulators. First, every simulator usually maintains its own data structures. In the federation of *ns–3* and *Manifold* for the purpose of network simulation, we only need to interchange network packets as described in the following. Second, simultaneously running simulators need to be time-synchronized in order to avoid that one simulator receives an event to be executed before a recently committed event.

Packet Exchanging: In *ns–3* a packet is represented by a data structure that models a byte array for the packet data itself, and maintains tags to enable the user to store additional data. In our *Manifold iSLIP* model we represent a packet by a data structure that only holds the size of the packet (rather than all the payload), the input and output port for the crossbar switch to determine where the packet should be sent, as well as a void pointer for arbitrary user data. When a packet is to be transmitted from *ns–3* to *Manifold*, we create the corresponding *Manifold* data structure, and fill in the corresponding fields. We utilize the void pointer field to add a pointer to a data structure (in *ns–3* memory) maintaining information about the callback function and data as well as an *ns–3* smart pointer to the packet itself.

For the communication from *ns–3* to *Manifold* we created a queue in the shared memory (see Figure 2). A packet is transmitted from *ns–3* to *Manifold* by enqueuing the required data. After the packet has passed the crossbar switch, it is stored in another queue and finally dequeued and processed by *ns–3*.

Time Synchronization: The thread based time synchronization benefits from memory shared by the lightweight threads. The time-stamp of the next event in the FES of each simulator is maintained in this shared memory. In order to determine whether it is safe to execute the next event, a simulator therefore needs to compare the time of the next event in its own FES to the time of the corresponding simulator. Additionally, it is necessary to verify that there is no transient message pending in the queues which might reduce the time-stamp of the next event. If both of the queues are empty, the simulator with the earlier event can proceed since we lock the shared memory during dequeuing and time-stamp adaption.

C. *MPI Based Federation*

In order to achieve a high degree of code reuse we decided to use the same interface between the simulators and the federation engine for both schemes (see Figure 3). However, for the MPI based scheme we have two *federation helpers*, one for each simulator, that interchange MPI messages to provide the required functionality.

Packet Exchanging: If simulator A transmits a packet to the federation helper for transmission to simulator B, the helper of A creates an MPI message including the relevant data and transmits this message to the process running B. The helper of B hands the incoming messages back to the simulator core whenever the core requests dequeuing a message. This is performed in the simulation main loop in order to determine the next event to handle.

Time Synchronization: Unfortunately, in an MPI based federation we cannot utilize shared memory since MPI processes might run on different machines. However, instead we can use MPI messages to exchange the necessary information. Both *ns–3* and *Manifold* already maintain the same time synchronization protocol based on the use of the MPI `Allgather` *collective* which is designed to produce a global minimum or maximum efficiently. Unfortunately, the time synchronization data structures exchanged via MPI are different, such that we could not reuse the implementation. However, the reimplementation of the time synchronization protocol was straightforward.

D. *Component Structure*

Both *ns–3* and *Manifold* are based on a modular component structure. We designed the federation to minimize the changes made to the existing components, but instead add further ones. This accords to the design goal to enable arbitrary *ns–3* models without requiring too many adaptions.

Figure 4 sketches the modules of an *ns–3* IPv4 router (left) as well as the modules of the *iSLIP* crossbar switch model (right). On both sides we added two connector components (colored black in the figure), one for packet reception, one for packet transmission.

If the MAC layer module of *ns–3* receives a packet from the network layer to store in the output queue or transmit on the channel, in the federated approach the *ns–3* module transmits

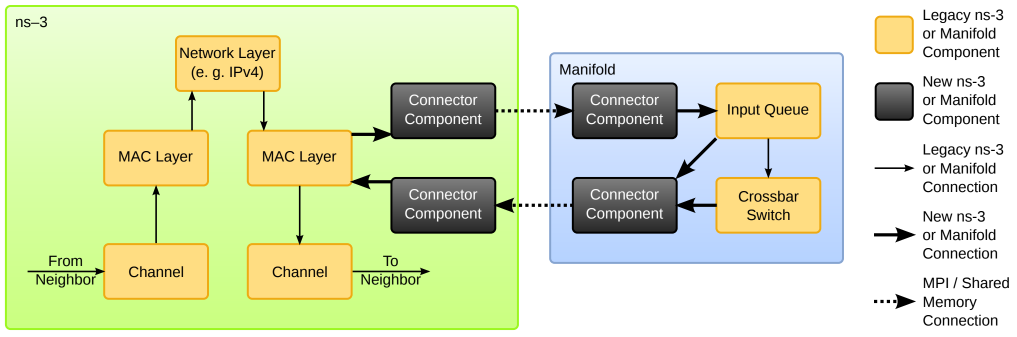

ns–3
Network Layer (e. g. IPv4)
MAC Layer
MAC Layer
Connector Component
Connector Component
From Neighbor
Channel
Channel
To Neighbor
Manifold
Connector Component
Input Queue
Connector Component
Crossbar Switch
Legacy ns-3 or Manifold Component
New ns-3 or Manifold Component
Legacy ns-3 or Manifold Connection
New ns-3 or Manifold Connection
MPI / Shared Memory Connection

Fig. 4. We added 2 new components to each model for packet exchange, and modify the *ns–3* MAC layer slightly.

the packet to the connector component instead. This requires a slight modification of the MAC layer, implemented by inheritance. The connector component now passes the packet via the federation helper depicted in Figure 2 and Figure 3 to *Manifold*. By means of the information included in the packet, the *Manifold* connector component needs to identify the input queue in which the provided packet should be stored. For this purpose, the *ns–3* MAC layer module also needs to piggyback its own identifier whenever it receives a packet from the channel. The input queue component stores every received packet in the respective virtual output queue. After the packet has passed the crossbar switch, it is forwarded to the *Manifold* connector component for transmission. This component again utilizes the helpers to transmit the packet back to *ns–3*. Here, we utilize the callback data referenced by the void pointer in the packet to find the correct MAC layer object. Now, the packet can be either stored in the output queue or transmitted to the channel.

If the input queue object determines that the corresponding virtual output queue is full, and the packet has to be dropped, it tags the packet accordingly. After that, it bypasses the crossbar switch and sends the packet back to *ns–3* where the cleanup can be performed.[1]

Effectively, our *iSLIP* model performs only two operations on each packet: It either drops the packet at a full input queue, or it adds additional delay introduced by input queuing and crossbar switch transmission.

E. Discussion

In the following we discuss certain aspects of our system design. We analyze the pros and cons of alternatives and justify our decisions.

Zero Look-ahead: The integration of the *iSLIP* router at the MAC layer of *ns–3* comes with a significant disadvantage: The messages exchanged between *ns–3* and *Manifold* do not include a delay. This means that the look-ahead between the two logical processes (LPs) of *ns–3* and *Manifold* is zero, which completely defeats parallel execution. Nevertheless, we stated the design goal to reuse the existing functionality of both simulators. *ns–3* provides a feature-full implementation of different network layer protocols such as IPv4 and IPv6 as well as MAC layer protocols. The only way to achieve look-ahead between LPs is to define an LP border cutting a delayed network link. However, this defeats the reuse of *Manifold* crossbar switch models and *ns–3* network layer models.

While we cannot achieve parallel speedup between *ns–3* and *Manifold* we argue that it is still possible to split the network into several LPs with look-ahead in between. This partitioning can then be applied to both *ns–3* and *Manifold*, i. e., two routers reside in the same LP in *Manifold* if and only if they reside in the same LP in *ns–3*. Defining an LP in both simulations containing only a single router would then allow the same degree of parallelism as if the router was completely implemented in *Manifold* with the only disadvantage being the decomposition of one router into two LPs with the corresponding overhead.

In this case, it is also possible to realize the federation by means of POSIX threads while running different LPs on different machines synchronized by MPI.

Packet Data Structure: When designing the *iSLIP* model we also had to decide how a packet is represented in *iSLIP*. We decided to use a simple data structure maintaining input and output port, packet size, and a void pointer for user data. While a representation of the packet as a stream of bytes like in *ns–3* could have eased the implementation of the packet exchange between the two simulators, we decided to use this approach to avoid the unnecessary transmission of the payload. Further, a byte stream does not inherently include the information about input and output port which is required by the crossbar switch matching algorithm. Instead, we keep the packet in the *ns–3* memory and the *iSLIP* model only computes the delay.

Information Exchange Paradigm: There are two common ways to federate two simulators. On the one hand every simulator can constitute its own system process. On the other hand the simulators can be linked into a single binary and executed in a threaded way.

While a tighter coupling allows faster information exchange, especially by the utilization of shared memory, linking two simulators into a single binary might lead to the problem of name clashes.

We decided to implement both approaches and compare the performance in Section V. Indeed, name clashes were not an issue since both simulators make strict use of C++

[1]Since the payload of the packet is not transmitted to *Manifold*, it needs to be deleted by *ns–3*. *Manifold* only stores a void pointer that cannot be used to release memory.

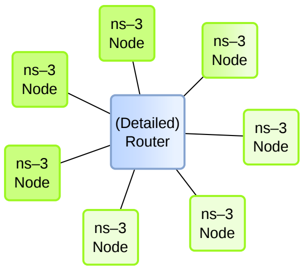


Fig. 5. The Topology of our Evaluation for an Exemplary 7 Node Network

name-spaces. However, due to the nature of the *ns–3* build system that copies all header files into a single directory prior to compiling, we had to rename one *Manifold* header file to avoid overwriting.

V. Evaluation

In this section we discuss the experiments we ran to evaluate the necessity of a detailed router model on the one hand as well as the computational overhead on the other hand. We first introduce the common evaluation environment before we focus on the specific experiments and discuss their results.

A. General Evaluation Environment

In general, we chose to keep the design of the evaluation environment simple. Although the interaction between multiple routers might be important in the performance analysis of larger networks, already a much simpler setup can show the importance of a detailed router model.

Our simulated network therefore consists of a simple star topology (see Figure 5) with one router in the center and a variable number of nodes sending packets to each other through that router. Every node has 10 independent UDP traffic streams sending to random destinations. The traffic follows an on-off-pattern with exponentially distributed on and off times, both averaging in 50 ms. After each on-off-cycle a new destination is selected randomly among all available nodes. The full duplex network links feature a data rate of 1 Gb/s, and a delay of 0.3 ms.

We ran all simulations on an Ubuntu 11.10 host system equipped with a 2.8 GHz Intel Xeon quad-core CPU and a total of 6 GB of RAM. We used the recent development versions of both *ns–3* and *Manifold*.

B. Necessity of a Detailed Router Model

First, we investigate whether it is actually important to model a router in close detail. This is an essential question since a detailed router model will obviously add computational effort during the simulation as well as manual effort to build the model. Certainly, the answer to this question will depend on the scenario and the purpose of the simulation. There are many applications like voice-over-IP and video streaming where end-to-end delay, jitter and packet loss matters. Since the delay and packet loss of the links is not affected by our approach, we focus on the effects at the router.

We therefore applied the simple star topology model described above, and connected 16 nodes to the router. During the on-time, each of the 160 applications continuously sent UDP packets with 512 B of payload at a net data rate of 100 Mb/s. On average, this results in a link utilization of about 50 %.

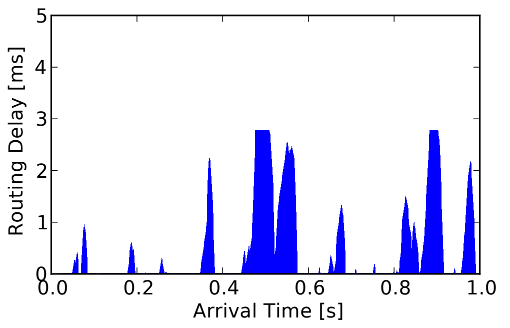
An important factor trading delay vs. packet loss is the buffer size of the routers. There is no consent in the research community of an appropriate buffer size [12]–[15]. Instead, there are different rules of thumb. For this simulation we applied the rule introduced by Appenzeller et al. in [12]. For an RTT of 250 ms (see RFC 3439), and 10,000 simultaneous flows (see [12]) the buffer capacity has to equal approximately 300 KB. For the abstract *ns–3* router we therefore set the output queue size to 640 packets (of 512 B payload each).

To create comparable results, it is important that the buffer capacity of the abstract router equals the capacity of the detailed router. We decided to equally split the 640 packets buffering capacity among input and output buffers. This means, the size of each output queue was 320 packets. However, while we had 16 input queues that can transmit packets to the possibly contended output link, only those which actually do transmit add to the buffer capacity. The remaining buffers stay empty. Unfortunately, we can not predict the number of inputs creating the congestion on the output a priori. If all inputs transmit to the same output, the buffer capacity of each virtual output queue needs to be 20 packets (320 packets total input capacity over 16 ports). If only a single input transmits to one output queue, the comparable input buffer capacity is 320 packets (320 packets over 1 port). As a compromise, we decided to assign an input buffer capacity of 80 packets (320 packets over $\sqrt{16}$ ports).

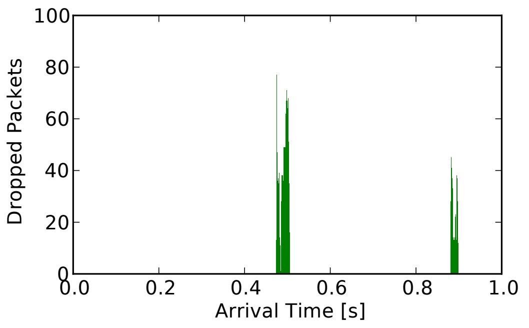
Another important factor is the performance of the router. The router consists of a crossbar switch and arbiters that cooperatively compute the matching. These components operate at a certain clock frequency, i. e., they are able to handle a certain amount of packets per second. In order to save money as well as energy these components should operate only slightly faster than the line rate. This means, for a line rate of 1 Gb/s, and a (fixed[2]) packet size of 512 B the crossbar switch must be able to handle 244,000 packets per second. We therefore set the clock frequency to 250 kHz, i. e., the router is able to forward 250,000 packets per second and pair of ports. Furthermore, we investigate the behavior of the router for a crossbar switch slower than the line rate (220 kHz).

We ran the simulation for 1 second, and measured the router introduced delay of every single packet that successfully passed the router. For each interval of 1 ms we determined the maximum routing delay for all packets arriving at the router in this interval. Since there is always a large number of packets traversing the router instantaneously via a non-congested link, we consider the average delay rather uninteresting. Instead, we plotted the maximum delay in each of those short intervals over the course of the simulation. Furthermore, we counted

[2]Note: High performance switches are usually installed in networks such as SONET or ATM where data is not organized in varying size packets, but fixed size frames or cells.

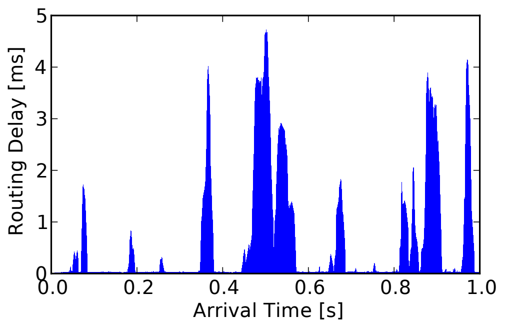


(a) Maximum Delay in Small Intervals During the Course of the Simulation Run

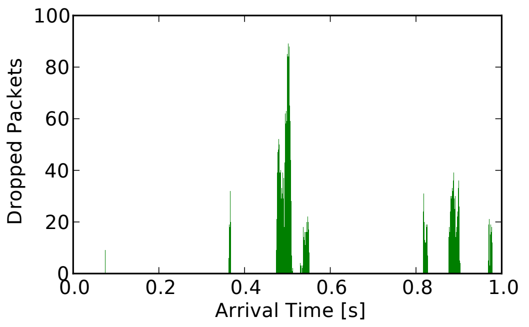


(b) Number of Lost Packets in Small Intervals During the Course of the Simulation Run

Fig. 6. Delay and Packet Loss at the Abstract *ns–3* Router



(a) Maximum Delay in Small Intervals During the Course of the Simulation Run



(b) Number of Lost Packets in Small Intervals During the Course of the Simulation Run

Fig. 7. Delay and Packet Loss at an Input-queued Router Computing 250,000 Matchings per Second

the number of packets arriving during this interval, but being dropped in the router as a consequence of buffer overflow.

Figure 6a shows the delay added by the abstract *ns–3* router, mainly caused by output queuing. We observe uncongested periods of time where the delay is almost zero, i. e., the first bit of the packet is forwarded as soon as the last bit has been received. However, when one output link is over-utilized, packets are stored in the output queue and cause significant delay. After the applications have stopped sending, the queue drains out again, and the delay reduces to almost zero.

Right after 0.45 s we observe a different behavior: The delay increases up to a level of 2.8 ms, and then constantly stays at this level for tens of milliseconds before reducing again. At the same point in time we observe a large number of dropped packets (see Figure 6b).

This behavior is easy to explain: While a large number of packets is addressed to one single output link, all those packets are stored in the corresponding output buffer. As soon as this buffer has reached its capacity, every additional packet is dropped. Nevertheless, packets are still dequeued and transmitted, such that another slot becomes available. Every packet stored within that slot has to wait for the preceding 639 packets in the queue to be transmitted. With the given link data rate and packet size this takes approximately 2.8 ms which is exactly the delay for that packet.

We now compare the results of the simple *ns–3* router to the results of the detailed *iSLIP* router with a clock frequency of 250 kHz (see Figure 7). The traffic generated by the applications equals the traffic during the simulation with the abstract router exactly. We used the same random number generator seed.

At the time of 0.45 s we observe a rather high delay of almost 5 ms on the detailed router (cf. to 2.8 ms for the abstract router). This is because the maximum possible buffering capacity of this router is higher than the capacity of the abstract router. Nevertheless, we also observe a higher loss rate.

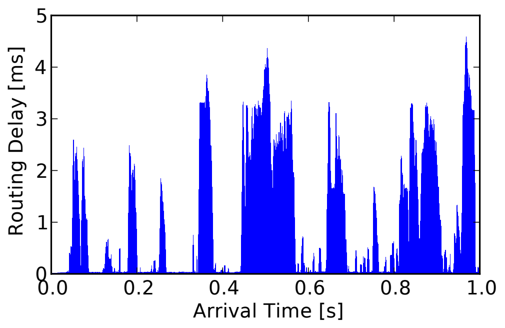
We now focus on the second delay peak at less than 0.1 s. On the abstract *ns–3* router the maximum delay in this period of time is about 1 ms. However, the delay of the detailed router increases up to almost 2 ms. This suggests that we chose the buffer capacity for the detailed router too high.

On the other hand, at the same point in time we observe a loss of several packets on the detailed router, while the abstract router is able to store all incoming packets in its output buffer. This means, to achieve the same loss rate as the abstract router achieved, we had to increase the buffer capacity. We observe this contradiction several times in the plots.

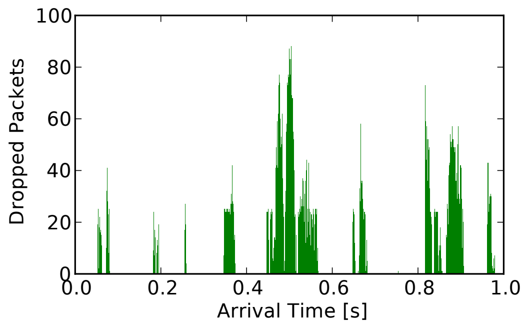
We conclude that, independent of the buffer size, the details modeled in the *Manifold* crossbar switch model significantly influence the performance of the router. Abstracting from these details can result in too high expectations on the performance of a router.

Furthermore, we observed that the size and the organization of the input queues influences the router performance. We therefore argue that in order to develop high performance network routers it is necessary to use a detailed model of these routers in order to be able to tune the buffer sizes accordingly.

By having a closer look to the bottom line of the plots we



(a) Maximum Delay in Small Intervals During the Course of the Simulation Run



(b) Number of Lost Packets in Small Intervals During the Course of the Simulation Run

Fig. 8. Delay and Packet Loss at an Input-queued Router Computing 220,000 Matchings per Second

also observe another fact: Although the delay in non-congested periods of time is almost zero for both the abstract and the detailed model, we can still observe a slightly higher delay in the detailed model. We attribute this to the fact that the detailed model additionally accounts for the time it takes to compute the crossbar switch matching, transmit the packet through the crossbar switch, and store it in one more queue.

Figure 8 shows the performance of the detailed router with a crossbar switch frequency of only 220 kHz. As expected, these results are even worse than the results of the faster crossbar switch. We argue that it might be also important to run simulations with a detailed router model in order to estimate which crossbar switch frequencies and matching algorithms provide reasonable packet delays and loss rates.

We therefore conclude that, in fact, the router model makes a difference in the network performance. Especially for router manufacturers it might be important to simulate their routers in the context of detailed network simulations. However, this comes at the expense of additional overhead.

C. Run-time Overhead

In order to investigate the run-time overhead added by the detailed *iSLIP* router model, we utilized a simulation setup similar to the one above. We expected the overhead to be mainly influenced by the router size (more ports increase the matching complexity), and the crossbar switch frequency (a higher frequency increases the number of matching events). We therefore varied the number of nodes (4 to 64), and the crossbar switch frequency (1 MHz to 100 MHz). We sent packets with a payload of 24 B at a net data rate of 70 Mb/s.

We defined a high buffer capacity of 640,000 packets (according to the widely used rule of thumb $B = RTT \times C$ for $RTT = 250\,\mathrm{ms}$ and $C = 1\,\mathrm{Gb/s}$). Again, we split this capacity for the detailed router in 320,000 packets output buffer and $\frac{320{,}000}{\sqrt{\#input\ ports}}$ packets per virtual output queue. These large queue sizes avoided the problem that the input-queued router could perform faster due to packet losses.

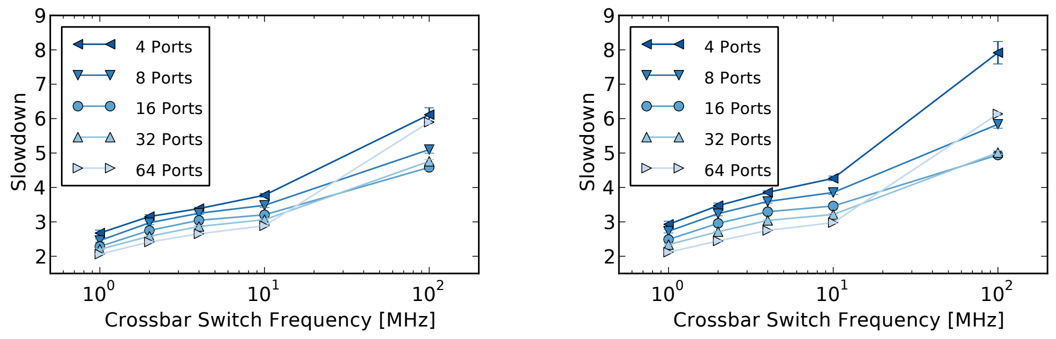
We ran the simulation for 0.2 s of simulated time, starting measuring the simulation run-time at 0.1 s to pay credit to the initial transient phase where all *ns–3* applications are in off state, and no contention occurs. We repeated each experiment 10 times. After that we divided the run-time taken with the detailed router model applied by the run-time taken by the simple router model simulation, yielding the slowdown resulting from the additional accuracy. We plotted the average slowdown as well as the 95 % confidence intervals for both federation schemes.

Figure 9a shows the slowdown by the detailed router with the POSIX threads based federation scheme applied. For low crossbar switch frequency values we observe that the slowdown with a low number of ports is higher than the slowdown with a large number of ports. This appears to be contrary to our claim that the effort to compute the matching increases with increasing number of ports. However, with increasing number of nodes the number of packets to simulate during the given time period increases. The total effort to simulate the packet traveling from the application layer of the source node to the application layer of the destination node is mainly evoked by the packet manipulation in *ns–3*. On the *Manifold* side the per packet effort is comparably low. The crossbar switch computes the matching every clock cycle independent of the number of packets in the input queues.

Nevertheless, we do observe more slowdown for a 32 or 64 port router (compared to a 16 port router) when the crossbar switch frequency is as high as 100 MHz. We explain this by the fact that for a 100 MHz router the per packet effort is comparably low while the effort to compute the matching constitutes most parts of the simulation run-time. The complexity of the matching itself is quadratic in the number of ports (every virtual output queue needs to be checked for a pending request), and therefore the effort to compute the matching for a router with many ports is significantly higher. However, this is still not significant for a lower frequency since the matching algorithm is executed less often.

For all the configuration parameters we observe a slowdown factor of up to 6. We argue that neither a lower nor a higher number of ports appears reasonable[3]. Furthermore, we set the crossbar switch frequency to the highest value mentioned in McKeown's paper. Therefore, we expect a higher slowdown only if the traffic rate is decreased. However, we believe that router models with a detailed queuing model are only necessary for highly utilized routers with congestion occurring.

[3]McKeown bases the *iSLIP* algorithm on 8 to 32 port switches [2].



(a) POSIX Threads Based Federation (b) MPI Based Federation

Fig. 9. Overall Slowdown in Simulation Performance

Figure 9b shows the simulation run-time with the MPI based federation scheme applied. The course of the curves is in general similar to the one observed for the thread based federation. However, as expected, the lightweight POSIX threads can slightly reduce the overhead as compared to the heavyweight MPI processes.

This can be observed best for a low number of nodes and high crossbar switch frequency. Due to the low traffic, the *ns–3* effort is comparably low. On the other hand there are a lot of events on the *Manifold* side and therefore a lot of synchronization messages have to be exchanged. This slows down the simulation with the MPI based federation scheme by a factor of 8 (as opposed to the factor of 6 for the thread based scheme).

We summarize that modeling a router in more detail comes at the expense of significantly reduced run-time performance. However, for all reasonable parameter values we chose, we observed a slowdown of less than one order of magnitude while former approaches to federating dissimilar simulators in order to achieve a higher degree of details showed even worse performance [6]. We therefore argue that the additional details are worth paying the fairly low overhead especially in the evaluation of network core components.

VI. Conclusion

In this paper we introduced our concept of federating the dissimilar simulators *ns–3* and *Manifold*. While *ns–3* is well suited for detailed network modeling, *Manifold* complements this by modeling the details of the hardware used to build the network nodes. We showed that a detailed model of input queued routers can significantly change the observed network performance. This additional accuracy however comes at the expense of additional overhead. In our evaluation the slowdown in simulation performance ranged at factors from 2 to 8.

We discussed that it makes sense to model the functionality of a network router partly in *ns–3* where e. g., a full implementation of IPv4 is already available, and partly in *Manifold*. This however does not enable speedup just by running two simulators simultaneously. Nevertheless, we argue that both simulators can still be parallelized by partitioning a big network with multiple routers. Future work should investigate the challenges and performance improvements of this idea. Orthogonal work could also deal with the *Manifold* router model and further increase the degree of details in order to investigate whether more factors affect the network performance significantly.

We conclude that the federation of *ns–3* and *Manifold* provides significantly more accurate results and argue that this tool can be very valuable, especially in the design of novel routers and switches.

References

[1] T. Henderson, M. Lacage, and G. Riley, “The *ns–3* Network Simulator,” http://www.nsnam.org, 2007.

[2] N. McKeown, “The *iSLIP* Scheduling Algorithm for Input-Queued Switches,” *IEEE/ACM Trans. on Netw.*, vol. 7, no. 2, pp. 188–201, 1999.

[3] “Manifold: A Scalable Simulation Infrastructure for Many Core Systems,” http://manifold.gatech.edu/, 2012.

[4] S. McCanne and S. Floyd, “The LBNL Network Simulator,” http://www.isi.edu/nsnam, 1997.

[5] J. Pelkey and G. Riley., “Distributed Simulation with MPI in *ns–3*,” in *Proc. of the 4th Int. ICST Conf. on Sim. Tools and Techn.* ICST, 2011, pp. 410–414.

[6] G. Yeung, M. Takai, R. Bagrodia, A. Mehrnia, and B. Daneshrad, “Detailed OFDM Modeling in Network Simulation of Mobile Ad Hoc Networks,” in *Proc. of the 18th Workshop on Parallel and Distr. Sim.* ACM, 2004, pp. 26–34.

[7] G. Riley, M. Ammar, R. Fujimoto, D. Xu, and K. Perumalla, “Distributed Network Simulations using the Dynamic Simulation Backplane,” in *Proc. of the 21st Annual Conf. on Distr. Comp. Systems.* IEEE Press, 2001, pp. 181–188.

[8] D. Xu, G. Riley, M. Ammar, and R. Fujimoto, “Split Protocol Stack Network Simulations Using the Dynamic Simulation Backplane,” in *Proc. of the 9th Int. Symp. on Modeling, Analysis and Sim. of Comp. and Telecomm. Systems.* IEEE Press, 2001, pp. 158–165.

[9] H. Wu, R. Fujimoto, and G. Riley, “Experiences Parallelizing a Commercial Network Simulator,” in *Proc. of the 33rd Winter Sim. Conf.* IEEE Press, 2001, pp. 1353–1360.

[10] M. Carson and D. Santay, “NIST Net – A Linux-based Network Emulation Tool,” *SIGCOMM Comp. Comm. Rev.*, vol. 33, no. 3, pp. 111–126, 2003.

[11] E. Weingärtner, F. Schmidt, H. vom Lehn, T. Heer, and K. Wehrle, “SliceTime: A platform for scalable and accurate network emulation,” in *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation.* USENIX Association, 2011, pp. 253–266.

[12] G. Appenzeller, I. Keslassy, and N. McKeown, “Sizing Router Buffers,” *SIGCOMM Comp. Comm. Rev.*, vol. 34, no. 4, pp. 281–292, 2004.

[13] A. Dhamdhere and C. Dovrolis, “Open Issues in Router Buffer Sizing,” *SIGCOMM Comp. Comm. Rev.*, vol. 36, no. 1, pp. 87–92, 2006.

[14] Y. Ganjali and N. McKeown, “Update on Buffer Sizing in Internet Routers,” *SIGCOMM Comp. Comm. Rev.*, vol. 36, no. 5, pp. 67–70, 2006.

[15] R. Prasad, C. Dovrolis, and M. Thottan, “Router Buffer Sizing for TCP Traffic and the Role of the Output/Input Capacity Ratio,” *IEEE/ACM Trans. on Networking*, vol. 17, no. 5, pp. 1645–1658, 2009.