Großer Beleg

Network Support on M^3

Georg Kotheimer

29. Mai 2018

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	M.Sc. Nils Asmussen

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 29. Mai 2018

Georg Kotheimer

Abstract

M^3 is a microkernel-based operating system that targets very heterogeneous platforms, containing mixed instruction set architectures and accelerators. The main evaluation platform of M^3 is gem5, which offers network device models such as Intel's 8254x line. However, M^3 lacks a network device driver and TCP/IP stack to make use of it.

The goal of this work is to port or develop a network device driver and design the basics of a TCP/IP stack for M^3 that takes advantage of M^3's unique properties.

Contents

List of Figures

1 Introduction

M^3 [2] is a microkernel-based operating system that targets very heterogeneous platforms, containing mixed instruction set architectures and accelerators. It does so by placing a hardware component, a data transfer unit (DTU), next to every compute unit (CU) that abstracts from the CU's details and exposes a common interface, allowing M^3 to treat them all in an equal manner. The CUs with their DTUs are integrated into a network-on-chip.

The main evaluation platform of M^3 is gem5 [3, 6], which is a flexible and extensible open source computer-system simulator. Out of the box gem5 offers several network interface controller (NIC) models such as Intel's 8254x line. However, M^3 lacks a NIC driver and TCP/IP stack to make use of it.

M^3's unique design demands a unique solution to integrate NICs. I propose to treat a NIC as just another CU. Therefore, I introduce a proxy component that allows me to accompany a NIC with a DTU, to integrate the NIC into M^3's network-on-chip. The NIC driver then can be executed on an arbitrary CU, using the communication mechanisms of its CU's DTU to configure and drive a NIC. On top of this, the network manager implements a TCP/IP protocol stack and provides an interface to applications for sending and receiving of messages. I am not going to implement a TCP/IP protocol stack from the ground up, but instead port an existing one.

In chapter 2 I will introduce the technical background relevant for this work. Chapter 3 discusses the design decisions I made regarding network support on M^3. Additionally certain implementation details are highlighted. Afterwards I evaluate the performance of the M^3 network stack, with Linux as a baseline (4). Then possible improvements for the future are discussed (5). The final chapter summarizes the work (6).

2 Technical Background

In this chapter I want to give an overview of the M^3 operating system and the hardware component it is building upon. Subsequently I introduce the gem5 simulator, which is the platform currently used to evaluate M^3. At the end I cover the basics of a network stack.

M^3 is a microkernel-based operating system developed by the TU Dresden Operating Systems Group [2]. It follows a novel approach to deal with the growing heterogeneity of recent computer systems, which combine many different compute units (CUs), such as general purpose cores, DSPs, FPGAs, and fixed-function accelerators, in a network-on-chip. The key idea is to place a hardware component, called data transfer unit (DTU), next to each CU, which abstracts from the CU's details and exposes a common interface. This allows M^3 to handle all CUs in an equal manner. The authors describe it as treating every CU as a "first-class citizen".

2.1 Data Transfer Unit

The combination of CU, DTU and memory, either a scratchpad or a cache, is called PE. All processing elements (PEs) of a system are connected via a network-on-chip. Communication between PEs can only happen through their DTUs. In particular, to access any external resource a CU must go through its DTU. For this purpose each DTU has a set of endpoints that it can use to communicate with other DTUs.

The CU can interact with its DTU via commands, which are scheduled by writing to some special registers of the DTU. The DTU's registers are accessible over memory mapped I/O.

The four most important commands supported by the DTU are send message, reply on message, read memory and write memory, which are handled by an endpoint of the respective type.

The DTU has three different endpoint types: send, receive and memory. A send endpoint can send messages to a receive endpoint. A receive endpoint stores the messages it receives into a buffer, located in its PE's scratchpad or cache. It also can reply to messages that it has received. A memory endpoint refers to a continuous range of memory, which can either be some kind of external memory or the internal memory of another PE. The state of each endpoint is represented by a set of DTU registers.

2.2 M^3

M^3 is microkernel-based in the sense, that the kernel includes only minimal functionality. Traditional microkernel operating systems exclude non vital functionalities from their kernel and instead run them in user space. Isolation between kernel mode and user space is achieved through CPU privilege levels. M^3 makes the same distinction, but instead of relying on CPU privilege levels to enforce isolation, it executes every process on a separate PE.

This guarantees isolation, because only a privileged PE can configure DTU endpoints and the only privileged PE is the one hosting M^3's kernel. Therefore all communication channels between PEs are controlled by the kernel. That enables us to execute untrusted code even on CUs that have no support for multiple CPU privilege levels, which are typically only found on general purpose cores (x86, ARM, ...).

The kernel manages processes as so called virtual PEs (VPEs). A VPE is initially assigned to one PE, but migration to another PE at later point in time is possible. There is also support for multiplexing multiple VPEs on one PE.

M^3 uses capabilities to manage permissions. Each VPE has its own capability space. A capability is a reference to a kernel object with permissions attached. For example, a VPE has a capability for each of its endpoints, which it can use to request the kernel to configure an endpoint on its DTU. Capabilities can be exchanged with help of the kernel via system calls. System calls are implemented as messages sent over DTU endpoints.

Besides the kernel, M^3 comes with a set of services, which are not running on the kernel PE, but instead each one on its own user PE. One example being the in-memory file system m3fs. Each service registers itself by the kernel, which in turn allows other applications to start a session with the service. Over a session capabilities can be exchanged, for example allowing client and server to establish a send/receive endpoint pair, so that the client can send messages to the service.

Accompanied with M^3 is also the library libm3, which provides abstractions for commonly used functionality, such as communicating with the kernel and services, accessing files, and using the DTU.

Although M^3 was already tested on real hardware (Tomahawk 2/3), current development is done on the open source hardware simulator platform gem5, mainly because its more flexible than working with real hardware.

2.3 gem5

The gem5 project [3, 6] is a flexible and extensible open source simulation platform, supporting a wide range of instruction sets. It offers diverse CPU models, system execution modes and memory models, differing in simulation speed and accuracy. In *full system mode* gem5 simulates the entire hardware of a computer system (e.g. x86, ARM), making it possible to boot an unmodified Linux.

To execute M^3 on gem5 a DTU model has been created. In order to meet M^3's aspiration for heterogeneity, additionally a few hardware accelerators have been modeled.

gem5 already contains several network interface controller (NIC) models, one example being Intel's 8254x line, which I use in this work. Due to its popularity there are a lot of resources available for it.

2.4 Network Stack

The complexity of a communication network can be largely reduced, by organizing it as a stack of layers, where each layer builds upon the services provided by the one below it [10]. Each layer has a distinct task and provides a service to upper layers. A protocol is a convention that two entities in the same layer use to communicate with each other.

Over time there were many standardization efforts. The two most important and relevant until today are the OSI model and the TCP/IP model. Both follow a layered approach. Neither one was perfect, but both had their strengths and weaknesses. The OSI model with its clearly separated layers, and the TCP/IP model with its well thought out protocols.

Of the seven layers originally specified by the OSI model, the following showed to be relevant in practice. Figure 2.1 depicts the interaction between layers and nodes in a network driven by such a protocol stack.

Physical layer As the lowest layer, the physical layer transports raw bits over a physical transmission medium. Commonly used media are twisted pair cables, optical fibers and wireless electromagnetic waves.

Data link layer The data link layer transfers blocks of data, so called frames, between nodes connected directly via a physical medium. Errors that occur during transmission have to be detected and corrected if possible. In a broadcast network it has to do medium access control, to arbitrate the shared medium.

Network layer The network layer transports packets from a source to a destination node, whereby, in contrast to the data link layer, these two nodes do not have to be directly connected. Therefore the transfer of a packet can traverse multiple intermediate hops, where each one has to make a routing decision, to ultimately guide the packet to its final destination.

IP with its utility protocol ICMP is an exemplary implementation of a network layer protocol. Hosts are identified by so called IP addresses.

Transport layer The transport layer offers end-to-end communication between peers on the source and destination nodes (see Figure 2.1). It allows multiplexing by accompanying a node's network layer address with a port number, which is assigned to a peer (program) running on the node.

Depending on the utilized transport protocol this layer offers different services. Transport protocols can be roughly categorized into two categories: connection-oriented and connectionless.

A connection-oriented transport protocol (e.g. TCP) typically provides a reliable byte stream. Therefore it has to segment the byte stream into chunks fitting the network layer and preserve the order of transferred segments. Additionally it has to ensure reliability through error detection/correction, acknowledgement and retransmission. Another desirable service is flow control, which prevents a fast sender overflowing a slower receiver.

A connectionless transport protocol (e.g. UDP) transports individual messages, and generally has no delivery or ordering guarantees.

Application layer The application layer provides high-level application protocols, such as HTTP for the web or DNS as a utility protocol to resolve (human-friendly) host names.

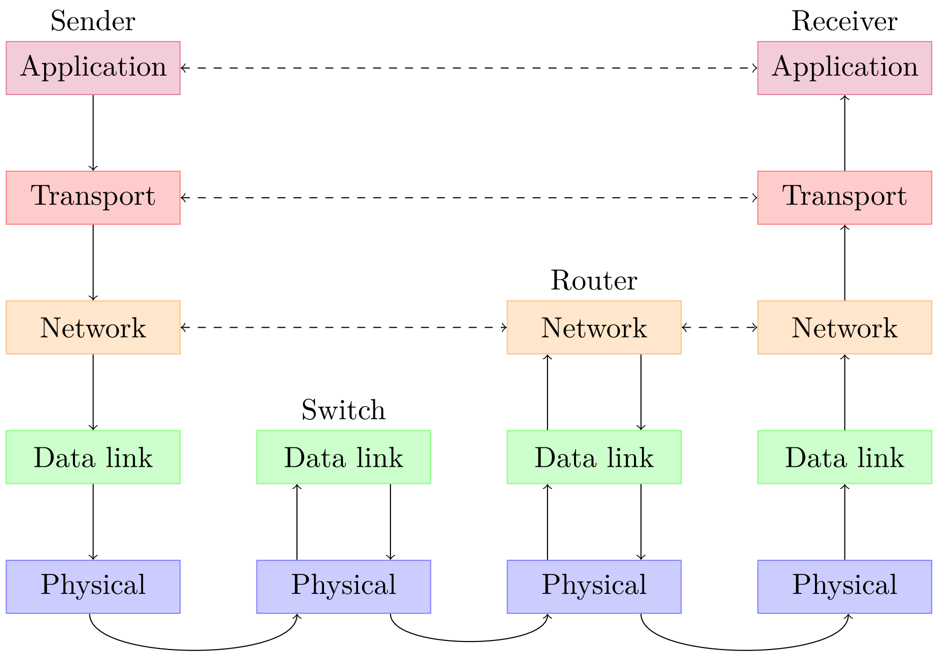

Figure 2.1: Layers of a network protocol stack.

The physical layer and the data link layer fall within the responsibility of the NIC. The network layer and the transport layer are typically implemented by the network stack of the operating system, but some NICs offer advanced optimization features, such as checksum offloading, that reach into upper layers. POSIX sockets [7] are a popular interface used by applications to interact with the operating system's network stack.

2.5 Peripheral Component Interconnect

Peripheral Component Interconnect (PCI) is a parallel bus originally developed by Intel in 1992, which in the meantime has undergone multiple revisions. It was intended as a successor for the Industry Standard Architecture bus, and has been ultimately superseded by PCI Express, which in contrast to PCI is a packet switching network, but backwards compatible to PCI at the software level. The PCI bus was mainly used to connect peripheral devices such as mice, keyboards, NICs or GPUs to the main system bus. Devices can be either circuits integrated on the motherboard or expansion cards.

Each device has a set of configuration-space registers, which are mapped into the configuration address space. The mapping address is calculated from the device's position on the bus. The configuration-space registers contain general information about the device, such as device ID and vendor ID, but are also used to configure how the device shall be made accessible in the system's port I/O address space or memory I/O address space.

For interrupt delivery PCI uses four dedicated interrupt lines, which are shared among all devices on the bus.

3 Design and Implementation

Integration of network interface controllers (NICs) should be coherent with M^3's policy of treating all compute units as first-class citizens. Therefore, the traditional approach of attaching the NIC to a general-purpose core, which simultaneously hosts the NIC's driver, is not a satisfying solution.

Instead I introduce a new kind of PE, a NIC processing element (NPE), whose solely purpose is to house a NIC. An NPE is integrated into the network-on-chip like every other PE. The NIC's driver then can run on an arbitrary PE, and use its DTU's communication mechanisms (message passing, shared memory) to configure and drive the NPE.

Next to the driver runs the network manager, which implements a network protocol stack and provides a socket-like API that can be used by applications to perform network communication.

When I talk about network in this work, I am assuming an Ethernet link layer with the TCP/IP protocol suite on top. However, the same principles would mostly also apply for a different composition.

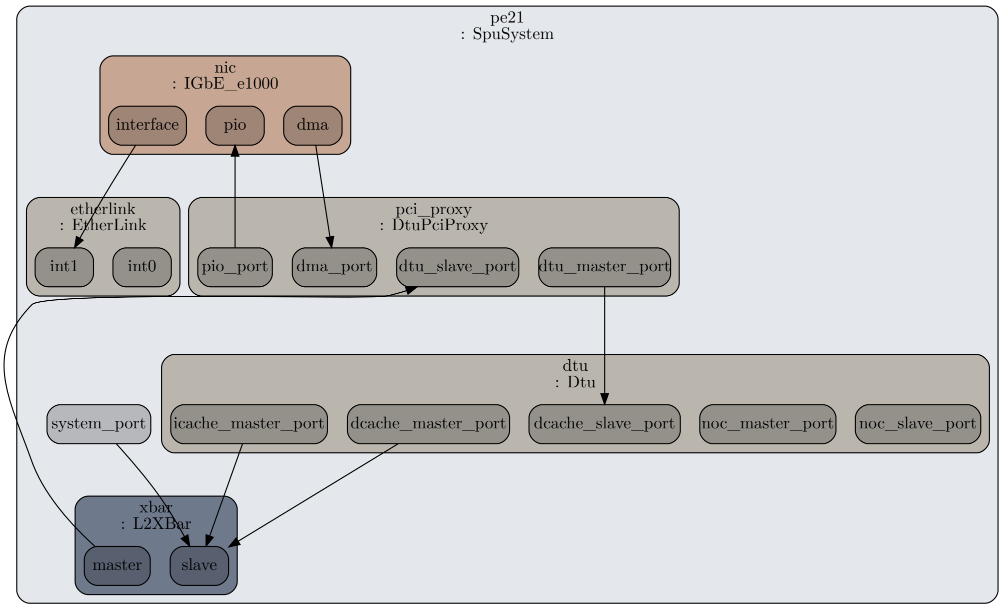

Figure 3.1: Structure of an NPE in gem5.

3.1 DTU-PCI-Proxy

PCI Express is the currently dominant technology to attach peripheral devices to a computer system. At the software level it is backwards compatible to its predecessor PCI. Fittingly, all the NIC models accompanied with gem5 are PCI devices.

One of them is build around the *Intel 82547GI Gigabit Ethernet Controller*, which is a member of Intel's *PCI/PCI-X Family of Gigabit Ethernet Controllers* [5]. Ethernet controllers from this family were mainly used in Intel's PRO/1000 network interface cards. I chose the NIC build around the *Intel 82547GI* over the other two available NIC models, because of its popularity and the thereof resulting availability of development resources.

The DTU has no means to directly attach a PCI device. Therefore a proxy component is needed to connect a PCI device to the DTU (Figure 3.1). This proxy component will be referred to as DPP).

A typical PCI device makes use of the following communication mechanisms.

Port-mapped I/O (PMIO) Port-mapped I/O uses a dedicated address space to access device registers. Often a special set of CPU instructions is designated to perform read and write operations in this address space. The x86 processor family offers the *in* and *out* instructions for this purpose.

Memory-mapped I/O (MMIO) Memory-mapped I/O, contrary to PMIO, uses the same address space to access device registers and main memory. Therefore no special instructions are necessary.

Direct memory access (DMA) Direct memory access allows a device to access main memory independently from the CPU. This has the benefit that in the meantime the CPU is able to perform other operations, whereas without DMA it would be occupied until the data transfer finishes.

Interrupts A device can signal an interrupt to its driver, to inform the driver of an event that requires its intervention (e.g. a network packet has been received). Interrupts themselves convey no content, so to determine the interrupts's cause the driver has to inspect the device.

The DPP has to translate these mechanisms into the primitives understood by the DTU. Translating MMIO is straightforward as the DTU offers means to export PE internal memory out of the box. The DPP maps the device's MMIO regions into its PE's address space. Currently this mapping is implemented as a static offset. A dynamic mapping, which is communicated to the device driver over a protocol implemented by the DPP, is conceivable. Information about the proxied device could also be made accessible through this protocol. For now the driver has no access to PCI internals such as the device's configuration space, and is therefore not able to identify the proxied device by e.g. its device ID. This becomes a problem when different devices are to be proxied in the same PE network-on-chip.

I will not bother with PMIO as MMIO is the preferable approach and also the one used by the *Intel 82547GI*. However, if the need arises to proxy a PCI device that uses PMIO, it should be relatively simple to add PMIO support to the DPP.

Interrupts are translated into messages (send commands) that are send to the driver PE via a preconfigured send endpoint. DMA is translated into DTU read and write commands, operating on a preconfigured memory endpoint. The configuration of the afore mentioned endpoints is carried out by the kernel on behalf of the device driver.

Both interrupts and DMA requests are translated into DTU commands. The DTU however can only execute one command at a time. Consequently an arbitration mechanism is required, which I implemented using the state machine depicted in Figure 3.2.

Depending on whether the state machine is in *Idle* state (DTU is currently not executing a command), interrupts and DMA requests received from the proxied device, are either translated into a DTU command and handed to the state machine for execution, or marked as pending. When in *Idle* state and instructed to execute a command, the state machine transitions into *Send* state and tells the DTU to execute the command, by writing to the respective DTU registers. Afterwards it transitions into *Wait* state, where it periodically checks if command execution has finished. Once the execution of a command finishes, the state machines looks for pending interrupts or DMA requests, and if so, subsequently executes them.

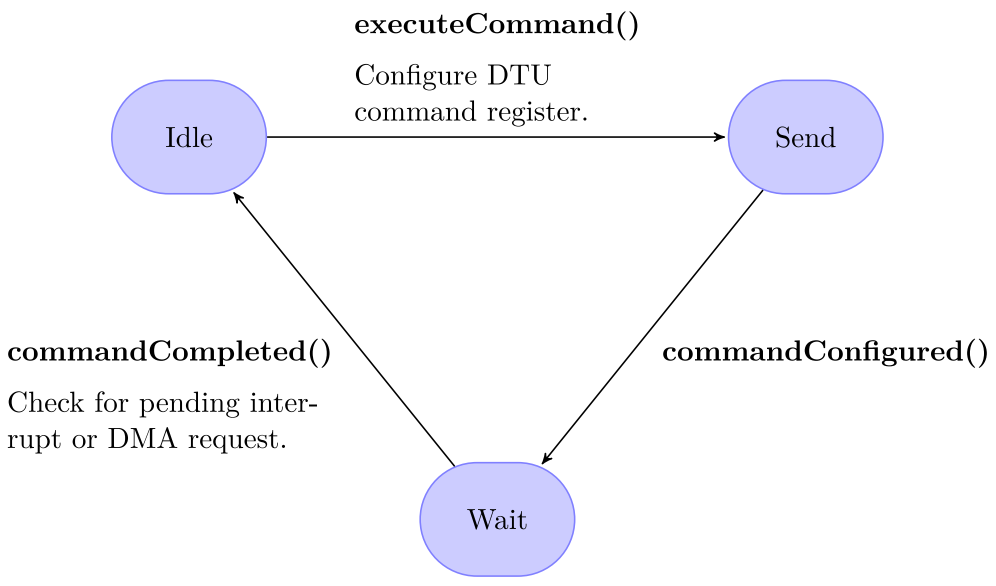
executeCommand()
Configure DTU command register.
Idle
Send
commandCompleted()
Check for pending interrupt or DMA request.
commandConfigured()
Wait

Figure 3.2: Command execution state machine.

3.2 Driver

3.2.1 Design

An NPE driver is a regular M^3 application that is responsible for operating the NIC of a NPE. The driver requests a NPE from the M^3 kernel, which results in the kernel creating a VPE of the PE type NPE, and giving the driver a capability for this VPE. Alternatively the driver could also receive a capability for a NPE by other means.

Subsequently the driver configures a memory and a send endpoint on the NPE. The memory endpoint is configured for read and write access to a memory region, allocated by the driver, that contains all the shared data between driver and NIC. The send endpoint is used by the NPE to signal interrupts to the driver, which listens on the corresponding receive endpoint.

As the creator of the NPE, the driver has a capability for the NPE and therefore can access its internal memory. This enables the driver to access the NIC's registers through DTU read/write commands, targeting the NPE's address space.

3.2.2 Implementation

Implementation of a driver is naturally highly dependent of the interface exposed by the driven device. Therefore I want to give a outline of a basic driver for Intel's *PCI/PCI-X Family of Gigabit Ethernet Controllers*, which roughly resembles the driver I implemented as part of this work. The controller specific part of the driver is inspired by Escape's [1] Intel e1000 driver.

Packet reception is coordinated through so called *receive descriptors*, which are organized in a ring structure (circular buffer). Such a descriptor contains the address of a data buffer, which stores the received packet, and fields to store additional packet information. The same applies to packet transmission.

Initialization of the Ethernet controller roughly consists of the following steps. Firstly the driver triggers a reset of the Ethernet controller, to ensure that all registers are set to there power-on reset values. Then it enables or disables features of the Ethernet controller, for example auto-speed detection. Next the driver sets up the transmit and receive descriptor ring, by allocating shared memory for the descriptors and their data buffers. Subsequently it sets the Ethernet controller's receive and transmit descriptor related registers accordingly. The driver also has to invalidate the receive descriptors, so that they can be used by the Ethernet controller.

When the driver is instructed to transmit a packet, it looks for the next free transmit descriptor. If successful, it writes the packet into the descriptor's data buffer and sets the descriptor's length field accordingly. Finally the driver transfers "ownership" of the descriptor to the Ethernet controller, by advancing the head pointer of the transmit descriptor ring.

The Ethernet controller offers timer-based and descriptor-based receive-related interrupts. When the driver receives an interrupt it checks the *interrupt cause register* to determine the cause of the interrupt. If the interrupt signals that packets have been received, the driver looks for valid receive descriptors and reads the received packets from their data buffers. Finally the driver invalidates the receive descriptors, and transfers ownership back to the Ethernet controller, by advancing the head pointer of the receive descriptor ring.

3.3 Network Manager

3.3.1 Design

The network manager functions as a glue between NIC drivers and applications that want to perform network communication. It interacts with the drivers and provides an interface to applications for sending and receiving messages over the network. Moreover, the network manager is responsible for the configuration of its network interfaces and has to deal with the utility protocols needed to participate in a network (e.g. ARP and ICMP).

To begin with I had to make two design decisions regarding the overall structure of the network manager. The seemingly simplest approach to implement a network manager is, to let each application have its own network manager. This simplifies the implementation, because an application can directly use its network manager's native interfaces, without message passing in between. Additionally, the absence of message passing should be beneficial in terms of performance. However, this design has the major drawback that a NIC has to be exclusively assigned to one application. That is so, because the network manager is the entity that evaluates network and transport layer address information, in order to provide multiplexing functionality. But in environments, where network and transport layer multiplexing among applications is not required, for example when VLANs [8] are used to partition the data link layer, this design could prove useful.

Another design is to share a network manager among multiple applications. Using M^3's *session* abstraction, which allows establishing a message passing channel between a client and a server (service provider), the network manager provides its services to applications. In contrast to the first design, this has the benefit that a NIC can used by multiple applications at the same time.

A less fundamental trade off is, whether the NIC driver should live in a different protection domain than the network manager. I had initially planned to let the driver run on a separate VPE, but in the course of implementing the network manager it became apparent that this would complicate matters unnecessarily. Running the driver on its own VPE leads to better isolation between network manager and driver. In case of a faulty driver this could improve error recovery ability and would limit the damage a malicious agent can cause to the VPE of the driver. On the other hand it would certainly deteriorate performance due to the additional layer of message passing between driver and network manager. Furthermore, not having to deal with the uncertainties and delays of message passing, simplifies the implementation of both components considerably. Therefore I decided to let driver and network manager run on the same VPE.

3.3.2 Implementation

The major part of the network manager is the network stack. That also includes the whole TCP/IP suite, with protocols such as IP, UDP, TCP, and utility protocols such as ARP and ICMP. I could have implemented a network stack from ground up, but that would have been a task in itself, and certainly out of scope for this work. Hence I decided to port an existing network stack to M^3.

My choice fell on the lwIP project [9], which is a lightweight and highly configurable open source implementation of the TCP/IP protocol suite. Fortunately its license (Modified BSD) is compatible to M^3's license.

lwIP can operate in two different modes, namely *mainloop* mode and *OS* mode. When running in mainloop mode, lwIP and application code are executed in the same thread, inside a loop. In OS mode, lwIP uses a distinct TCP/IP thread that communicates with the application threads via mailboxes.

The only requirement for mainloop mode is a time provider. As the system M^3 is currently evaluated on has no dedicated hardware timers, I had to add a new CLOCK register to the DTU. In combination with the DTU's TSC (timestamp counter) register, it can be used to measure time, and can therefore act as time provider for lwIP. OS mode demands further abstractions, namely threads, semaphores, mutexes, and mailboxes.

lwIP offers three different APIs. Firstly the *raw* API, which is an event-driven callback-style API, and the lowest-level interface to lwIP. The *sequential* API is implemented on top of the raw API. It is built around blocking open-read-write-close calls, similar to POSIX sockets [7]. The sequential API is only available when lwIP is running in OS mode. Finally lwIP has a POSIX sockets compatibility API, built on top of the sequential API, that aims to ease porting of existing applications.

The network manager does not need the high level functionality provided by lwIP's sequential API. Moreover, using the sequential API implies running lwIP in OS mode, and with libm3's non-preemptive thread model it would be difficult to provide all of the required abstractions. Last but not least the official lwIP documentation states that the sequential API has a significant overhead in comparison to the raw API. I was later able to confirm this claim in the benchmarks that I have done for evaluation 4. That is because applications and the lwIP TCP/IP core run in different threads, which requires synchronization and adds a layer of indirection.

Hence the network manager runs lwIP in mainloop mode and uses only the raw API.

I want illustrate the interaction of lwIP, driver, and application interface, by examining the path an incoming packet takes until it eventually reaches its destination. The network manager takes an incoming Ethernet packet from the driver and feeds it into lwIP. Internally lwIP processes the packet layer by layer. The processing action is highly dependent on the received packet.

The packet can be intended for the network manager itself, one example being the Address Resolution Protocol (ARP), which is a link layer utility protocol that given a network layer address (e.g. an IP address) attempts to discover the associated link layer address (e.g. a MAC address). lwIP kindly handles this case for us.

Secondly the packet can be addressed (on network layer level) not to the local machine, but instead to another participant of the network. If the network manager functions as a router, it will try to forward the packet in the direction of its ultimate destination. Otherwise it will drop the packet. lwIP has routing capabilities, but they were not put to the test in this work.

Thirdly the packet can be intended for a local application (on transport layer level). In this case the network manager has, on behalf of an application, created a socket by lwIP in advance and has registered a receive callback. When lwIP receives a packet addressed to a socket, it calls the callback registered for this socket, which will in turn pass the payload of the packet on to the application.

If neither of the above cases apply, the network manager will drop the packet.

A Packet submitted to the network manager by a local application, is handed to lwIP using one of its socket send functions. lwIP brings down/encapsulates the layer 4 packet into an Ethernet packet, and then hands it over to the driver.

It is also worth mentioning the case where a network manager employs multiple NICs. Even though I have not implemented it for now, extending the network manager by this feature should only require few changes. However, by concentrating multiple NICs to one network manager, it runs the risk to become a bottleneck. With multiple network interfaces at its disposal, the network manager can make use of lwIP's routing functionality, to act as a router.

There is one edge case that will bring the network manager in its current implementation to a temporary halt. M^3 supports multiplexing of a PE by the means of VPE context switching. Thereby, the kernel stores the state of a VPE in global memory, and assigns another VPE to the PE that the now suspended VPE was running on. At a later point in time the execution of the suspended VPE can be continued on the same or on another compatible PE. When VPE context switching is enabled and the network manager sends a message to a receive endpoint of a VPE that is currently suspended, the send operation will fail. The network manager then has to make a *forwardmsg* system call to the kernel, which will try to resume the target VPE and deliver the message. For the duration of the system call the network manager will be blocked.

Most of M^3's services use the workloop abstraction provided by libm3 to avoid blocking in this situation. A workloop consists of a set of worker threads, whereby all threads execute the same set of work items in a loop. When one of the threads blocks (e.g. waiting for an upcall from the kernel), another thread can serve the workloop. This works great for applications explicitly designed to run in a workloop. However, lwIP was not designed with this style of operation in mind, as it uses only a single TCP/IP thread. Therefore, lwIP potentially could break, when a callback blocks its TCP/IP thread, and lwIP is re-entered in another workloop thread. In order to be able to make a reliable statement about lwIP's behavior in a workloop environment, a thorough analysis would be necessary to cover all edge cases. But I assume that it should be possible to make lwIP workloop compatible.

3.4 Application interface (Socket API)

The network manager exposes a socket-like interface to applications. Therefore the network manager registers itself by the kernel as a service, with which applications can start a session. Once a session is established, the client obtains a capability for a send endpoint from the service. Over this send endpoint the client can invoke the following operations on the network manager.

`create(type, protocol)`
: The `create` operation creates a socket of the specified `type`, which can be either a stream socket (TCP), a datagram socket (UDP) or a raw socket. For raw sockets, which are sockets that allow direct sending and receiving of IP packets, additionally an IP `protocol` number has to be specified.

  On success `create` returns a socket descriptor that can be used to identify the socket in further operations.

  As the network stack is still in a proof-of-concept state, currently only datagram sockets (UDP) are implemented, but the API is designed to support also stream and raw sockets.

`bind(sd, address, port)`
: The `bind` operation associates the socket `sd` with a local socket address, which is the combination of an IP `address` and a `port`. An exception are raw sockets, where the socket address consists only of an IP address.

  This socket address can be used by a foreign node to connect to a stream socket or send messages to a datagram socket.

`listen(sd)`
: The `listen` operation sets the stream socket `sd` into listen state, which allows it to accept incoming connections.

`connect(sd, address, port)`
: The `connect` operation associates the socket `sd` with a remote socket address, which is the combination of an IP `address` and a `port`.

  Associating a stream socket with a remote socket means establishing a connection with the remote socket. Whereas for a connectionless socket, such as a datagram or a raw socket, no connection is established, but instead the remote socket is set as the default destination.

`close(sd)`
: The `close` operations closes the socket `sd` and frees all of its resources.

When an application asks the network manager to create a socket, the network manager creates a corresponding lwIP socket and internally maps it to a socket descriptor, which is returned to and subsequently used by the application to refer to the socket. For all of the remaining meta operations, namely `bind`, `listen`, `connect`, and `close`, the network manager looks up the lwIP socket for the specified socket descriptor and then executes the corresponding lwIP function on the socket.

Applications use a shared memory buffer to send and receive data over a socket, because the amount of data would be too large to be directly transferred over the send endpoint. A slightly modified version of libm3's *direct pipe* abstraction, which provides a uni-directional pipe between two PEs, can be used for this purpose. As the pipe is uni-directional, two pipes, one for receiving and one for sending, are required. They are established at the time of the creation of a socket. On top of these two pipes the `send` and `recv` operations are implemented, which are each available in a blocking and a non-blocking variant.

4 Evaluation

The two fundamental measures for the performance of a network stack are latency and throughput. Based on these measures, I evaluated the network stack that I have implemented for M^3. Linux served as a baseline, because it is a popular and well-optimized operating system.

4.1 Evaluation Platform

The M^3 system I used for evaluation consists of the following components. Multiple PEs each containing a x86_64 out-of-order core, clocked at 3 GHz. As physical memory 1 GiB of gem5's *DDR3_1600_8x8* model, clocked at 1 GHz. Two NPEs each containing a *Intel 82547GI* NIC. The two NICs are directly connected to each other via an Gigabit Ethernet link.

Because Linux runs on gem5, I could perform all my benchmarking on gem5. This has the advantage that Linux uses the same *Intel 82547GI* NIC model as M^3. It also uses the same x86_64 out-of-order core and physical memory model.

To execute Linux on gem5, I used a benchmark infrastructure featuring a Linux 4.10 kernel, which was previously used for benchmarking by the authors of M^3. The Linux system is built using Buildroot [4], which allowed for an easy integration of the Intel e1000 driver and network utility programs. Integration of the NICs into Linux posed a challenge, because gem5 contained no example config for attaching a PCI device to the system. Moreover, two small modifications of gem5 were necessary to get it working.

I experimented running Linux with different amount of cores. Two cores turned out to be a good compromise between benchmark performance and system stability. Especially when being operated with multiple cores, Linux tended to sporadically produce segmentation faults and kernel panics. This instability, in combination with the long simulation times, made benchmarking on Linux rather tedious.

Because of the long simulation times, I used representative, but short-running benchmarks.

4.2 Benchmarks

The latency as well as the throughput benchmark use UDP as transport protocol, because the proof-of-concept socket API on M^3 only supports UDP for now. The benchmark setup consists of a client and a echo server. The echo server responds to incoming messages with a message containing exactly the same payload as the incoming message.

On M^3 I implemented two variants of the benchmarks. For each NIC a network manager runs on one of the x86_64 cores. The first variant uses the socket API provided by the network manager (*m3*). The second variant runs inside the network manager and interfaces directly with lwIP (*m3-net*). The latter I implemented mainly to measure how much of an overhead the socket API entails.

On Linux I implemented three different variants of the benchmarks. The first variant uses a Linux datagram socket (*linux*). As Linux will not answer to ARP requests for local IP addresses, two directly connected network interfaces (NICs) can not communicate with each other. A solution is to put the network interfaces into different network namespaces. A network namespace is in essence a virtual network stack, with its own routes and network devices. The second variant uses lwIP's sequential API (*linux-lwip-seq*). To place lwIP as close as possible to the NIC, I implemented a network interface for lwIP that uses a Linux raw socket to send and receive Ethernet frames. Both client and server use their own instance of lwIP. As the performance results with lwIP's sequential API were rather underwhelming, and even the official documentation states that the sequential API has a significant overhead in comparison to the raw API, I decided to implement a third variant using lwIP's raw API (*linux-lwip-raw*).

4.2.1 Latency

I measured the the round-trip time of an UDP datagram sent to the echo server and tested the effect of varying payload sizes. Per size fifteen samples were collected, outliers were removed, and the mean values calculated.

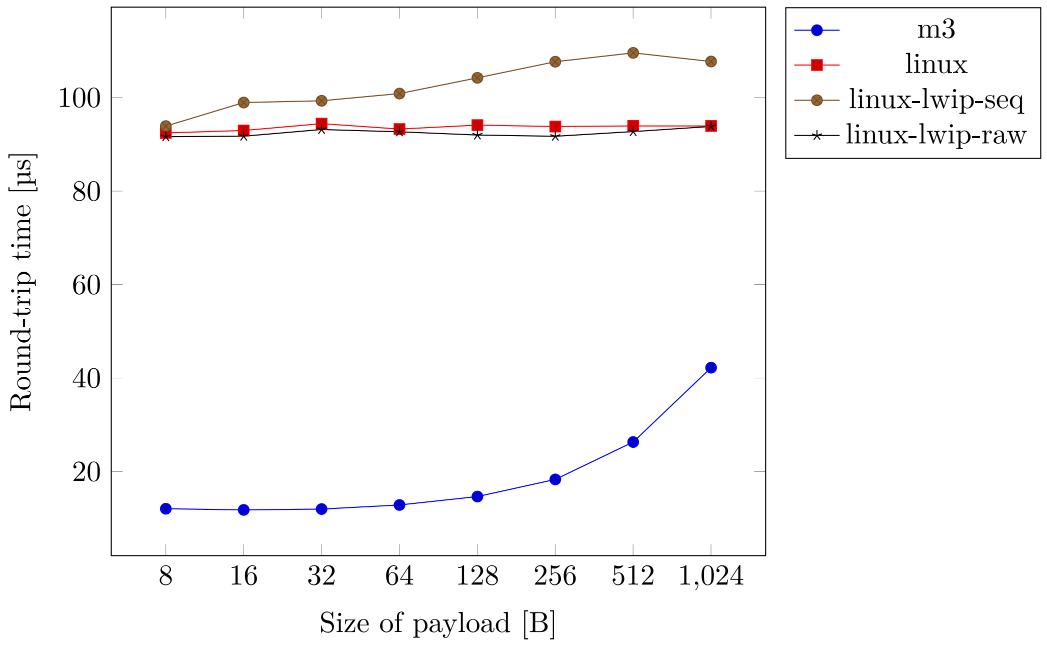

Figure 4.1: Comparison of round-trip times.

As can been seen in Figure 4.1, the measured round-trip times on M^3 are considerably shorter than on Linux. On M^3 the round-trip time is correlated with the payload size, whereas on Linux no such correlation is observable. After a bit of investigation I figured out that the longer round-trip times on Linux are caused by an optimization used by its e1000 driver. The Linux e1000 driver uses the NIC's interrupt throttling feature [5, p. 309], which allows it to steer the interrupt rate, by configuring a guaranteed inter-interrupt delay between interrupts triggered by the NIC. When a system has to handle high packet rates, throttling the interrupt rate maximizes throughput, because less time has to be spent on handling interrupts and more time can be spent on packet processing. Per default the Linux e1000 driver uses the interrupt throttling mechanism in a dynamic mode, which adapts the inter-interrupt delay to the current data rate. The values I observed during my benchmarks showed that an interrupt could be delayed by as much as 50 µs.

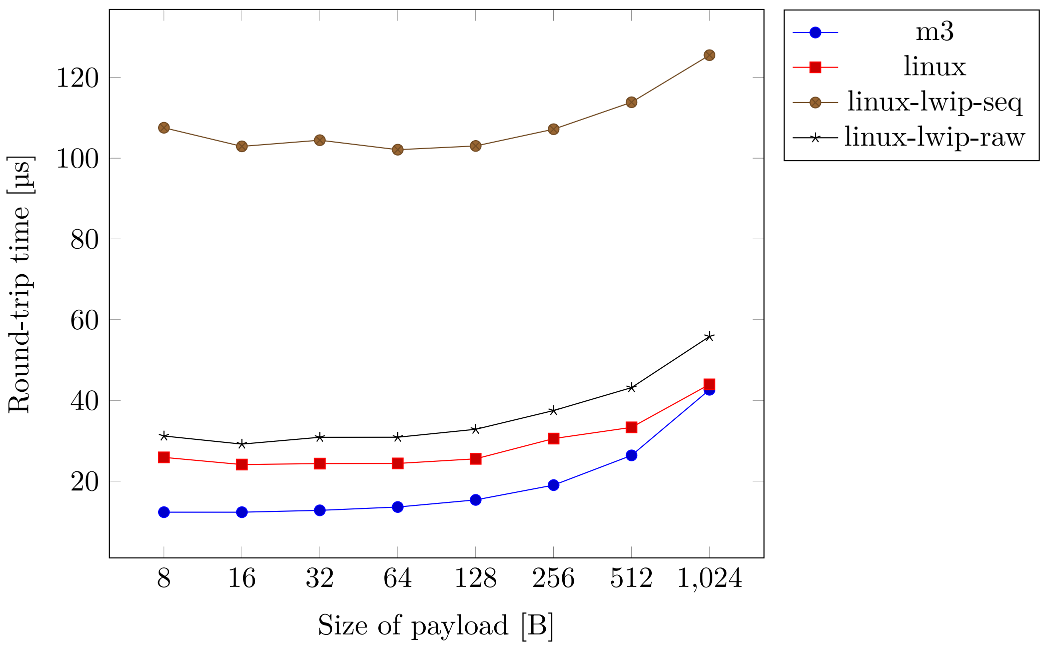

Figure 4.2: Comparison of round-trip times with interrupt throttling disabled.

Therefore, to make the results more comparable to M^3, whose NIC driver does not use such optimizations, I disabled the interrupt throttling feature in the Linux e1000 driver (Figure 4.2). As expected the gap between M^3 and Linux shrunk considerably. Also the round-trip times on Linux now show a correlation with the payload size, which is to be expected alone due to the increased transfer duration over the wire. The remaining overhead observable on Linux comes from time spent in the kernel.

The lwIP sequential benchmark did not improve at all. I suspect this is caused by scheduling and the four additional context switches it needs, because the application (client/server) and the lwIP TCP/IP core run in separate threads.

The round trip-times on M^3 increases more with payload size than on Linux. To measure the influence of the socket API application interface, I reimplemented the benchmark directly in the network manager. As can be seen in Figure 4.3, for both benchmark setups most of the increased round-trip time comes from the increased transfer duration over the wire. However, the remaining part has to have other causes. So I analyzed gem5's debug output and was able to identify checksum calculation, DMA, and the application interface as the other causes. The detailed distribution is illustrated in Figure 4.4.

The calculation and verification of checksums is currently done by lwIP in software, but it could be offloaded to the NIC. DMA transfers are split into multiple cache-line-sized chunks by the DMA controller. Therefore, for a larger packet more chunks have to be transferred. Because the DTU-PCI-Proxy (DPP) translates DMA requests into DTU commands, and the DTU can only execute one command at time, all the chunks have to be processed sequentially. In contrast on Linux, which uses conventional DMA, all request chunks are send out, before the first response even arrives. This explains the strongly size dependent DMA performance on M^3. A possible solution would be to design a different type of DTU interface for NPEs that allows a more efficient DMA request processing.

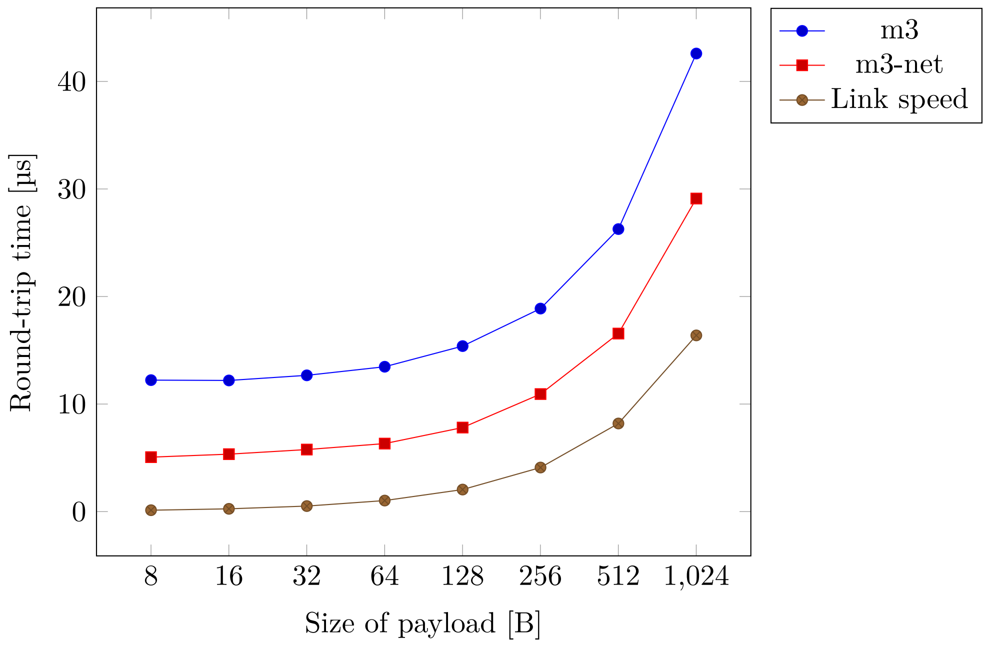

Figure 4.3: Comparison of M^3 round-trip times.

4.2.2 Throughput

To measure the throughput the client sends a fixed amount of UDP datagrams, each with 1024 bytes of payload, to the echo server. It measures the time between sending the first packet and receiving the response for the last packet. From the difference of these two points in time and the amount of transferred payload bytes, the goodput is

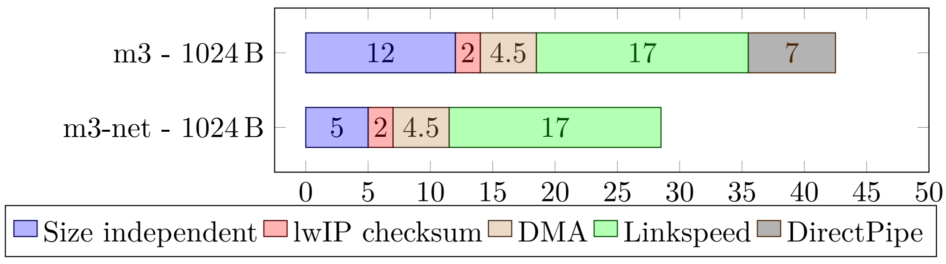

Figure 4.4: Payload size dependent increase of round-trip time on M^3, between 8 B and 1024 B of payload.

calculated. By adding the overhead of UDP, IP, and Ethernet headers to the goodput, I get the throughput, which is the data rate that goes over the wire.

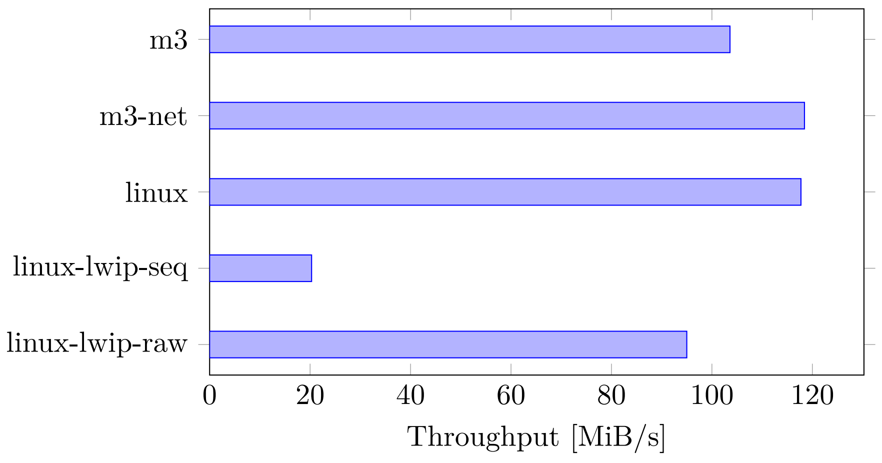

Figure 4.5: Comparison of throughput between Linux and M^3 in different configurations.

As can be seen in Figure 4.5, both *linux* and *m3-net* reach wirespeed. *m3* is about 15 % below wirespeed. The reason for this is that the network manager, which has to communicate with the NIC and the client at the same time, uses more endpoints than its DTU can provide. Thus, the network manager has to multiplex the endpoints of its DTU, by “overwriting” endpoints it currently does not need. Each of these endpoint switches requires a costly system call, because only the M^3 kernel can configure endpoints. For further details on endpoint depletion see chapter 5.

Unexpectedly *linux-lwip-raw* is not able to reach wirespeed, although *m3-net* also uses the the lwIP raw API. I measured the time that the send() function call on the underlying raw socket needs before it returns. The results show quite a few spikes, which could be the cause for the inferior performance. I was not able to identify the reason these spikes occur with the raw socket, but not with the datagram socket used by *linux*.

5 Future Work

The network manager for now only offers a basic set of functionalities. It lacks support for IPv6 and is unable to handle more than one NIC. Moreover, the socket API is designed to support stream, datagram, and raw sockets, but currently only datagram sockets are implemented. Furthermore, advanced NIC features, such as checksum offloading and interrupt generation features, which would improve performance in certain situations, are not utilized. All of these functionalities should be relatively easy to implement.

The network interfaces of the network manager are currently configured via parameters passed to network manager on startup. Instead it would be nicer, if the network interfaces, and also other settings, could be configured during run time, via an API provided by the network manager.

lwIP comes with basic routing capabilities. Evaluating how these can be integrated into the network manager could be an interesting task.

The data transfer mechanism currently implemented by the socket API which uses two uni-directional direct pipes per socket to transfer data. It consumes a huge amount of endpoints, three on each side of a pipe. This leads to depletion of the network mangers endpoints, when multiple applications want to interface with it, because endpoints are a highly limited resource. At the time of writing a DTU had only 12 endpoints, although this limit was chosen rather arbitrarily. The optimal amount of endpoints has yet to be determined. Optimal in this case means minimal but sufficient. Deciding for a number of endpoints is a trade off, because more endpoints mean less chance of depletion, but consume more chip surface and energy. It is even conceivable that different types of PEs have a DTU with a different amount of endpoints. For example a powerful general purpose core will probably make more use of its endpoints, than a fixed function accelerator.

libm3 has basic endpoint switching capability, but with a few limitations, namely that receive endpoints can not be disabled and that send endpoints with outstanding credits (replies) lead to a crash when switched. Moreover, each endpoint switch requires a costly system call to the kernel. Therefore, an optimal solution would use only a minimal amount of endpoints. To save endpoints on the side of the network manager, the multiplexing capability of receive endpoints can be used. By assigning *labels* to corresponding send endpoints, a receive endpoint can distinguish among senders.

So in terms of endpoint consumption an improved socket API implementation could look as depicted in Figure 5.1. Unfortunately send endpoints have no multiplexing support, so the network manager needs one send endpoint per application. Of course this still scales considerably better than the current implementation, which needs three endpoints per socket.

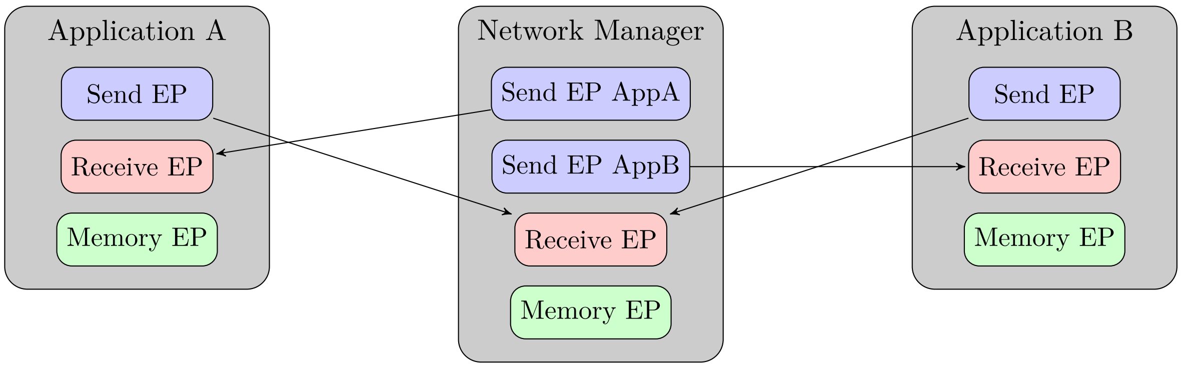

Figure 5.1: An improved socket API implementation. The network manager needs one receive endpoint, one memory endpoint and per application one send endpoint. An application needs one endpoint of every type.

6 Conclusion

In this work I designed and implemented network support for M^3. In section 3.1 I extended gem5, which is M^3's current evaluation platform, by a PCI proxy component that allows integration of PCI devices, such as the NIC models accompanied with gem5, as a separate PE into M^3's network-on-chip. The proxy component exposes its PCI device, through the communication primitives of its PE's DTU, to other PEs.

In section 3.2 I implemented an abstraction to interface with proxied PCI devices. Using this abstraction I ported an e1000 driver for the *Intel 82547GI* NIC accompanied with gem5.

The network manager implements a TCP/IP protocol stack, configures network interfaces for its NICs, and provides an interface to applications for sending and receiving messages over the network (3.3). To cover the first two tasks I ported the open source network stack lwIP to M^3. As application interface I implemented a socket-API service (3.4), which is designed to provide stream, datagram, and raw sockets, but currently only datagram sockets are implemented.

In direct comparison with a Linux 4.10 my network stack for M^3 proved to be superior in terms of round-trip time and competitive in terms of throughput, at least at Gigabit Ethernet speeds (4). As gem5 has no above Gigabit Ethernet NIC models, I was not able to test higher speeds.

Acronyms

CU	compute unit
DMA	direct memory access
DPP	DTU-PCI-Proxy
DTU	data transfer unit
M^3	microkernel-based system for heterogeneous manycores
NIC	network interface controller
NPE	PE housing a network interface controller
PCI	Peripheral Component Interconnect
PE	processing element
VPE	virtual processing element

Bibliography

[1] Nils Asmussen. *Escape. A UNIX-like microkernel operating system.* last checked: May 14, 2018. URL: `https://github.com/Nils-TUD/Escape`.

[2] Nils Asmussen et al. "M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores". In: *SIGPLAN Not.* 51.4 (Mar. 2016), pp. 189–203. ISSN: 0362-1340. DOI: `10.1145/2954679.2872371`. URL: `http://doi.acm.org/10.1145/2954679.2872371`.

[3] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: `10.1145/2024716.2024718`. URL: `http://doi.acm.org/10.1145/2024716.2024718`.

[4] *Buildroot. Making Embedded Linux Easy.* last checked: May 17, 2018. URL: `https://buildroot.org/`.

[5] Intel® Corporation. *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual.* Revision 4.0. March 2009.

[6] *gem5 - A modular platform for computer-system architecture research.* last checked: Apr. 16, 2018. URL: `http://www.gem5.org`.

[7] The Open Group. *The Open Group Base Specifications Issue 7, 2018 edition. General Information - Sockets.* last checked: May 19, 2018. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10`.

[8] "IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks". In: *IEEE Std 802.1Q-2011 (Revision of IEEE Std 802.1Q-2005)* (Aug. 2011), pp. 1–1365. DOI: `10.1109/IEEESTD.2011.6009146`.

[9] *lwIP - A Lightweight TCP/IP stack.* last checked: Mar. 13, 2018. URL: `https://savannah.nongnu.org/projects/lwip/`.

[10] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks.* 5th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010. ISBN: 0132126958, 9780132126953.