Master Thesis

Using modularity to scale a Multikernel network stack

Author(s):
Gerhard, David

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007343285

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 62

Systems Group, Department of Computer Science, ETH Zurich

Using modularity to scale a Multikernel network stack

by

David Gerhard

Supervised by
Pravin Shinde, Prof. Timothy Roscoe

June 1, 2012
Abstract

The recent addition of hardware features in network card make the networking subsystem more complex which makes it harder to benefit from the additional features. Furthermore the complexity is increased as an optimal solution for a one problem might not be beneficial for another. A proven concept to deal with complexity is modularity.

This thesis introduces netg, a modular and dynamic network stack for the Barrelicfish operating system. The protocol processing and the storage of connection data is done in a modular way to maximize flexibility. Netg allows for dynamic reconfiguration of the network stack based on available hardware features of the network card and the currently applied load.

In the evaluation we show that the modularity does not hinder the performance of a network stack and show some additional features which use the new flexibility to improve performance. A modular approach does not only lead to a more flexible system it is also easier to implement additional features.
# Contents

Contents

1 Introduction 1

2 Background 3
   2.1 Barrelfish 3
      2.1.1 CPU drivers 3
      2.1.2 Monitors 3
      2.1.3 System Knowledge Base (SKB) 4
      2.1.4 Application processes 4
      2.1.5 Communication 4
   2.2 Networking 5
      2.2.1 Barrelfish 5
      2.2.2 Other operating systems 7

3 Related Work 11
   3.1 Modularity 11
   3.2 Parallelization 12
   3.3 Buffer handling 13

4 Approach 15
   4.1 Network Graph Overview 15
      4.1.1 Endpoints 16
      4.1.2 Nodes 16
      4.1.3 Queue Nodes 18
      4.1.4 Communication between Nodes 19
      4.1.5 Scheduling 19
      4.1.6 Network Connection Information 19
   4.2 Network Graph Optimizations 19
      4.2.1 Global Optimizations 20
CONTENTS

4.2.2 Local Optimizations .................................. 20

5 Implementation ........................................... 21
  5.1 Services ............................................... 21
    5.1.1 Network Resource Manager (NRM) ............... 22
    5.1.2 Device Manager (NDM) .......................... 23
    5.1.3 Queue Manager .................................. 23
    5.1.4 Network daemon (netd) ......................... 24
    5.1.5 Dynamic starting ............................... 24
  5.2 Network Graph Structure ............................. 24
    5.2.1 Initialization .................................. 25
    5.2.2 Network Endpoints .............................. 25
    5.2.3 Buffer Management ............................. 27
    5.2.4 Network Graph Nodes ........................... 28
  5.3 Application Interface ................................ 30
  5.4 TCP/IP processing in the Network Graph ............ 31
    5.4.1 Ethernet & IP .................................. 31
    5.4.2 UDP ............................................ 32
    5.4.3 TCP ............................................ 33
    5.4.4 Resulting Graph ................................ 36
  5.5 Network graph optimizations ........................ 38
    5.5.1 Transmit buffer reuse ........................... 38
    5.5.2 Dynamic queue assignment ........................ 40
    5.5.3 Load balancing of TCP connections ............... 40

6 Evaluation ............................................... 43
  6.1 Network Graph Evaluation ........................... 43
    6.1.1 Latency per Node ................................ 43
    6.1.2 Total Latency for Packet Processing ............ 44
  6.2 Queue switching ..................................... 46
    6.2.1 Difference between Queue Configuration ........ 46
    6.2.2 NRM distributed Queue Configuration ........... 47
  6.3 Load Balancing of TCP Connections .................. 48
  6.4 Conclusion ........................................... 48

7 Conclusion ............................................... 51
  7.1 Future Work ......................................... 52

A Benchmark Hardware Appendix .......................... 53

B Network graph module list ................................ 55

Bibliography .............................................. 59
Chapter 1

Introduction

Current Operating Systems were designed at a time where computer were single core machines with relatively low memory capacity. While there has been a lot of work to extend these Operating Systems to scale with multi and many-core systems the focus of these optimizations were focused on computing performance. The same goes for the networking subsystem, which are more or less still using the same techniques. While the network subsystem for these Operation Systems was optimized to perform in multi- and many-core environment, they were only adapted and never really designed for it.

Memory consumption played an entirely different role because the computer had only very limited amount of memory available. So the network subsystem was designed to use as little memory as possible. Nowadays we nearly have an overabundance of memory and if not used excessively applications and operating systems can try to optimize performance by using the available memory.

Not only the hardware of the computer itself changed in the recent years but the Network Interface Cards (NIC) themselves have had dramatic changes. To reduce the amount of computation needed for the processing of network packets modern network cards can take over some steps in packet processing. One thing most recent network cards can do is calculate checksum for outgoing TCP or UDP packets and verify it for incoming packets, because the checksum computation can easily be done in hardware and is quite computation intensive on the CPU when there are millions of packets per second. Another feature of recent cards is to offload the TCP connection to the network card.

There are different mechanisms for doing it, in one called Large Segment Offload (LSO) the network stack to sends a large segment (e.g. 64kB) and a header template for the TCP, IP and data link layer to the network card,
which then breaks this buffer into smaller TCP segments and sends them to the destination without further intervention of the network stack. The same can be done on the receiving side, which is then called Large Receive Offload (LRO).

The faster network connections placed more stress on the packet processing in the network stack and without optimizations to the network stack it would not be possible to operate these NICs at line speed since access to the NIC has to be synchronized if the packets are processed on multiple cores. An optimization to parallelize the packet processing in Windows is called RSS [RSS]. While older NICs have only one transmit and one receive queue ring newer cards support multiple queue rings, the number of queue rings can vary from two to over a thousand. This enables parallel processing of network packets on multiple processor cores simultaneously without introducing synchronization overhead.

By placing these queue rings and their buffers in the same NUMA region as the packet processing CPU core one can reduce the load on the shared memory and reduce the latency of incoming and outgoing packets.

While this relieves the network stack to some extent, some network cards can do the complete TCP connection in hardware, so that the network stack is only responsible for the three-way connection handshake. After that the network stack transfers the responsibility to the network card by handing over the essential connection information (IP address, port, sequence number...).

On the software side changes are needed in order to fully use the new capabilities of these newer network cards. These changes to the network cards together with the changing in the computer hardware itself make it very hard to deal with this complexity since most network stacks are inflexible and highly optimized for complete packet processing in software. A network stack should be able to adapt to the hardware circumstances and be optimal for the current hardware situation.

This cannot be done without additional flexibility in the network subsystem and one way to achieve flexibility is by modularizing a process. Modular programming is a well known software design technique used to break a complex problem into simpler tasks, which then can be easier implemented and maintained. But not only the processing of the packets should be modular but the amount of globally stored information about networking related things should be minimized in order to reduce synchronization between different threads.

The usage of a uniform interface between the modules allows them to be exchanged or even the transformation of the way packets are being processed. Once the flexibility is in place the network stack can even adapt its structure to optimally deal with the current load situation.
2.1 Barrellfish

Barrellfish is a new research operating system being developed by researchers from ETH Zürich in collaboration with Microsoft Research. The main focus is on how to structure an OS for future multi- and many-core systems. Barrellfish introduced the multikernel architecture [BBD+09] in order to scale with the rapidly growing number of cores computer systems will have and to exploit possible heterogeneous hardware resources. Barrellfish achieves this by treating the computer itself as a distributed system and by using well known distributed systems and networking techniques.

2.1.1 CPU drivers

The CPU driver is responsible for handling the privileged operations on every core. This includes handling scheduling, the memory subsystem, low-level resource allocation and protection enforcement and authorization. For this the CPU driver runs in a privileged process on each core. The source code of the CPU driver is very CPU architecture dependent and highly optimized for each architecture. This has the advantage that the rest of Barrellfish can be written in a more architecture independent way.

2.1.2 Monitors

The Monitor is a user-process running on each core and implements much of the mechanism and policy normally found in the kernel of traditional OS. Global state data structures are stored in each monitor and replicated over messages and any requests that access the global state are handled by the monitors. Monitors are scheduled by the local CPU driver.

The Monitor is responsible for the setup of communication channels between user-space processes and perform the waking up of blocked local
2. Background

Table 2.1: Inter-Dispatcher Message Cost

<table>
<thead>
<tr>
<th>Type</th>
<th>CPU Cycles</th>
<th>µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMP</td>
<td>6277</td>
<td>2.5</td>
</tr>
<tr>
<td>UMP</td>
<td>753</td>
<td>0.3</td>
</tr>
</tbody>
</table>

processes in response to messages from other cores.

2.1.3 System Knowledge Base (SKB)

The System Knowledge Base is a service in Barreelfish which is used to store and retrieve information about the hardware of the system. It is populated from various information sources as the ACPI tables as well as the PCI configuration information. Barreelfish runs several online measurements on the hardware it is running on which allow more knowledge about the its topology and these informations are stored in the SKB too. An example for this is the latency measurement in the communication between all the cores which is used to build an optimal communication setup between different cores.

2.1.4 Application processes

An application process in Barreelfish differs from the typical process structure. A process is represented by a dispatcher on each core it runs on. These dispatchers are responsible for the communication between each other and are scheduled by the local CPU driver. Unlike traditional multiprocessor processes in Barreelfish processes are not migrated between cores by the OS.

2.1.5 Communication

Barreelfish is strongly focused on messages and therefore provides excellent support for inter-dispatcher communication (IDC) and supports intra-core as well as inter-core communication. For the x86_64 architecture exist two different type of communication channels. Intra-core communication is achieved by Local Message Passing (LMP), a Barreelfish specific implementation of LRPC [BALL90], which stores the message in hardware registers. Inter-core communication, called user-level message passing (UMP), is done by a similar implementation to user-level RPC (URPC) [BALL91] using shared memory.

The inter-core messages are performance wise cheaper than the intra-core messages because they do not involve a context switch and have no need for a TLB flush. Table 2.1 shows the latency cost of a single message using UMP and LMP on the sbrinz1 test machine (Appendix A).

Interfaces between processes are defined in an interface definition language called flounder and support asynchronous message passing as well as re-
mote procedure calls (RPC). From this interface the flounder stub compiler generates C-code for all available communication channels.

**Bulk transport mechanism**

The bulk transport mechanism is a bidirectional transport mechanism for larger messages. It is implemented as a library and uses a shared memory pool for the messages and a meta-data queue per direction. The memory pool is split into equal sized chunks and is referenced by an index and by virtual addresses. The index of a sent message is delivered using the IDC mechanism and the index is sent back when the buffer is no longer used.

Currently the bulk transfer mechanism does not allow the destination endpoint to read ahead even if messages were available.

### 2.2 Networking

The following sections provide an overview of the state of the network subsystem in Barrelfish prior to this thesis and the current state of commercially used operating systems.

#### 2.2.1 Barrelfish

**Overview**

The current network stack in Barrelfish consist of three software parts and the network card, as shown in figure 2.1. The network card is controlled by the device driver (e.g. e1000 driver) which is linked together with the multiplexing and demultiplexing handling etherserv library. Port management as well as DHCP is handled by the network daemon (netd), which links to the lwIP [Dun03] library. The user application links to the lwIP library as well. These components are explained in further detail in the following chapters.

**Network card driver**

The network card driver is responsible for managing the NIC device and allocates transmit and receive ring buffers and performs the dequeuing and enqueuing of packets from these buffers. The etherserv library, to which every NIC driver links to, exports two IDC interfaces to other applications. The ether.if is used by user application for sending and receive packets from the network device. The ether.control.if is used by the network daemon to inform the driver about changes to the packet filter.

Received packets are passed on to the etherserv library, which first of all handles IP fragmentation. If the packet is reassembled or not fragmented, it is passed on to the packet filter, which is implemented in the bfdmuxvm
2. Background

![Diagram: Overview of existing networking architecture]

library and accessed over the bfdmuxtools library. The packet filter resolves the packet to a bulk-transfer buffer from a user application and the etherserv library copies the packet into this buffer and sends it to the user application by the bulk-transfer mechanism of Barrelfish.

ARP packets and packets with no corresponding user application buffer are sent to the network daemon by the same bulk-transfer mechanism.

**Network Daemon (netd)**

The network daemon manages TCP and UDP ports and is responsible for downloading the filters to the etherserv in the NIC driver. To handle the ARP, DHCP and ICMP protocol, the network daemon links to the lwIP library.

User Application connect to the network daemon over the `netd.if` interface. The network daemon itself connects to the network driver over the `ether_control.if` for filter related updates and two `ether.if` connections to the etherserv library of the network driver.
User Application

To get access to the network any user application has to link to the lwIP library and initialize it with a NIC driver name. The lwIP opens an idc connection to the network daemon and two idc connection to the network driver. Afterwards lwIP allocates a receive and a transmit buffer and initializes the pbuf’s in these buffers. Upon binding a port to a protocol control block (PCB), lwIP sends a bind request to the network daemon containing the port, protocol, receive and transmit buffer. These are the bulk-transfer buffers to which the packet filter in the network driver resolves a packet to.

A received network packet gets handed of to the lwIP network stack, where its IP and TCP or UDP headers are processed and ultimately the receive callback of the user application is called with the data buffer of the received packet.

2.2.2 Other operating systems

In this section we provide an overview of how other operating systems network architecture is structured.

FreeBSD

The FreeBSD’s network stack is a descendant of the original reference TCP/IP implementation from the University of California at Berkeley and is explained in “TCP/IP Illustrated Vol. 2” [SW95] in detail. The stack is situated in the kernel and uses mbufs to store network packets in. The mbuf is a mechanism to cope with the demands of the network protocols. These include buffer manipulation of various sizes, prepending and appending data as well as removing data from the buffer. Some of these features are handled by chaining mbufs together.

Packets arriving from the network are put into mbufs and then processed by the network protocols. These protocols are implemented in a inflexible but highly optimized way to reduce copying and maximize performance. Connection information is stored globally and therefore in multi-core system any operation on these information has to be protected by a lock, which reduces the scalability. Since features in the protocol processing are mostly done in a corresponding function but the interface between these functions is not uniform the processing path of a packet cannot be changed easily and features offloaded to the hardware.

Linux

The Linux network stack originated from the FreeBSD implementation with the exception that instead of mbufs Linux uses sk_buffs and tries to fit a
2. Background

packet completely into one buffer. If not possible the whole data is copied into a new sk_buf which is big enough for the packet. While this is faster it wastes more memory for smaller packets. The sk_buf corresponding to a packet is allocated when the packet arrives. If hardware interrupts are used this is done in the hardware interrupt handler and then the sk_buf is enqueued since interrupt handler code should be as small as possible and the soft interrupt corresponding to the packet handler is activated. The soft interrupt handler then takes care of processing the packet. These allocations are expensive and slow the processing of packets down.

Linux is able to use multiple receive or transmit queues on the network card and process packets on different cores which is used to balance the load and distribute interrupts among the cores. But the data is still stored globally and has to be protected by locks. And the network stack is still accessed over a single queue into which the multiple network card queues are merged which results in a bottleneck. This is because the user application on Linux can be scheduled on each core and have to be able to read a network socket on every core and the read has to be in-order for a TCP connection.

Furthermore Linus Torvalds said in an interview that Linux should not support protocol feature offload to the network card since the firmware on the network card is proprietary and do not have an uniform interface over which they can be used across cards from different vendors. He said also that the right place to do protocol processing is in the OS and not on the card.

For these reasons multiple network card vendors provide custom Linux kernel images or patches to enable Linux to use the hardware features.

Windows

Microsoft Windows introduced with Windows Vista a new network stack “Next Generation TCP/IP Stack” [MS0], which provides support for Receive-Side Scaling(RSS) [RSS] and a new mechanism for TCP protocol stack offload to hardware (TCP Chimney Offload). On most hardware only the checksum calculation is offloaded to the network device, but if the network device supports complete TCP offload, then the TCP connections get offloaded after the TCP Handshake. Most hardware offloading is unfortunately disabled by default for consumer Windows versions.

The new stack is more modular and introduced more flexibility to extend the network stack with modules, which can be dynamically inserted or removed. The new network stack also supports NetDMA, which uses DMA to allow processors to be freed from moving data from network card data buffers and application buffers, but is only supported if the Intel I/O Acceleration is enabled.
We believe that Microsoft chose a more modular network stack in order to attain the needed flexibility for features like TCP offloading.
Chapter 3

Related Work

The usage of modularity in the network related area is not a new concept and has been researched before. In recent research it has been used mostly to build configurable software routers and getting them to scale in a many- and multi-core environment. Another area currently being researched the optimal exploitation of the current network cards hardware and maximize the performance by using parallelization and better buffer management systems. In this chapter we present an overview of the existing research related to this thesis.

3.1 Modularity

The x-Kernel [HP91] is a framework for implementing and composing network protocols at connection endpoints where packets are delivered to user space application. An x-kernel configuration is a graph of processing nodes and packets are passed between nodes using a uniform interface. This configuration graphs are always acyclic and layered as the nodes are intended to represent parts of the protocol stack. Connection between the nodes is bidirectional and packets pass alternatively between “protocol” nodes and “session” nodes, where the session nodes correspond to end-to-end connections like TCP sessions.

In an additional paper [OP92] they use the x-kernel framework to try to answer the question of how a protocol graph for a given protocol should look like or in other words, how many layers a protocol needs. To enable a more dynamic network architecture they break existing protocols apart into small reusable microprotocols, which implement a single function, and virtual protocols. The resulting protocol graph performance is comparable to static protocols, but enables reuse of existing microprotocols, which leads to a more flexible network architecture.
Scout [MP96] is an operating system with a focus on data passing through multiple layers and the implementation of these “paths”. The session nodes known from the x-kernel configurations are no longer mandatory and cyclic configuration is partially supported. Execution in Scout is focused on “paths” and each path has implicit queues on its inputs and outputs and is run by a single thread and has a CPU priority. By classifying packets early on the correct path the scheduling of them can be optimized for their content or importance.

The companion paper [MPBO96] focuses on improving performance for the execution of these paths. By optimizing the usage of the memory system the protocol latency can be reduced. Mostly the improvements were in cache optimizations, especially for the I-cache. By modifying the compiler and instruct the network code with hints about the normal path, they were able to get a very dense code for the processing of a non erroneous packet. This is not directly applicable to the implementation in Barrelfish but it might be prudent to hold it in mind while designing the graph nodes for the network processing.

Click [MKJK99] is a C++ framework for building flexible and configurable routers. This is achieved by building a router configuration from fine grained elements. The graph in Click allow cyclic routes since they are needed for software routing. Since the main purpose of Click is routing many of its module are not of use in an endpoint system. Communication between modules can be either push or pull but is defined on a module basis. Queues and special Pull-to-Push elements are able to switch between the different connection types. Click normally runs inside a kernel thread and the packets processing is scheduled over a task queue. Only elements which can start processing a packet are placed on the task list. These Click routers are configured with Click-language configuration files which are passed to the kernel thread and support hot swapping of elements. Click shows that modularity does not introduce any overhead in routing.

3.2 Parallelization

SMP Click [CM01] is an extension to Click and allows Click configurations to parallelize packet processing tasks and adaptively or statically load balance those tasks across the available CPUs. Paths with multiple computational expensive elements can be pipelined across multiple CPUs by inserting queues in between. If the router does not have multiple expensive elements the packets can be load balanced by careful hash demultiplexing them onto different CPUs. To achieve good performance the management of sk_buff’s had to done by SMP Click itself so that the buffer are reused on the designated CPU.
RouteBricks [DEA+09] is a clustered software router system which uses a modified version of Click as a basis. Using parallelization across multiple servers and within the servers they show that multi queue NICs are essential for fast parallel processing of network packets in a multi-core environment. Normal application driven batching is extended by NIC batching to reduce PCIe and I/O buses transaction at the cost of a slightly higher latency. For RouteBricks NUMA-aware data placement is not performance wise relevant.

### 3.3 Buffer handling

PacketShader [HJPM10] extends the idea of a software router by offloading the routing functions to a GPU, on which routing tasks can be even more parallelized than on a normal multi-core CPU. Since the skb buffer system does not handle multi 10G NICs very efficiently they introduce a new buffer system called huge packet buffer. The driver allocates two huge buffers, one for metadata and one for data, with fixed sized cells. By reusing the cells any memory allocation for new buffers can be avoided. PacketShader processes packets in batches and passes them around in batches by copying them into a consecutive user-level buffer. They claim that this copying does not affect the performance very much and facilitates the abstraction. PacketShader uses also RSS and NUMA aware data placement to achieve in a similar machine as RouteBricks uses a four times higher throughput.

netmap [RL11] introduces a new buffer system similar to the huge packet buffer system in PacketShader. Packets that arrive on the NIC are stored in statically allocated buffers which can be mapped into the user programs address space. Along these buffers resides also a structure called netmap rings, which essentially replicate the NIC rings in a device independent way. Since all the buffers and netmap rings for all adapters are in the same memory region, shared by kernel and user applications, results in a zero copy forwarding between different interfaces by swapping the buffer indexes. netmap exposes the contents of all packets to all user applications, but user applications can’t crash the system because NIC rings are maintained by netmap in the kernel.
Chapter 4

Approach

This chapter explains the general architecture and design decision that led to the implementation of the new graph based network stack for the Barreльfish OS, called netg. The new network stack aims at providing scalable, flexible and robust networking capability for user-space applications.

4.1 Network Graph Overview

The network subsystem can be treated as a graph, where nodes process network packets and edges communicate between these nodes. This communication can be within a process or over inter-dispatcher communication.

As shown in figure 4.1 the graph starts at the network card and ends at the user-space application.

For security reasons if the network card cannot access to the memory by virtual addresses a user application should not be allowed to have write access to the receive and transmit rings of the network card, since an application can store any physical address in the receive ring and the network card would write an arriving packet into it. Therefore the path from the network card to the user-space application has to span at least 2 different protection domains.

When a network card has only one queue someone is needed to distribute the packets among the applications. This means the path from the network card to the user-space application has to cross at least to address spaces since one application has to act as a distributor. This application has access to all packets received on the network card.

Figure 4.2 shows a simplified overview over the different application of network card queues.
4. Approach

Figure 4.1: Simple overview of a network graph

4.1.1 Endpoints

Since network connections in Barrelfish are implicitly bound to a CPU core the end nodes of the network graph are always on the same core. Therefore without any explicit action the core locality of the data path endpoints cannot change. Because the destination core is known the data path can be optimized in a way that the packets are stored NUMA-aware in the memory or that the packet is already loaded into the cache.

4.1.2 Nodes

The complete network stack processing of packets is done in fine grained nodes. The nodes do some protocol specific work on the packet and then forward it to the next node or discard it (if for example the checksum failed). Classifier nodes are needed when a node has multiple next nodes and only
4.1. Network Graph Overview

Any node can have multiple following nodes from which it can chose to send the packet to after it has processed it. An example for a this kind of node is the protocol type classifier, which demultiplexes between UDP and TCP packets.

The nodes structure has to allow for dynamic graph changes without dropping any packets. Hence the transformation has to be either atomic or done when no packet is currently processed at a node. This can be done by the insertion of a queue node before the transformation begins. The queue node buffers the incoming packets and passes them along after the transformation is complete and then be removed. An atomic transformation for example is the removal of a single node (e.g. UDP checksum processing) because the only operation is to overwrite the information about the next node in the parent node with the to be removed node’s next node. The information about the structure of the network graph is stored in the nodes in form of a reference to the next node.

The communication interface between nodes needs to be generic and not protocol dependent in order to allow more flexibility. Unlike Click the communication between nodes is always in a push like fashion since we did not think a pull mechanism would bring any advantage in the case of a net-
work endpoint. For network routers this is a different case and there pull communication can be used to an advantage.

### 4.1.3 Queue Nodes

Queue nodes can be placed anywhere in the graph and can be used to pipeline the packet processing. Queue nodes should not be confused with hardware transmit or receive queue rings, since they are purely software queues and do not exist on the network card. If implemented in a thread-safe way, they can be used for thread synchronization with the limitation in the case of a lock-less queue that only a specific thread can put packets into the queue and only a specific thread can get packets from the queue. Queue nodes can only be used if both the producer and the consumer share the same virtual address space.

An example usage of queue nodes for an application with three threads and only a single connection to the previous node in the network graph (e.g. Queue Manager) is shown in figure 4.3.

Virtual queue nodes can be used for the various queues needed in the TCP processing. They are called virtual because they exist once per TCP connection and not only in the network graph. An example for one of this virtual queue nodes is the retransmission queue for a TCP connection since it has
to store packets based on their connection and not globally.

4.1.4 Communication between Nodes

For communication between different virtual address spaces and therefore different dispatchers IDC mechanism have to be used, especially the IDC based bulk-transfer protocol. These mechanism are already implemented and well tested in the Barrelfish OS and the cost of different IDC types is known.

For communication between nodes in the same address space but different cores and therefore different threads queue nodes or IDC mechanism can be used. IDC will have the lower latency per packet than synchronizations over queue nodes but adds overhead for the communication.

Communication between nodes in the same address space and the same core can be done by function calls except if multiple threads are to be used on the same core. In this case queue nodes have to be used.

4.1.5 Scheduling

Lock-free thread-safe queue nodes need to be scheduled on a fixed thread and in order to guarantee this, they have to be registered for scheduling. The scheduler stores a for each core a task list of queues which have packets to process.

This has the disadvantage that the scheduler has to be called on every core, where packets should be taken out of a queue, periodically to ensure liveliness.

4.1.6 Network Connection Information

A modular approach to packet processing requires a modular way to store connection information in order to take full advantage of the new flexibility. Because the nodes have a uniform interface to communicate with each other the connection information for TCP and UDP (unconnected and connected) connections needs a similar structure.

The information about the network cards needs a more flexible structure in order to handle multiple network card queue connections or direct hardware queue usage.

4.2 Network Graph Optimizations

Now that the network graph and the types of communication between the nodes is defined we can focus on trying to optimize the network graph to
4. **Approach**

achieve an optimal representation for the specific hardware used and the current usage of network resources. We differentiate between two different type of optimizations. One is the global optimization where the limited hardware resources like network card queues and processor cores play an important role. The other is more application local, in which applications can try to change the network graph for the packet processing to gain better performance.

4.2.1 **Global Optimizations**

The need to ensure optimal and fair distribution of the hardware resources led to the creation of a new service called Network Resource Manager (NRM), which acts as a broker. This NRM service distributes the available network resources among the applications with enabled networking based on which processor core they run on. As we have seen in section 2.1.5, communication is faster than 1mp. In order to reduce the latency cost the network card queue assigned to an application should if possible not be on the same processor core.

While some application may profit from a dedicated hardware queue or even user-space networking for a lot of small services this plays not really a major role, since they are only used sporadically. Take the network daemon for example, which is in the new network subsystem responsible for ARP lookups and DHCP and has no need at all for a dedicated queue. These services and applications with sporadic network usage should always use a shared queue.

To assign an application to a queue the NRM needs to keep track of the network usage of each process using network resources and processes need to be able to switch queues without interrupting their

4.2.2 **Local Optimizations**

Under local optimizations fall the packet processing network graph nodes which reside in the address space of a single process. These changes have no impact on the other networking applications and therefore do not need a global coordinator.

An example for a local optimization is the reuse of buffers for connected UDP connections which results in a smaller processing overhead for each packet but does not influence any other part of the network graph. How this works is explained in detail in section 5.5.1.

Another example would be the offloading of packet processing features to the network card. If the network card allows offloading settings per queue these feature do not influence other application with a dedicated queue or the shared queue.
This chapter explains the implementation of the network services and the new TCP/IP stack in details. It starts by explaining the changes to the services needed in order to add flexibility. In the second section focuses on the graph based TCP/IP stack.

5.1 Services

The most significant change in the architecture of the network subsystem (Figure 5.1) from the existing in figure 2.1 is the rearrangement of the network card driver and netd into the network device manager (NDM), Queue Manager and netd as well as the new network resource manager service. The splitting of the network card driver into NDM and Queue Manager enables the usage of multiple network queues on different cores. The figure shows only the case of a single queue network card where the Queue Manager and the network driver can be run in the same process. For cards with multiple queues for each queue a Queue Manager has to be spawned and the driver can run as a standalone process or be coupled with a queue.

As we have seen in section 2.1.5 the communication between dispatcher on different cores is much faster, which results in an optimal setup, where Queue Manager and networking application do not reside on the same core.

The Queue Manager is responsible for a single hardware transmit and receive buffer ring pair. The network device manager handles the NIC initialization and the port management and is responsible for the queue managers. The network resource manager service tries to find an optimal distribution of the available network resources to the requesting applications. While the existing network daemon (netd) is stripped of its port management responsibilities it is now handling the ARP lookups and has an ARP Table. The
5. Implementation

Figure 5.1: Overview of the new networking architecture

The implementation part is taken over by the Network Device Manager. The services are explained in detail in the following sections.

5.1.1 Network Resource Manager (NRM)

The Network Resource Manager acts as a broker for the available network resources. The NRM periodically requests network usage statistics from each application with assigned network resources. Based on this statistical knowledge the NRM can then distribute the available network queues to the applications with the highest need for a dedicated queue. Application with low network usage or low importance suffer a relative small performance penalty for using a shared queue.

Exported interfaces:
5.1. Services

- net_resource.if

5.1.2 Device Manager (NDM)

The Network Device Manager is split into a device independent and a hardware specific part. The hardware dependent part initializes the network card and handles card specific features. The device independent part is responsible for the management of the ports for the network card. For this it registers filters for ports with the relevant Queue Manager or for dedicated Queue Manager sends an IDC request to the driver to use the hardware filters. The interface for port registration was extended to allow a bound port to be moved from one queue to another in order to allow queue migration of an application without interrupting the network connections.

**Exported interfaces**:
- net_ports.if

**Consumed interfaces**:
- net_soft_filter.if

5.1.3 Queue Manager

There are two types of Queue Manager, one is the dedicated kind, where every incoming packet is forwarded to a single bulk-transfer endpoint. The other is the shared version of the Queue Manager where the bulk-transfer endpoint is determined using the previously registered filters. The software filters contain not only information about the network connection but also about the bulk transfer endpoint any packet should be sent to. For each incoming packet the filters are executed and if the filter matches the connection information in the packet it is copied into a buffer for the bulk transfer endpoint specified in the filter. Filters normally use only on the destination port and the connection type but can use the IP addresses and the source port too.

The part of the Queue Manager in communication with the network card is device dependent, the rest of it is device independent.

The special case of a loopback queue was added to facilitate development and to get performance figures without the hardware.

Every Queue Manager has a queue ID and it is used to identify it. The Queue Manager with queue ID 0 is always assumed to be a shared queue.

**Exported interfaces**:
- net_if_raw.if
- net_soft_filter.if
5. Implementation

5.1.4 Network daemon (netd)

The network daemon run once for every network card in the system and is doing the ARP lookup and stores a network card wise ARP table. Furthermore netd handles the IP address configuration. By default this is done dynamically over DHCP but netd can be configured to use a static IP configuration.

The only reason netd still uses lwIP instead of netg is because netg uses the DHCP implementation of lwIP. Once we have implemented DHCP in netg or over UDP in netg itself the dependency to lwIP can be resolved.

Exported interfaces:
- `net.ARP.if`

Consumed interfaces:
- `net.ports.if`
- `net.resource.if`
- `net_if.raw.if`

5.1.5 Dynamic starting

We did not implement dynamic spawning of the services but the idea was that the NRM starts the device manager and the netd based for all the network cards listed in the SKB with available drivers. The device manager then starts a single shared Queue Manager.

Dedicated Queue Manager would be spawned by the device manager dynamically based on the demand.

5.2 Network Graph Structure

The netg library contains the nodes for the TCP/IP processing of network packets and is able to configure itself based on configuration parameters and feedback from other networking service. After initialization the netg library is setup for sending and receiving packets for the specified network card. In the following sections the inner-workings of the netg library are further explained.

Consumed interfaces:
- `net.ports.if`
- `net.ARP.if`
5.2. Network Graph Structure

- `net_resource.if`
- `net_if_raw.if`

### 5.2.1 Initialization

The netg library is initialized by a function call to `netg_init`. This function has to be called for every network card the application wants to use because in it all the connections to the networking services are setup. The application has to pass along the network card it wants to use and what type of service it expects from the network stack.

In this initialization function netg will then setup connections to the required services and initialize a basic network graph in the following order:

1. connects to the NRM service
2. creates a basic network graph for UDP and TCP processing
3. asks NRM for access to the specified network card to which NRM will answer with a queue ID
4. connects to the port management service in the NDM of the specified network card
5. connects to the ARP service of the netd instance responsible for the network card
6. connects to the Queue Manager with the queue ID supplied by the NRM and initializes the buffer management (see section 5.2.3)
7. gets the IP settings from netd through the ARP service

This setup phase is shown in the sequence diagram in figure 5.2. If it is the second call to `netg_init` (for another network card) the steps are the same, except that netg already has a connection to NRM.

### 5.2.2 Network Endpoints

A not trivial problem was the unification of network connection information for TCP and UDP, especially since UDP is a connectionless protocol. This was needed in order for the network graph nodes to use the same interface for their communication with each other despite the fundamental differences in TCP and UDP.

A connection is now represented by 2 instances of the `netg_endpoint` (Listing 5.1) structure, which store information about the connections endpoints. Both of these `netg_endpoints` refer depending on the connection type to a transport layer state structure. This structure stores the information about
5. Implementation

Figure 5.2: Sequence diagram showing the initialization of the netg library

the connection specific states. Unconnected UDP endpoints are not linked together in their UDP state structure. The remote endpoints stores in the mac_addr field the next hop mac address.

```c
struct netg_endpoint {
    // UDP or TCP
    enum netg_endpoint_type type;
    uint16_t port;
    ip_addr_t ip_addr;
    mac_addr_t mac_addr;
    net_interface_t net_interface;
    // closed, connected, listen
    enum netg_endpoint_status status;
    // pointer to the UDP or TCP transport layer informations for this connection
    void* transport;
}
```
5.2. Network Graph Structure

```c
struct netg_endpoint* next;
```

**Listing 5.1:** Endpoint structure

### 5.2.3 Buffer Management

The buffer management plays an important role in any networking domain. For that reason we tried to take advantage of the results from the huge packet buffer system that was introduced in PacketShader [HJPM10].

Upon initialization two chunks of memory are allocated, one where the buffer data is stored in and the other is for the buffer meta-data. These buffers are split in half, where as the first half is used for the receiving of network packets and the second half is for the sending of packets.

The buffer chunk is split into fixed sized 2048 byte cells and the meta-data is populated with these addresses. If the buffers are used to communicate with a Queue Manager the memory containing the buffers is shared with the Queue Manager. Afterwards the receive buffers are registered with the Queue Manager for receiving and the transmit buffer are stored in a free list in the Queue Manager connection structure.

When the application requests a new transmit buffer the first buffer on the free list is returned and its reference count is set to 1. Before the buffer is returned netg looks at the protocol the endpoint is using and adjusts the payload pointer in order to safe enough space at the head of the buffer for all the protocol header needed.

Afterwards the application copies some data into the buffer and sends it to the Queue Manager. Upon completion of the processing of the buffer the Queue Manager informs the netg library that the buffer can be freed. When freeing a buffer the reference counter is reduced and if it is zero the buffer is returned to the front of the free list.

When protocol optional options are used the buffer needs to be copied into a new buffer because the reserved space at the buffer head is too small. This is not a problem since protocol options are rarely used in the critical path. For the TCP connection setup this does not apply since there the packet do not have a payload and the TCP options can be appended without copying.

Received buffers are set to have reference count 1 and have to be freed by the user application when they are done reading from the buffer.

In most cases the reference counting is only used for TCP retransmission, where the buffer is freed when the corresponding acknowledgment is received.
5. Implementation

Figure 5.3: Buffer allocation

```
struct netg_buffer {
    // connection information
    struct netg_endpoint* local_endpoint;
    struct netg_endpoint* remote_endpoint;
    // network card information
    uint64_t queue_id;
    uint64_t net_interface;
    // start address of the buffer
    void* data;
    // dynamic start address for user payload
    void* payload;
    // length of the user payload
    uint16_t length;
    // length of the header
    uint16_t header_length;
    uint8_t transport_flags;
    uint8_t ref_count;
    // used for time measurements in the network graph
    uint64_t timestamp;

    struct netg_buffer* next;
};
```

Listing 5.2: Buffer structure

5.2.4 Network Graph Nodes

Network graph nodes (Listing 5.3) consist of packet processing function and an optional node state structure as well as a next pointer for simple nodes. The processing function takes as argument the currently being processed
buffer and a pointer to the node where it is being processed at. The last part
is needed for the processing function to know where its position is in the
network graph. Listing 5.4 shows the process function of the dummy node,
which was used to measure the latency introduced per module.

```c
struct netg_node {
    // function pointer to the processing function
    errval_t (*process)(struct netg_buffer*, struct netg_node*);
    // pointer to the node state
    void* node_state;
    // next network graph node
    struct netg_node* next;
};
```

Listing 5.3: Network graph node structure

The buffer structure contains all the information needed for the complete
processing of a network packet. When the packet is received from the Queue
Manager the endpoints are not yet known and therefore NULL. This is only
allowed for receive buffers up until the endpoint classifier of the protocol the
network packet is using. Sending buffers cannot have NULL as an endpoint.

```c
errval_t process_dummy(struct netg_buffer* buffer, struct netg_node* this) {
    if (this->next != NULL) {
        return this->next->process(buffer, this->next);
    }
    return SYS_ERR_OK;
}
```

Listing 5.4: A process function of the dummy node

If the nodes path at the current node does not split the next pointer points
to the next node and is invoked with the buffer and the pointer to its node
structure. When the node is connected to multiple nodes a structure is
needed to store the node structure pointers. An example for this is the
structure in Listing 5.5.

```c
struct netg_node_state_tcp_rx_classifier {
    struct netg_node* listen;
    struct netg_node* normal;
};
```

Listing 5.5: Network graph node state structure

The lock-less thread-safe queue node uses the node state pointer with a
queue node state structure to store the buffers in a ring queue. The next
pointer of the buffer is not used for the queue nodes since a buffer can be in multiple queues at any time, for example a TCP segment can be in the retransmit queue, where it waits for acknowledgment or a timeout, as well as in a normal queue node. The next pointer is only used for the free list and maybe in the future for batching the packet processing.

For each protocol netg supports a start node is specified, which can be exchanged while running. This start node is called by the sending function of the corresponding protocol and that’s how the journey through the graph starts for the sending of packets. A special node handles the transfer of the buffer to the Queue Manager and frees the buffer upon receiving the sent call from the Queue Manager.

On the receiving side things are handled similarly. When a call comes from the Queue Manager that there is a new packet the buffer of the packet is sent to a globally referenced node which starts the processing of the network packet. At the end of the receiving side are nodes which call the packet arriver callback which the user application registers for each endpoint that can receive packets.

When processing a received packet the buffers payload pointer is first set to the beginning of the data chunk. When the processing of a protocol part is finished the buffers payload pointer is set to the beginning of the next protocols header or the beginning of the user payload if the last protocol is processed.

5.3 Application Interface

We chose to leave the application interface similar to the interface of lwIP [Dun03], as this is a well proven interface. Furthermore this makes it easier to port an application from using lwIP to netg. The only real difference in the interface for TCP or UDP connections is that netg does not have a protocol control block (pcb) as lwIP has, for that reason netg replaces the occurrence of pcb with netg_endpoints and the pbuf reference of lwIP are replaced by netg_buffer references. A cosmetic change is that in netg we tried to use the normally in Barreelfish used types for integers and error values. Listing 5.6 shows the difference between the tcp_write call in lwIP and netg.

```c
// lwIP
err_t tcp_write(struct tcp_pcb *pcb,
    const void *dataptr,
    u16_t len,
    u8_t apiflags);

// netg
errval_t tcp_write(struct netg_endpoint* endpoint,
    const void *dataptr,
```
Changes were needed for the library initialization which was described in section 5.2.1. Furthermore we introduced a new function called *netg_tick* which schedules queue nodes containing packets to process. The buffer handling functions for getting, freeing and claiming (increase reference count) are different as well, since they handle *netg_buffer* instead of *pbuf*.

The only two structures which are present in the application interface are the *netg_buffer* and the *netg_endpoint* structure. The remaining netg structures are abstracted from the user application by the interface.

### 5.4 TCP/IP processing in the Network Graph

Now that we have defined how the components of the network graph look like it is time to define what the network graph looks like when the things are put together. The currently implemented network protocols are TCP and UDP but the addition of another protocol is quite simple. During the implementation of the protocol functionality the modular design was very helpful and greatly reduced debugging time since the misbehaving node was found fast and the problem could be localized without a lot of searching.

The implementation of the protocols was based done with the help of the books TCP/IP illustrated (Volume 1 [Ste93] and Volume 2 [?]).

#### 5.4.1 Ethernet & IP

In the IP layer of the network graph the IP header is added to the buffer and another node calculates the IP checksum. These nodes were split to enable future versions of the network graph to disable IP checksum calculation by just removing the IP checksum node.

The Ethernet part of the network graph is relatively straight forward. The *arp_lookup* node checks if the MAC address of the destination endpoint for the current buffer is known. If not the node makes an rpc call to the *netd* to get the MAC address. In this module the routing decision is made based on the if the destination IP address is in the local subnet, if not the packet is sent to the gateway IP address. Once the MAC address is known the buffer is passed along and the MAC address is set for all the connections to this endpoint.

Another node adds the Ethernet header and passes the buffer to the node which is responsible for sending it to the Queue Manager. The Ethernet CRC is done in hardware and does not need to be calculated in software.
5. Implementation

On the receive side the Ethernet processing node passes the buffer along if the packet is an IP packet. Then the IP checksum is checked again in a separate node for the same reasons. The IP processing node sets the buffers length from the IP header and passes the buffer along to the protocol classifier which is used to split the UDP and TCP packets.

5.4.2 UDP

Similar to the way IP is implemented as nodes the UDP nodes for sending a packet consist of two nodes, one for the header and one for the checksum calculation. This way the checksum calculation can be disabled by removing the node from the graph if it is done in hardware or one does not want the checksum calculated since it is an optional UDP feature.

```c
struct netg_udp_state {
    struct netg_endpoint* local;
    struct netg_endpoint* remote;

    // callback
    void (*recv) (void *,
                  struct netg_endpoint*,
                  struct netg_buffer*,
                  ip_addr_t,
                  uint16_t);
    void* arg;

    // used for buffer reusing
    struct netg_buffer* buffer_free_list;
};
```

Listing 5.7: UDP protocol state structure

Any received UDP packet is first checked if the checksum is correct or disabled. The next node searches the corresponding endpoints of the packet and adds them to the buffer. This search is done in the following way:

1. search remote endpoint
2. no remote endpoint found?
   create new endpoint
3. is connected?
   get local endpoint from `netg_udp_state` structure and go to step 6
4. search local endpoint
5. no local endpoint found?
5.4. TCP/IP processing in the Network Graph

discard packet, endpoint closed or erroneous packet

6. add endpoints to buffer

If the endpoint lookup was successful the buffer payload and length is updated and the last node calls the `recv` callback from the `net_udp_state` structure.

### 5.4.3 TCP

The TCP implementation is based on the TCP New Reno protocol and has more than double the node count than the UDP implementation. Since we are trying to stick as close to the lwIP interface as possible we do have to copy the data into the buffers in the netg library instead of as in the UDP in the user application.

**Sending**

The `tcp_write` function splits the data into multiple segments if the data is bigger than the maximum segment size allows. In the next node the TCP header is added and passed to the node which checks if the TCP segment is allowed to be sent by the send window or if it has to be enqueued. The queue is stored in the `netg_tcp_state` (Listing 5.8) instead of the queue node state since the queue is connection dependent. If enqueued the queue is set as runnable and will be scheduled soon after till it is empty. After being able to send the queue is stored in the retransmit queue and passed to the node which update the `ACK` field in the header. The reason this is split from the rest of the header is that always want the most recent `ACK` in the header. Afterwards the TCP checksum is calculated and the buffer is given to the IP node described in section 5.4.1.

```c
struct netg_tcp_state {
    struct netg_endpoint* remote;
    struct netg_endpoint* local;

    // TCP state e.g. SYN_SENT, ESTABLISHED, FIN_WAIT1...
    uint8_t state;

    uint16_t maxseg;

    // send window
    tcp_seq snd_unack;
    tcp_seq snd_next;
    tcp_seq iss;
    uint32_t snd_window;
}
```
/ congestion avoidance
uint32_t cwnd;
uint32_t ssthresh;

recv window
tcp_seq rcv_next;
tcp_seq rcv_acked;
tcp_seq irs;
uint32_t rcv_window;

retransmission
int16_t rtx_timer; // retransmission timer
int16_t rtx_timeout; // retransmission timeout
uint8_t rtx_count;

fast retransmission
tcp_seq last_ack;
uint8_t dup_acks;
bool is_fast_recovery;

time_wait timer
int16_t two_msl_timer;

delayed ack
uint8_t full_sized_segment_count;

connection specific queues and lists
struct netg_buffer * reorder_list;
struct netg_node_state_queue* retransmit_queue;
struct netg_node_state_queue* unsent_queue;

callbacks
void* arg;
errval_t (*accept) (void *arg, struct
netg_endpoint* endpoint, errval_t err);
errval_t (*connected) (void *arg, struct
netg_endpoint* endpoint, errval_t err);
errval_t (*recv) (void *arg, struct netg_endpoint*
endpoint, struct netg_buffer * p, errval_t
err);
errval_t (*sent) (void *arg, struct netg_endpoint*
endpoint, uint16_t len);
void (*err) (void *arg, errval_t err);
5.4. TCP/IP processing in the Network Graph

Listing 5.8: TCP protocol state structure

Receiving

The first node as always checks the TCP checksum for correctness and the next looks for the endpoints structures in a similar way as UDP does it, except there are no unconnected TCP endpoints. If no connected endpoints are found the node looks for a listen endpoint and if there is no listen endpoint the packet is discarded. The buffer is then forwarded either to the listen node in the case of a listen endpoint or to the node containing the TCP state machine. The listen node will be explained later on.

After the TCP state is updated it is forwarded to the next node which is responsible for updating the connection information. This node removes acknowledged TCP segments from the retransmit queue and sends an ACK-packet if the sequence number is out of order as well as starts the fast retransmission process if three duplicated acknowledgments arrive. If a TCP packet does not contain any data (ACK-packet) the packet is discarded in this node.

The next node reorders packets that are out of order. The next node depends on the acknowledgment strategy being used. Either the node sends an immediate ACK-packet or it only does it if three full sized segment arrive after each other and the acknowledgment either happens by an outgoing packet or after the acknowledge timer fires.

TCP timers and Resend

When the first TCP endpoint is created by tcp_new the TCP timer is initialized and fires every 250 ms. If the delayed acknowledgment strategy is used it checks each connection for unacknowledged packets and sends out the corresponding ACK-packets.

Every second time the TCP slow timer is executed which increased the retransmit timer for unacknowledged sent packets. If a timeout occurs the buffers from the retransmit queue of that connection are transfered to the send window check queue and that queue is scheduled.

The timer removes closed endpoints in the TIME_WAIT state after their 2MSL timer.

Connection Establishment

The connection three way handshake jumps a few nodes and directly starts by handing a hand crafted TCP SYN-packet to the retransmission node.
5. Implementation

When a SYN-packet arrives at the TCP endpoint classifier and a corresponding listen endpoint is found the listen node creates a new TCP endpoint for this connection and sets the netg/tcp/state up. Then it creates a SYNACK-packet and hands it to the retransmission node.

The received SYNACK-packet causes the TCP state machine node to call the tcp_connect callback and sends an ACK-packet back. Which when received brings the state machine node to call the tcp_accept callback with the endpoint created in the listen node.

TCP Options

TCP comes with many options and currently netg only supports the maximum segment size option. Other TCP options could be added but we did not think they are important at the moment.

The only exception would be selective acknowledgments and probably window scaling which.

5.4.4 Resulting Graph

This put all together looks like the graph in Figure 5.4 and is the base network graph for any application.
5.4. TCP/IP processing in the Network Graph

Figure 5.4: Initial configuration of the network graph
5. Network graph optimizations

In this section we provide some information about the implemented optimizations to the network graph and how they work.

5.5.1 Transmit buffer reuse

As the figure 5.5 shows the changes in a buffers header for the same UDP endpoints are relative small as the only thing that changes are the length fields in the IP and UDP header as well as both checksums. An idea was to preallocate buffers for connected UDP endpoints to decrease the time needed to travel the network graph. But we didn’t want to preallocate a buffer count for all the connected UDP endpoints since this would waste a lot of memory.

Instead of preallocating and filling the header of all the buffers assigned to this connection we simply reuse buffers once they are sent. This is done by storing a free list per connected UDP endpoints pair in the netg udp state. When a buffer is requested for this connection the free list is checked and if empty a empty buffer is returned. Buffers from the free list are marked with a reuse flag and then use a fast path through the network graph. In the beginning of UDP processing a classifier is inserted which switches between buffers with the reuse flag and those without. In the fast path a new node is inserted which sets the UDP length as well as the IP length fields. Afterwards only both checksums are calculated and the Ethernet layer is skipped by directly sending it to the Queue Manager. Figure 5.6 shows the changes to the network graph introduced with this.

This has the advantage that a lot of nodes and header filling steps can be
5.5. Network graph optimizations

Figure 5.6: The new fast path for connected UDP endpoints
skipped at the cost of sending a packet to the wrong MAC address if it would change. Since MAC address changes do not happen often we think this is acceptable.

While this does not change the network graph at runtime it is still a dynamic optimization since the path of the buffer is determined at runtime.

5.5.2 Dynamic queue assignment

The Network Resource Manager asks each application which requested network access every 5 seconds for statistics. These statistics include the sent and received packet count since the last statistics request was issued and the status of the queue connection (no connection, single, multiple). These packet counts then are averaged using the following formula:

\[ \text{pkt}_{\text{avg}} = (0.9 \times \text{pkt}_{\text{avg}}) + 0.1 \times \text{pkt}_{\text{count}} \]

Where \( \text{pkt}_{\text{avg}} \) is the calculated average and \( \text{pkt}_{\text{count}} \) is the packet count from the statistic response.

If an application is above a threshold and has a single connection to a shared queue but a dedicated queue is available the NRM searches for an unused dedicated queue. If the NRM finds one it sends the new resource information to the application which switches the queues.

When the applications packet count drops below the half of the before mentioned threshold the NRM tells the application to switch back to the shared queue and frees the dedicated queue that it can be used again.

5.5.3 Load balancing of TCP connections

Another optimization to the network graph we implemented is load balancing of TCP connections. An application can load balance a listen endpoint by invoking the \texttt{tcp\_listen\_loadbalanced} on a unbound TCP endpoint. This results in a call to the new netg\_loadbalancer application which then initiates a connection for the network card the application wants to do the load balancing on if it doesn’t already has one. When the connection to the network card is working the netg\_loadbalancer binds to the port and saves the application as a recipient for \textit{SYN}-packets for this port. For further applications which want to participate in the load balancing of the same port this can be skipped and they are only added to the list of recipients. Arriving \textit{SYN}-packets for a listen endpoints are then shared equally among the registered recipients for this listen endpoint.

Since \textit{SYN}-packets are small and in most of the cases fit into a single cache line they are sent over the normal IDC path from the load balancer to the application.
5.5. Network graph optimizations

Once the packets arrive at the application they are copied into a buffer and sent to the network graph for the protocol processing. Before that the application registers the specific connection with both IP addresses and ports with the port manager that the following packets are directly received.

Currently the netg_loadbalancer can only be initiated once, but it is not a technical restriction and load balancers could be initiated for each port to share or each network card separately.
In this section we present the evaluation of the design presented in the approach chapter by running several benchmarks on the the implemented library netg. The objective of these evaluations is to prove that modularity of our design does not adversely affect the performance and give a few examples of the flexibility of this design.

6.1 Network Graph Evaluation

In this first part we are investigating the performance of the packet processing graph with the help of the loopback Queue Manager.

For that reason the network graph is rather simple since the focus is on how long it takes to process a single packet.

6.1.1 Latency per Node

Since modularity comes with a price we wanted to investigate how much overhead the modularization adds and what the time cost were for a single node.

To test this we chained 1000 dummy nodes together and put a timestamp node at the front and an evaluate timestamp node at the end (Figure 6.1). Through this chain 500 buffers were sent and the evaluate timestamp recorded the total cycle count used to traverse the chain. As Table 6.1 shows the cost of a single dummy node on the nos6 machine is very low as was expected.

![Figure 6.1: Node latency benchmark setup](image-url)
6. Evaluation

<table>
<thead>
<tr>
<th>Complete chain</th>
<th>1 dummy node</th>
<th>average CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6404</td>
</tr>
</tbody>
</table>

Table 6.1: Cost of modularization

![Diagram showing packet processing benchmark setup]

Figure 6.2: Packet processing benchmark setup

6.1.2 Total Latency for Packet Processing

In order to find out how long netg takes to process a single packet we measured the time between the send call and the receive callback. Furthermore we instrumented the netg network graph to record the time when the packet is handed off to the Queue Manager and the time the buffer is received from the Queue Manager (Figure 6.2).

For a warm up phase which was not counted 1000 packets were sent and then for the test 100’000 packets were sent through the Queue Manager and the time needed for each step was summed up. The tests were all run on the nos6 machine.
6.1. Network Graph Evaluation

<table>
<thead>
<tr>
<th></th>
<th>UDP</th>
<th>connected UDP</th>
<th>lwIP UDP</th>
<th>TCP</th>
<th>lwIP TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>4243</td>
<td>4197</td>
<td></td>
<td>5154</td>
<td></td>
</tr>
<tr>
<td>Transmit</td>
<td>1130</td>
<td>1009</td>
<td></td>
<td>1610</td>
<td></td>
</tr>
<tr>
<td>Queue Manager</td>
<td>2132</td>
<td>2234</td>
<td></td>
<td>2120</td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>981</td>
<td>954</td>
<td></td>
<td>1424</td>
<td></td>
</tr>
<tr>
<td>Protocol cost</td>
<td>2111</td>
<td>1963</td>
<td></td>
<td>3034</td>
<td></td>
</tr>
<tr>
<td>µs</td>
<td>0.75</td>
<td>0.70</td>
<td>0.8</td>
<td>1.08</td>
<td>1.92</td>
</tr>
</tbody>
</table>

Table 6.2: Cost of packet processing

![Figure 6.3: Packet processing cost comparison](image)

**UDP**

The test was run with unconnected UDP as well as with connected UDP and buffer reusing and the results are in Table 6.2. The connected UDP was 148 cycles faster. This shows that the buffer reuse is a simple way to optimize the network graph and achieve better performance at no cost. The test was run multiple times and the values for the transmit and the receive part are very stable but the part of the Queue Manager may vary quite a bit. This is because the Queue Manager is on another core and the IDC messages cost vary more than simple function calls.

Our implementation of the UDP protocol is quite fast since and the overhead of the modularization is not that high because UDP is a very simple protocol and does not need many nodes in order to be processed. The buffer reuse does not improve the latency significant but shows that the protocol graph is flexible and can be arranged differently without any problems.
6. Evaluation

TCP

The same test was run for TCP and it is as expected that the TCP processing takes more time since TCP is a much more complex protocol. The results from table 6.2 are very interesting because unlike lwIP where the TCP protocol processing time is more than double than for the UDP protocol in netg the time increase is less significant. That netg is this much faster is probably due to the optimized buffer management system and that netg has a very basic TCP protocol support whereas lwIP supports many optional TCP features.

These results show again that the packet processing performance is not hindered by the modularity. Furthermore the results were very stable and only changing between different machines and CPUs. We ran each test at least 5 times and the difference in the cycle count for the processing cost was at most 40 cycles.

6.2 Queue switching

In this section we are showing the difference in performance of different queue configurations. For this evaluation we used two instances of a very simple HTTP server. The server just replies to any incoming requests with a static web page reply sized at 1135 bytes (175 bytes header 960 bytes content). These server run on different cores and use different ports. The one running on port 80 is used to measure the latency and the one on port 8080 is used to generate load. For both the measurement and the load generation httperf is used.

6.2.1 Difference between Queue Configuration

To have a baseline the measurement we did a run without the load generator on a shared queue which was run on a different core than the HTTP server. The results are acceptable considering that this simple HTTP request each consists in the end out of 9 TCP packets (3 connection handshake, 2 payload, 4 close). But in this chapter we are not focused on latency of a HTTP request, but the difference in the latency between different queue setup strategies.

The next run was the same expect that we did use the other HTTP server to generate some load on the shared queue. This leads to that both application use the same Queue Manager which should increase the latency of the requests. As the table 6.3 shows this is the case, but the increase in latency is not very dramatic.

Finally we moved both HTTP servers to a dedicated queue and therefore eliminated the Queue Manager as a possible bottleneck. In this case each
6.2. Queue switching

<table>
<thead>
<tr>
<th>Configuration</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared, no load</td>
<td>0.2</td>
<td>0.3</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>shared, with load</td>
<td>0.2</td>
<td>0.6</td>
<td>14.3</td>
<td>2.0</td>
</tr>
<tr>
<td>dedicated, with load</td>
<td>0.2</td>
<td>0.4</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td>NRM, with load</td>
<td>0.2</td>
<td>0.4</td>
<td>0.8</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 6.3: Performance overview for different queue configurations. All the units are in microseconds.

HTTP server connects to its own Queue Manager and the interference between them should be reduced. The filtering of the packet does not have to be done in software on the Queue Manager anymore but is instead done in Hardware. The improvement over the shared queue is not significant but the latency is slightly reduced but we expected more improvement out of this setup.

6.2.2 NRM distributed Queue Configuration

We have shown that the queue ring distribution is important and that netg with the help of the NRM can be used to take over the configuration of the distribution on the hardware network card queues from static configuration.

For our test case with the two HTTP servers NRM decided that the load generator server deserves a dedicated queue manager. The server used for the latency measurements was not assigned a dedicated queue since the load is below the threshold NRM uses to be assigned one. Unsurprisingly
6. Evaluation

<table>
<thead>
<tr>
<th>Configuration</th>
<th>normalized request per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 server (no load balancing)</td>
<td>1.0</td>
</tr>
<tr>
<td>2 load balanced servers</td>
<td>1.98</td>
</tr>
<tr>
<td>3 load balanced servers</td>
<td>2.95</td>
</tr>
<tr>
<td>4 load balanced servers</td>
<td>3.91</td>
</tr>
</tbody>
</table>

Table 6.4: Normalized benchmark results for different load balancing configurations

the results are very similar to those from the test with the dedicated queues. This shows that the NRM chose an optimal solution for this application and load setup and did not waste a dedicated queue which would not result in any performance gain.

6.3 Load Balancing of TCP Connections

Unfortunately the driver for the e10k network card supports at the moment only the 128 5-tuple hardware filter of the card and for that reason only 127 TCP connections can be load balanced. Because every connection needs a filter and the listen port needs one as well.

Therefore we had to test the load balancing on a single shared queue, which will be the bottleneck for the HTTP server introduced in the section 6.2. To transfer the bottleneck from the queue to the server we introduced a thinking time in the HTTP server, which simulates the calculation of the result for the given request. The thinking time is constant and for all requests the same.

To test the load balancing we then ran httperf with only one modified HTTP server without load balancing and with multiple server with load balancing. Table 6.4 shows the normalized request per second the servers achieved and were averaged over five runs each. We omitted the error bars in Figure 6.5 because the variation between the different runs were too small.

The increase in performance is nearly linear. The reason that it does not completely scale linear is probably the added overhead for the software filter installation for every new connection over the same network device manager and the shared usage of the same queue manager. We believe that we can improve the results with dedicated queues and the hardware filters.

We are currently working on the e10k driver to enable the usage of the hash-filters. Once the driver supports these we can load balance multiple thousand connection on dedicated queues.

6.4 Conclusion

In this chapter we have not only proven that the added flexibility through modularity does not adversely affect the performance of the network stack
but also enables optimizations like load balancing or dynamic queue assignment.
In this thesis we have introduced netg, a new modular network stack for the Barrelfish OS. Netg supports the complete TCP protocol in the New Reno specification as well as full UDP support. The buffer system of netg is inspired by recent works on improving software routing.

Our evaluations show that the usage of modularity to gain more flexibility does not only work but the performance is also comparable to the inflexible existing network stack. The computational cost the modular nodes introduce is very small and if one does not build huge chains can be neglected. As the time needed for the processing of a TCP packet shows, in which netg outperforms the existing lwIP network stack in Barrelfish. For the simpler UDP protocol the performance is comparable to that of lwIP. While the reusing of UDP buffers did not show a significant improvement in the processing performance it did show that the protocol processing graph can be adapted to new ideas very quickly and efficiently.

The modular approach facilitates the implementation of protocols in general and really helped with the implementation of the complete TCP/IP stack. Even for debugging the modular network graph nodes are of help, since it is very easy to find out in which node the error occurs. Because the nodes should only implement one feature of the protocol the error can be dealt with efficiently. New protocols can easily be implemented in netg and then added to the network graph.

We think that using modularity in the network stack is the right way to deal with the new features of the network cards offer and that netg will show its real strength when the network drivers for Barrelfish allow the use of the advanced features of the network cards like complete user-space networking.
7. Conclusion

7.1 Future Work

The current implementation of our approach in the netg library is already usable and can be used as it is now. But the introduced flexibility offers a lot of room for experiments and can definitely be used as starting point for further work.

While we have support for offloading protocol features in software the hardware drivers still do not support this very useful feature. We think it would really improve the scalability of the overall system.

Currently the implementation of the queue node scheduling does not support multiple threads. We think the usage of queue nodes to span the network graph of a single application onto multiple threads on possibly different dispatchers in the same address space would be worth looking into as well.

The bulk transport mechanism doesn’t support read ahead at the moment. This could be used to process the packets in batches and recent works in software routing have shown that this improves the performance.

With the right network card and the support of the driver netg would allow complete user-space networking, which we think would reduce the latency significantly. Netg would need to be extended to support the manipulation of the network cards queue ring buffers instead of using bulk transfer to communicate with a Queue Manager. The change would be confined to the last graph node on the sending path and the first on the receiving path.
The hardware specification of all the machines used in the evaluation benchmarks. The machines \textit{sbrinz1} and \textit{emmentaler} are connected over a 10G Ethernet switch. \textit{nos6} was only used for loopback experiments without any real networking.

\textbf{nos6}:

- CPU: AMD Santa Rosa (Opteron 2200) 2.8 GHz
- Cores: 2x2
- RAM: 8 GB

\textbf{sbrinz1}:

- CPU: AMD Shanghai 2.5 GHz
- Cores: 4x4
- RAM: 16 GB
- NIC: Intel 82599 (e10000)

\textbf{emmentaler}:

- CPU: Intel Xeon E5345 2.33 GHz
- Cores: 2x4
- RAM: 16 GB
- NIC: Intel(R) PRO/1000 (80003ES2LAN)
Appendix B

Network graph module list

Here we define all the current modules and what their function is.

**UDP**

Here are the UDP related nodes including the buffer reuse optimization.

**tx_is_reuse** :

Checks if the current buffer has the reuse flag set. If it has it forwards the buffer to the reuse path otherwise to the normal path, which is normally `tx_udp`

**tx_udp** :

Here the UDP header is written into the buffer

**tx_udp_reuse** :

The UDP length and the IP total length fields of the header in the buffer are updated.

**tx_udp_checksum** :

The UDP checksum is calculated and inserted into the UDP header in the buffer.

**rx_udp_checksum** :

Counterpart to the node before, here the Checksum is calculated and if it does not match the buffer gets discarded.

**rx_udp_endpoint_classifier** :

Looks up the corresponding netg endpoints for this connection or creates the remote endpoint if none already exist.
B. Network graph module list

rx_udp :

    Updates the header length and the payload length in the netg_buffer and sets the payload pointer of it to the start of the user payload in the buffer.

udp_recv :

    Calls the user specified callback function for the receiving of a UDP packet. The callback was set on a per endpoint level.

TCP

TODO

IP

The IP related packet processing nodes are listed here.

tx_ip :

    Writes the IP header for the current connection into the buffer. The checksum field is set to 0.

tx_ip_checksum :

    The IP checksum is calculated and added to the header in the buffer.

tx_ip_checksum :

    The IP checksum is calculated and if it matches the packet is forwarded, otherwise it is dropped.

rx_ip :

    Updates the header length and the payload length of the netg_buffer structure and sets the payload pointer of it to the start of the next header.

Ethernet

The three Ethernet packet processing node consist of these.

tx_ethernet :

    Adds the Ethernet header to the buffer. The checksum calculation is done in hardware and is omitted.

ARP_lookup :

    Checks if the MAC address of the next hop for this packet is known, otherwise looks it up by requesting it from netd. If successful or known the buffers are forwarded to the next node.
**rx_ethernet** :
  Only forwards if the type IP is specified in the type field of the Ethernet header. The node updates the header length and the payload length of the buffer and sets the payload pointer to the start of the IP header.

**Common**

**classifier** :
  Classifies packets and sends them to the node for that protocol. Currently only TCP and UDP are supported.

**discard_buffer** :
  Frees the supplied buffer. This is globally known module and all modules can reference it.

**runnable_queue** :
  This is the queue node described in ??.

**timestamp** :
  Sets the time stamp field in the netg_buffer structure.

**evaluate_timestamp** :
  This prints out the time since the time stamping of the buffer in cycles. When creating the evaluate time stamp a string can be supplied which is printed out as well.

**sendto_queue** :
  Sends a buffer to the corresponding queue and updates sent statistics.

**recvfrom_queue** :
  Resets buffer values and sets the buffer length fields and then forwards it to the next node.

**Debug**

Here the useful nodes are described for debugging the network graph.

**dummy** :
  Does nothing except forwards the packet to the next node. Used to measure the overhead of modularization.

**debug_buffer** :
  Prints some debug information of the buffer.
B. Network graph module list

**dump_buffer** :
Prints a hex-dump of the complete buffer.

**loopback** :
This resets the buffer statistics and sends the buffer to the next node, can be used to directly connect the tx and the rx branch of the network graph.

**packet_loss** :
This node is initialized with a parameter which specifies every which packet has to be dropped. These packets are then sent to the *discard_buffer* node instead of the next.
Bibliography


