ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Ethernet Message Passing for Barrelfish

Distributed Systems Lab

Jonas Hauenstein, David Gerhard, Gerd Zellweger

July 8, 2011

Advisors: Prof. Timothy Roscoe, Akhilesh Singhania

Department of Computer Science, ETH Zürich

Abstract

We introduce Ethernet Message Passing (EMP) for the Barrelfish Operating System. EMP is an interconnect driver (ICD) which is able to send and receive messages between dispatchers among multiple machines over an attached network.

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 Barrelfish

Barrelfish is a research operating system being developed collaboratively by researchers in the Systems Group at ETH Zurich in Switzerland and at Microsoft Research. It is intended as a vehicle for exploring ideas about the structure of operating systems for hardware of the future. Barrelfish anticipates the main challenges for operating systems will be scalability as the number of cores increases, and in dealing with processor and system heterogeneity [1].

1.1.2 IDC: Inter Dispatcher Communication

In Barrelfish, a dispatcher is a unit of kernel scheduling and responsible for the management of its threads. The scheduling of these dispatchers is controlled by upcalls from the kernel. Communication between different dispatchers (so called Inter Dispatcher Communication or IDC) is performed over different channels. The two central mechanisms in Barrelfish for X86 hardware are LMP (local message passing) and UMP (inter-core user-level message passing). While LMP is used for IDC between two dispatchers on the same core, IDC between different cores uses UMP. Which mechanism is used is determined at binding time.

1.1.3 LWIP

The LWIP library is used as the Operating System´s network stack [2]. LWIP is a independent and lightweight implementation of common network protocols including UDP and TCP/IP. It was originally written by Adam Dunkels of the Swedish Institute of Computer Science for his master's thesis [3]. Our EMP Channel uses LWIP to send and receive messages over UDP.

1.2 Problem and Motivation

Barrelfish is an OS which treats the inside of a multicore machine as a distributed, networked system. Therefore, the step towards an Interconnect Driver (ICD) which supports communication not only among multiple cores within a single machine (as UMP does) but also among cores on multiple machines is most reasonably and at hand. The main goal of this distributed systems lab project is to provide a basic solution and a prove of concept for this need.

1.3 Overview

Chapter 2 provides an overview to basic structure of the EMP message channel and contains a discussion about the most crucial design decision we took. Chapter 3 describes our implementation in more detail. While Section 3.1 covers the EMP channel implementation, Section 3.3 describes the modifications on the monitor to offer inter-machine communication and the corresponding connection setup. Chapter 4 evaluates our implementation using different benchmarks programs. In Chapter 5 we discuss the known issues and limitations in the current implementation. Finally, we do a conclusion of the project and discuss the possibilities for future work in Chapter 6.

Chapter 2

Approach

2.1 Target Architecture

We implemented EMP for X86-64 Hardware. However, porting our current EMP implementation to a different platform should be trivial as long as LWIP and networking is supported by Barrelfish. The only platform dependent change we made to Barrelfish is an additional system call to get the machine ID from the kernel (see Section 3.3.1).

2.2 Structure

2.2.1 Library vs. Daemon

Our first decision was about the general architecture of EMP, whether we should implement the EMP functionality in a daemon like fashion which runs in its own dispatcher or if we should implement it as a library much like UMP or LMP.

	Pros	Cons
Daemon	Network connection set-up	Additional indirection layer
	Allows for more optimization	Daemon may become a bottleneck
	Can reuse UMP stubs	Single point of failure
Library	Better performance	Marshalling only in flounder
	Feels more natural in Barrelfish	Requires own flounder back end
	More robust	More complex channel set-up

Table 2.1: Comparison between daemon and library approach

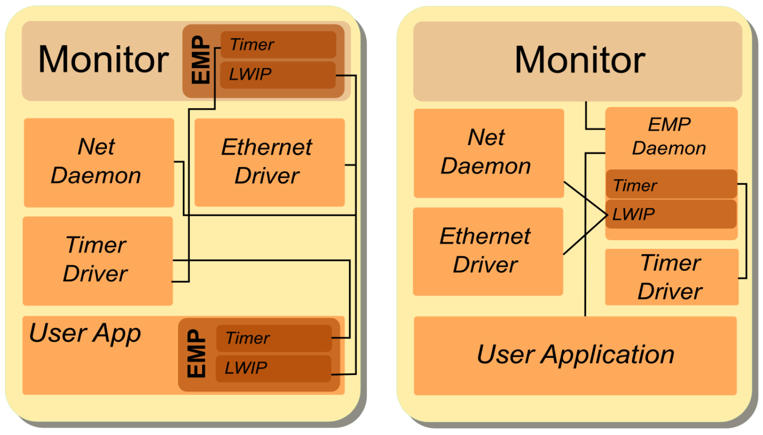

Figure 2.1: Architectural comparison between EMP as a library (on the left) and EMP as a daemon (right)

After comparison of the two approaches (see Figure 2.1 and Table 2.1) we decided to implement it as a library. The main reasons were that we expected it to perform better and because the implementation as a daemon would be kind of unnatural since all existing interconnect drivers are already implemented as part of libbarrelfish.

We started by implementing the EMP channel as a part of libbarrelfish, mainly because the other ICD drivers (UMP, LMP) also reside there. However, we soon ran into circular dependency problems because EMP depended on LWIP which in its turn depends on libbarrelfish. We then moved all the EMP code from libbarrelfish into a separate user-library. The downside of this is a less convenient integration with hake (see Section 5.3) and the need to expose some of the internal waitset functions to the outside of libbarrelfish.

2.2.2 Monitor Inter-machine Communication

In Barrelfish monitors on different cores exchange messages for various reasons, like exchanging interface references or clock synchronization, but also to set-up a connection between cores: Whenever a dispatcher wants to communicate with another dispatcher on a different core, the connection is set-up by their respective monitors communicating with each other first. This implies that we have to support a way for intra-machine message passing between monitors on different machines.

Before EMP, Barrelfish had support for at most intra-core messaging which on X86-64 was solved by having UMP channels between each monitor. For intra-machine messaging we explored two different options:

- Connect all monitors on all machines together
- Use a distinguished networking monitor per machine and route all inter-machine requests through this core

The first option means that every monitor needs to have networking capability and can directly send messages to a specific monitor on another machine. The amount of channels to set-up increases with the number of cores across all connected systems. This would lead finally to

$$\binom{total\ number\ of\ cores}{2}$$

connections. With an increasing amount of systems connected, this could become a problem. Especially given the expectation that the amount of cores per machine will increase a lot in the future, resources like the amount of available ports can become scarce.

On the other hand, this approach would reduce a level of indirection and some of the code complexity because bind requests and replies must not be routed. We tried to implement this approach first but it failed when we wanted to set-up LWIP in a monitor which does not run the networking processes. The reason for failure is a bug which made it impossible to send capabilities over the monitor's loopback binding.

We then chose to implement the second option where we used a distinguished monitor on every system to set-up EMP channels to all other machine. This implies that every inter-machine request has to be routed through the networking capable monitor using UMP first (see Figure 2.2).

2.2.3 Name Server

The name server (called chips in Barrelfish) provides functions to register interface references (irefs) with an associated name, which in turn can be used by clients to look up the reference after registration. This reference is used to establish a communication channel between two dispatchers later. Before EMP the name server only contained interface references for the local machine and it had no ability to communicate with other name servers. Since EMP enables us to communicate with services on distant machines, we needed to extend the system to allow lookups of interface references registered on remote systems as well.

We considered three different ways to do that:

1. Run a single name server across all machines
2. Run one name server per machine

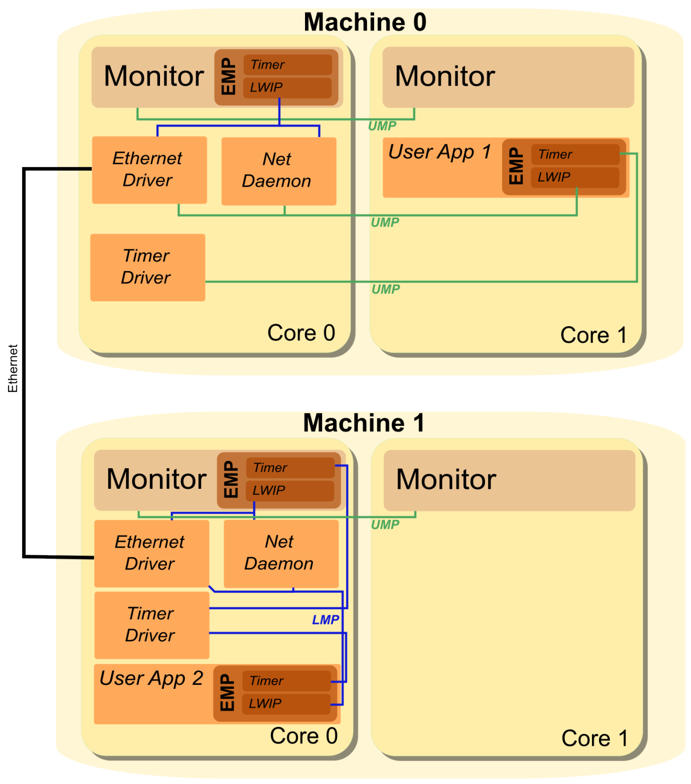

Figure 2.2: Architecture of two machines with two cores each communicating over EMP

3. Run one name server on each core

The first option requires only little changes to the name server itself. Because of EMP, clients could theoretically just establish a connection to a remote name server to store and retrieve interface references. However, some changes would be necessary in the boot process since chips is started earlier than the net daemon, Ethernet and timer driver which rely on chips for various service lookups. Although it would be possible to change the boot sequence, we decided against it since it would require some hacks to make sure that networking can be started without chips. In addition, this solution makes it a lot harder to guarantee fault tolerance in future work.

One name server per machine is what we chose to implement. Although it requires less changes to the initial boot sequence, since chips can be started before networking, we needed to add communication protocols between name servers to keep them synchronized (see also Section 3.4). The advantage is that it allows to build a more robust system where machines can fail without compromising the functionality of other machines in the same cluster.

The third option would also be a reasonable alternative since it means that the name server is fully distributed across all machines and cores. Conceptually, it is not a big step from the previous approach to this one. It is rather a trade off decision between system resources and reliability.

Chapter 3

Implementation

3.1 EMP Channel

The initial goal of the project was to do all the message passing using just Ethernet frames. The idea arised from the vision to be able to run a cluster of machines in a single rack where all of them would be directly connected to each other by a switch. Communication using just Ethernet frames would therefore lead to a minimal amount of overhead.

We decided in the initial phase of the project and after talking to our mentors to use UDP instead of plain Ethernet for the communication between multiple machines. The reasons for this decision were:

- The network driver currently does not allow user-space programs to install its own packet filters.
- We expect the performance gain using just Ethernet frames over UDP to be minimal.

Whereas for UMP the key concept behind it is shared memory, for EMP it's a network card which enables us to transfer a buffer of bytes from one location to another. Because changing a memory location in a shared frame is transparent for 2 processes due to cache coherency, in UMP there is no need to explicitly send a message. Whereas in EMP, we need to tell the network card explicitly if we want to send. This also implies that we need to have a way to allocate a buffer as well as clean up its memory after we sent it.

The comparison of these differences with the existing ICD backends lead us to the required guarantees our channel should fulfill:

Reliable message transfer No messages shall be lost during the transfer from one machine to another

In order message transfer Messages are delivered in the same order as they are sent

Even tough the handling of reliability and message ordering inside the flounder generated stubs (see Section 3.2) would also be a plausible alternative, we decided to provide theses guarantees by the channel itself. This leads to a design where an exchange of the underlying network protocol is possible without the requirement to change the flounder stub generator. Since some of these properties may already be guaranteed by the underlying network protocol, one can also take advantage of this circumstance.

Since we need reliable and in order transfer, we also thought about using just TCP/IP first. But given the previously mentioned scenario (cluster of machines in a single rack, minimal amount of network hops between them) and the fact that majority of the flounder messages are expected to be very short, we estimate our implementation to be faster than TCP/IP due to the overhead of the TCP protocol. Having our own protocol also integrates much nicer into flounder and leaves open more flexiblity for future work. It also means that the switch to just raw Ethernet can be done easier if desired at a later point in time.

The EMP API is kept very similar to the already present ICD backends. Nevertheless, the conceptual differences of EMP lead to some API differences in comparison with UMP. Since in a typical usage scenario the EMP API itself is wrapped by the flounder generated stub API, these differences do not influence the handling of the channel itself for the user process.

A crucial difference of EMP as compared to UMP is that waitset events are triggered directly by network events while the normal UMP variant uses polling. This saves us a certain amount of clock cycles which are otherwise wasted during polling in UMP. Contrary to UMP, an EMP endpoint uses two waitset events: One for incoming messages (as UMP does) and a second one to control the flow of message creation during sending.

The EMP API uses the emp_message structure which resembles a single UDP packet. What is finally sent over the network is wrapped in a emp_payload structure (see Figure 3.1) - which is a member of emp_message. emp_payload consists of a 32 bit header structure followed by the actual message data. The maximum size of the payload is restricted by a defined constant EMP_PAYLOAD_WORDS but the actual payload size can vary independently (see Figure 3.1).

The 32 bit emp_header structure contains a message type, sequence number and flounder type field. The type field is used to store the message number generated by flounder. The ctrl field is used by the underlying network implementation to store sequence numbers and to distinguish between ACKs and payload messages. These ctrl field values are used for reliability and ordering.

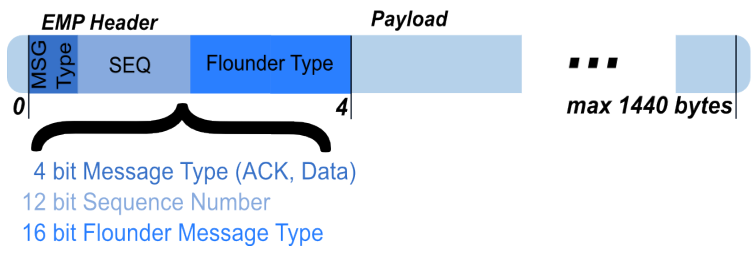

Figure 3.1: emp_payload - Structure of the messages sent over the network

To ensure those properties (see Section 3.1.2), we introduced two types of messages: Data and acknowledgement (ACK). To clearly identify and order the received messages, we attach a sequence number and already mentioned message type to each message. Therefore the ctrl field is divided into the message type and the sequence number. The only code-wise dependency between an emp_message and the underlying network implementation is this ctrl field in the emp_header structure.

3.1.1 LWIP

Our EMP Channel uses the LWIP library [2] to send and receive messages over UDP. We separated LWIP as good as possible from the EMP implementation so that it would be easy to switch to a different library or use a different network protocol. Message transmission and reliability are also independent from the EMP channel and endpoint.

LWIP uses packet buffers (pbufs) which can be allocated, filled with payload and passed on to the network card. In case of udp_send LWIP will prepend the necessary protocol headers to this buffer and forward it to the network card. Our emp_payload structure directly references the buffers memory location so we can avoid unnecessary copying. After the creation of the emp_message structure a channel implementation specific function is called to initialize the LWIP related parts of the message. Aside from setting up the pbuf, the initializing function also sets implementation specific function pointers for sending and freeing of messages. The later is necessary since the pbuf needs to be freed separately from the message.

Sequence numbers and resend timers used for reliability and ordering are tracked and stored per EMP endpoint in the respective emp_endpoint structure.

3.1.2 Reliability and Ordering

The UDP protocol does not provide ordering or reliability. Messages can get lost, arrive out of order or appear duplicated. Because all ICD in Barrelfish are based on ordered and reliable message passing, we had to build our own reliability protocol on top of UDP. We chose to do a version of Selective Repeat ARQ [4].

For congestion control we introduced a sliding window with sequence numbers. If the sequence number of a message is not in that window, it is enqueued and as soon as it will become part of the window, it will be sent. The window starts at the longest not acknowledged message and is of fixed, but configurable size.

The packet buffer is not discarded after sending, but instead kept with the complete message structure in a queue until the message is acknowledged by the receiver. The resending of messages is initiated by a timer which fires continually in a configurable interval. Whenever the timer expires, all messages in the sending window are resent.

For message retransmission, a resend decision handler function is called to check if the message should be resent or if the channel should stop doing further resending attempts. This resend decision handler function is provided by the client of the channel. It was introduced to inform clients when messages stop arriving, because sending a message does not provide the client with immediate feedback about transmission success or failure.

At the receiver, incoming messages are stored in a priority queue and are passed in order to the client. After receiving a data message, an ACK message containing the received sequence number is sent back to the sender. Duplicated messages are dropped but still acknowledged.

3.2 Flounder Stub Generator

When we first started writing the channel, we wrote the stubs which flounder normally would generate by hand to get a better understanding of how the stubs work. After the channel implementation was more or less working and with a stable API, the flounder stub compiler was extended by an EMP backend to generate EMP headers and stubs. The four standard control functions (can_send, register_send, change_waitset, control) as well as the rest of the flounder stub api are implemented according to the specification in the Barrelfish Technical Note 011 ("Inter-dispatcher communication in Barrelfish").

We basically adapted the UMP flounder backend and had to modify most of the function interfaces according to our needs. Every interface defined with the flounder IDL can be used to generate the corresponding EMP headers and stubs. The IDL (Interface Definition Language) was not extended or changed in any way.

3.2.1 RPC

Besides standard asynchronous messaging, Flounder has support for synchronous messaging in the form of Remote Procedure Call (RPC) calls. RPC calls are implemented on top of asynchronous messaging. If a client wants to use RPC, he has to set-up an RPC client for a given service binding. When doing an RPC call the client will change the waitset for the binding to its own internal waitset and send a message to the server. Then it will handle incoming events on its own waitset and wait for a reply from the server before the RPC call returns in the client. The problem with EMP is that an incoming message will trigger a receive event on the waitset where LWIP is registered. By changing only the binding of our EMP channel we are no longer able to receive any messages since the binding between LWIP and the Ethernet card is registered on the default waitset.

We did not really come up with a good solution for this problem other than changing the waitset of LWIP internal bindings as well as the bindings used by the timer library (to make sure resend events continue to happen) every time we change a waitset for an EMP binding. This does not break RPC semantics as long as only the EMP channel uses the timer and LWIP library in the dispatcher. But as soon as they are used for other purposes, those events will be handled as well during an RPC call.

3.3 Monitor

3.3.1 Machine ID

Because there are multiple machines communicating with EMP we now needed a way to distinguish them. We added a unique number, called machine id, to the Barrelfish kernel. The number is set in the menu.lst file as an argument to the CPU module. Programs can do a system call in the kernel to obtain the ID.

The ID is used to encode the machine location in the interface reference (Section 3.3.2), for static channel set-up in the name server (Section 3.4) and for clients to tag service names (Section 3.4.1) to make them unique across multiple machines.

3.3.2 Interface Reference

Once a service has registered itself in the local monitor, it is ready to accept incoming connections. The monitor will provide the dispatcher exporting the service with an interface reference (iref) which uniquely identifies the service in the Barrelfish instance. If a client wants to connect to a service a decision is made by the monitor which ICD driver should be used. This decision is based on the iref because it encodes information on which core the service is registered and where the monitor can find the information it stores about the service. We extended the iref to encode the ID of the machine as well. This makes the irefs unique among all machines.

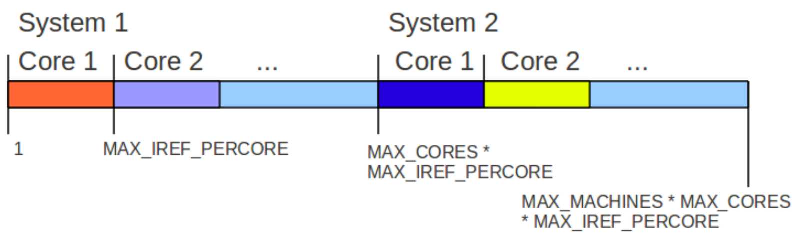
System 1
Core 1
Core 2
...
System 2
Core 1
Core 2
...
1
MAX_IREF_PERCORE
MAX_CORES * MAX_IREF_PERCORE
MAX_MACHINES * MAX_CORES * MAX_IREF_PERCORE

Figure 3.2: Interface reference distribution with multiple machines

3.3.3 Intermachine Communication

As mentioned in the previous chapter, a connection between two endpoints is always initialized by their respective monitors talking to each other first. The only exception for this is the communication between monitors and name servers. For the inter-monitor communication we obviously needed a static channel set-up. At the moment this is done by a table which maps machine IDs to IP addresses. The local bind and remote port are chosen based on the machine ID as well. For future work it would be nice to have support for dynamic detection of available machines. Then it would also be possible to transfer the interface reference of the remote name server over the existing monitor communication channel to get rid of the static setup in chips.

As discussed in Section 2.2.2 we connect only one designated monitor in a machine with every other machine using EMP channels. Having networking available in the monitor means that the network driver, the timer driver and the net daemon (netd) have to be started at boot time (prefixed with boot in menu.lst). Since the name server might not be ready at the time, we had to change some of the networking code to send the monitor the necessary interface references directly.

3.3.4 EMP Bind Process

Once we were able to use EMP in the monitor we started writing the interaction protocol (as defined in the intermachine.if interface) between monitors on different machines. Currently we only exchange messages for EMP channel construction. The binding process looks a lot like a UMP bind process but with an additional level of indirection.

Figure 3.3 shows the steps and the involved actors during the EMP bind phase:

1. A client sends the iref and its listening port to its monitor. A free port is chosen at random by netd.
2. The monitor will extend this information with its core and machine id and forwards the request to the EMP capable monitor.
3. The EMP capable monitor can figure out the remote server location based on the iref and forward the request to the remote server using an EMP communication channel.
4. The receiving monitor will forward the request to the correct local monitor based on the core ID which is also read from the interface reference.
5. The local monitor will store the remote machine and core id and send a bind service request to the server.
6. The server will construct an EMP channel and connect it to the remote server based on the machine ID and the port passed on by the local monitor. The server then sends an error value and its local bind port (in case of successful channel creation) back to the his monitor.

7-9. The error value and the servers listening port are sent back to the client monitor.

10. The client monitor forwards the error value and the remote port to the client.

In case the client and/or server runs on core with the networking capable monitor, the additional routing steps are skipped.

3.4 Nameservice

In Barrelfish the name server (chips) is implemented on top of a hash table which maps strings to interface references. As mentioned in Section 2.2.3 we use one name server per machine. Having multiple name servers leads to some additional problems:

First, name servers need some way to communicate with each other to exchange information about newly registered interface references or removed references to make it possible to lookup irefs registered on distant machines.

And secondly, their state needs to be kept consistent among all name servers. This is especially difficult in case we have two dispatchers on different machines, which register a service with the same name simultaneously. In that case at least one of them should return an error to the user. More generally speaking, we want the hash table to be consistent (i.e. never have two entries with same name but a different reference over all machines). Before Barrelfish allowed registrations of the same name with different irefs and would always return the latest registered iref for a particular name. After talking to our mentor we decided to change this policy and do not allow it anymore.

3.4.1 Master Slave Replication

We implemented the name server using a master/slave pattern. In combination with EMP, chips can operate in three different modes: Standalone, Slave and Master. After booting, chips starts running in standalone mode where the behavior is no different from the existing implementation in Barrelfish. This allows all programs spawned before networking to do lookups or registrations before networking is available. These registrations will later be replicated across all machines.

After networking is available we decide based on the machine id (see Section 3.3.1) what mode the name server will operate in. For convenience, we defined the name server running on machine 0 to be the master and all other machines to be slaves.

A slave will subsequently try to contact its master as soon as networking is up. If the master comes on-line, he will send a merge request to the slave. During the merge phase, the master and slave will try to merge their respective hash tables to bring them in a consistent state which means that both hash tables will contain the entries of all machines currently being connected including the ones of the newly accepted slave. A conflict during this phase (i.e. master and slave both having an interface reference with the same name) will result in a rollback on the master and an abort on the slave.

We changed the service names of daemons like spawnd and device drivers to always include the core id (if necessary) and machine id at the end of their name. This way we can avoid conflicts during the merge phase. Once the merging is complete, service registrations and removals on slaves need to be accepted by the master which will broadcast the request to all other slaves to keep them consistent or return with an error if a service is already registered under the same name. This avoids future conflicts, since a registration or removal request on any of the name servers is now coordinated by the master.

3.5 Spawn Daemon

One goal of the project was to implement spawning of dispatchers across two different machines (i.e. one machine creates and starts a dispatcher on another machine). Once we had the EMP channel, the distributed name server and the Flounder stub generator for EMP, we enabled EMP in the spawn daemon and used the existing interface to communicate with the daemon across machines. Because the library functions for spawning new dispatchers reside in libbarrelfish, we could not extend them. As a user-library, EMP can not be used in libbarrelfish (see Section 2.2.1). Instead we wrote a demonstration program called remote_spawn which allows to spawn a dispatcher on any remote machine in the cluster. A perhaps better approach would be to enable EMP in a dedicated spawn daemon on every machine and route local spawn requests for remote machines through this daemon (similar to the approach used in the monitor).

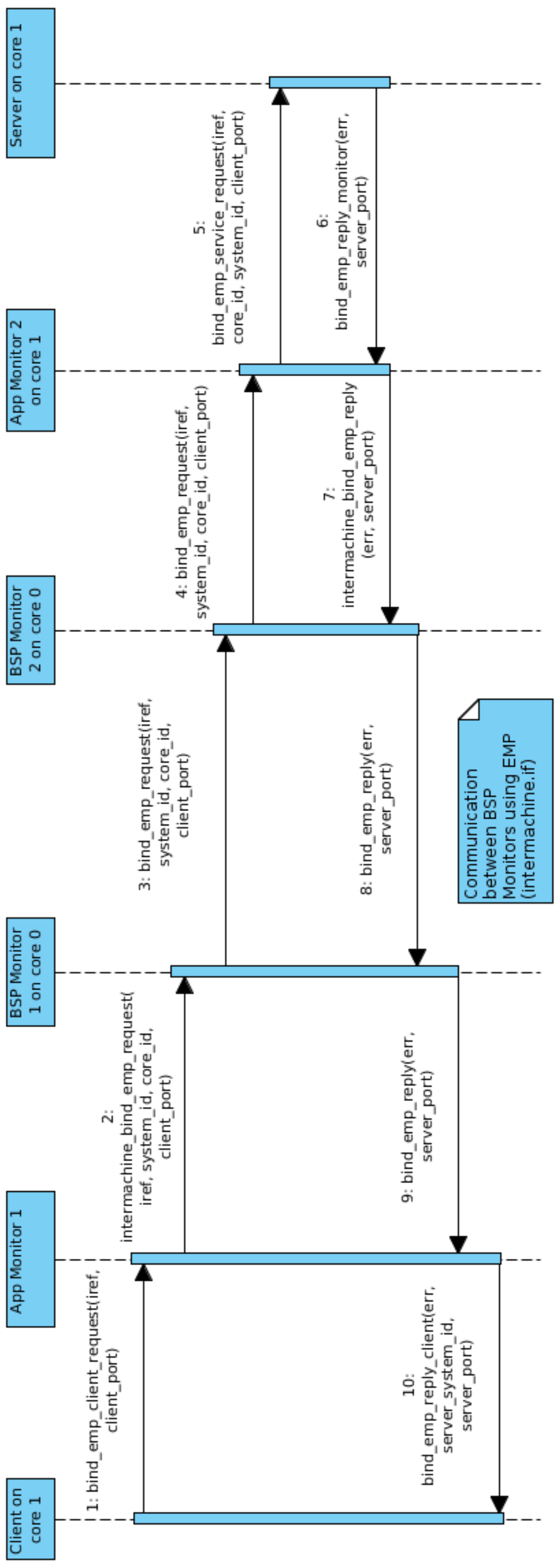
Client on core 1
App Monitor 1
BSP Monitor 1 on core 0
BSP Monitor 2 on core 0
App Monitor 2 on core 1
Server on core 1
1: bind_emp_client_request(iref, client_port)
2: intermachine_bind_emp_request(iref, system_id, core_id, client_port)
3: bind_emp_request(iref, system_id, core_id, client_port)
4: bind_emp_request(iref, system_id, core_id, client_port)
5: bind_emp_service_request(iref, core_id, system_id, client_port)
6: bind_emp_reply_monitor(err, server_port)
7: intermachine_bind_emp_reply (err, server_port)
8: bind_emp_reply(err, server_port)
9: bind_emp_reply(err, server_port)
10: bind_emp_reply_client(err, server_system_id, server_port)
Communication between BSP Monitors using EMP (intermachine.if)

Figure 3.3: Exchanged messages for EMP connection setup between client and server running on core 1 and EMP capable monitors running on core 0

Chapter 4

Evaluation

4.1 Benchmarks emp_bench

Since the message transport of EMP is quite different from UMP's exchange of messages over shared memory, we implemented some new benchmark applications for EMP. Those benchmarks all use the flounder generated stubs for message passing and can therefore be executed between different cores on a single machine using UMP as well as with EMP between two seperated, phyiscal machines.

When comparing with UMP, one should keep in mind that some of the performance wise disadvantages of EMP are due to the overhead caused to guarantee reliability on UDP. The presented measurements are all done between two directly connected machines in a single rack. We did not encountered any loss of UDP packets during the benchmarking process, so no message retransmissions influence the measured times.

Every emp_bench benchmark has a client and a server side. The measurements are always taken at the client side. The server side basically just acts as mirror (emp_pingpong) or as a receiver which sends back a single message answer on completion (emp_throughput).

The presented values were measured between the two Systems Group rackservers ″nos5″ and ″nos6″ at ETH Zurich. Both of them are running on two AMD Santa Rosa (Opteron 2200) and are equiped with an Intel Corporation 82572EI Gigabit Ethernet Card. The two servers are connected to a HP ProCurve 2510G-48 rackmount switch. The machine ″nos5″ always took the server role while ″nos6″ was acting as the client for our benchmarks.

	mean (ns)	mean (cycles)	std deviation (ns)	std deviation (cycles)
emp_pingpong (EMP)	130.0	365456.3	0.991	2711.2
emp_pingpong (UMP)	-	1106.0	-	60.4
linux ping	124.0	-	2.5	-

Table 4.1: emp_pingpong benchmark results

4.1.1 emp_pingpong

This benchmark exchanges basic massages (a single integer) between the client and the server. The clients sends a ”ping” message to the server and waits for the server's ”pong” message. The measured value equals the timespan between the call of the flounder send handler for the ”ping” message and the occurrence of the receive callback for the ”pong” message. In other words: The round-trip time. The measurement included 1100 ping-pongs. For the resulting values, the first 100 measurements were discarded.

This measurement was also done using UMP. For this purpose, the client and server process were spawned on two different cores on a single machine.

For comparison, we measured the time of an ICMP ping request between the client and server. For this purpose, we booted the client machine (”nos6”) with a linux kernel and used linux's ”ping” command against the server machine (”nos5”) running Barrelfish to measure the round-trip time of an ICMP ping pong.

As the comparison between the emp_pingpong benchmark results using EMP and the linux ping time shows, the overhead of the EMP channel does not have a significant impact on the networking performance. Since the measured times are extremely similar, we come to the conclusion that the determining factor for the response time of our EMP channel implementation is the performance provided by the networking infrastructure.

Running the same benchmark over a UMP channel (on the same machine) results in significant smaller cycles counts below any reasonable nanoseconds scale. Given the profound differences between the two channel types (see Section 3.1) and the simple fact, that benchmarking with the use of UMP on a single machine does not cause any buffer allocation or networking overhead to message passing, this performance gap is no surprise.

4.1.2 emp_throughput

This benchmark sends a buffer of configurable size from the client to the server. As soon as the buffer is completely received, the server sends back

Buffers size (MB)	mean throughput (MB/s)	std deviation (MB/s)	measurments
1 MB	5.545	0.046	110
10 MB	5.570	0.014	110
20 MB	5.569	0.008	55
50 MB	5.576	0.004	55
100 MB	5.572	0.005	55

Table 4.2: emp_throughput benchmark results

an empty "pong" message to the client. The measured value equals the timespan between the call of the flounder send handler for the buffer message and the occurrence of the receive callback for the "pong" message. For the calculations, the first 10% of all measurements were discarded.

Independent of the buffer size, the throughput value is constant. This clearly fulfills our requirements on the EMP channel. But given the network infrastructure (single rack with directly connected machines), the throughput values we achieved are without any doubt below our expectation. We ran out of time to fix it but we managed to pinpoint the problem. A detailed discussion can be found in Section 5.1.

Chapter 5

Known Issues and Limitations

5.1 Throughput

As described in Section 3.1.2, we implemented reliability and ordering on top of UDP with a sliding window protocol. This was expected to provide a significant performance gain over simpler protocols without a sliding window.

Our emp_throughput benchmarks (see Section 4.1.2) resulted in significant lower throughput compared to TCP/IP. Further investigations, debugging and the use of barrelfish's tracing framework lead us to the conclusion that LWIP's udp_send function does not behave as expected.

Without going into further detail, the sliding window basically ensures (or should insure) that the event of receiving an acknowledgement immediately causes the sending of a new message / another UDP packet to the network card driver (if there are any unsent messages left in the send queue). This is what should happen when calling LWIP's udp_send function when receiving an acknowledgement.

Unfortunately the call to udp_send does not trigger an immediate UMP message to the network driver. Instead, the UMP messages directed to the network driver are sent as a bulk in sporadic time intervals. The visualized trace of an emp_throughput process clearly emphasizes this fact. (see Figure 5.1).

This behavior sooths any potential performance gain using a sliding window down. As Figure 5.2 illustrates, the intended behavior (left side) has a significant advantage over the current behavior (right side). The red dots on the side mark the actual time, when udp_send is called in our implementation.

These circumstances make it impossible to achieve higher throughput values. As already mentioned we ran out of time to find the cause for this send delay.

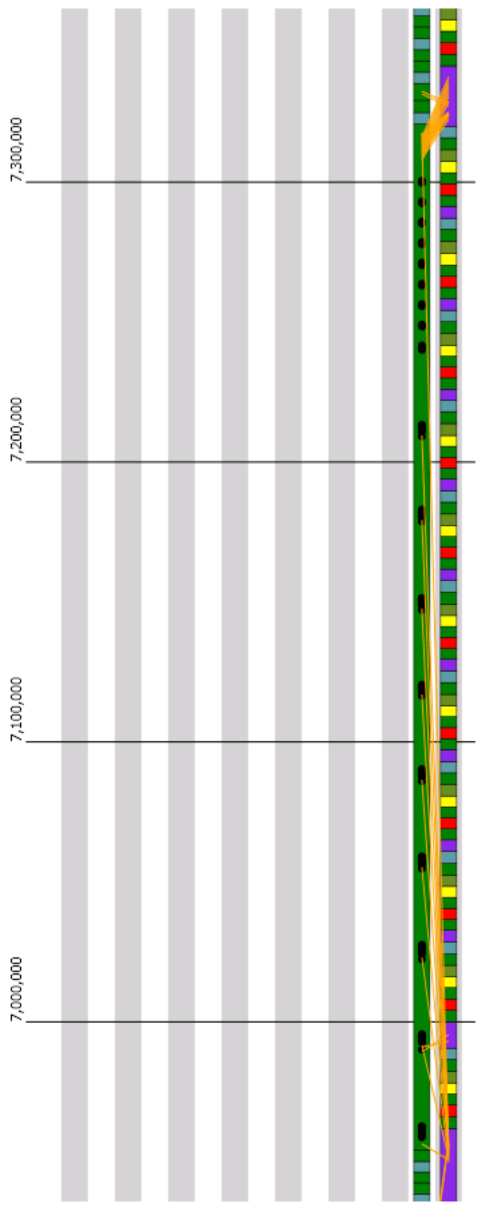

Figure 5.1: Aquarium visualization of emp_throughput trace

At the moment we suspect either LWIP deferring the sending on purpose or some blocking due to issues with the waitset. We suppose that our sliding window implementation will perform significantly better if UDP packets would be delivered immediately to the network driver.

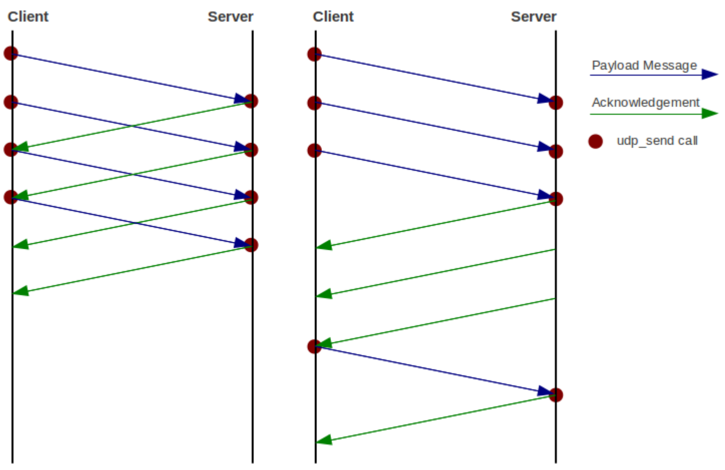

Figure 5.2: Sliding window with window size 3 - intended vs. current behavior

5.2 User-Library Limitations

Since EMP is implemented as a user-library (see Section 2.2.1), EMP can not be used from within libbarrelfish. This is especially limiting for remote spawning (as explained in Section 3.5) since the spawn client code currently resides in libbarrelfish.

5.3 Hake Integration

The integration of EMP in hake is not very straightforward at the moment. As explained in Appendix A, the use of EMP in a user program needs some modification to the Hakefile.

The EMP flounder stubs are generated for all present interface definitions but not compiled because hake is modified not to set the corresponding precompiler flags "CONFIG_INTERCONNECT_DRIVER_EMP" and "CONFIG_FLOUNDER_BACKEND_EMP". In order to use EMP in a user program, these two flags needs to be added within the user program's Hakefile.

A cleaner integration of EMP to hake and flounder would probably involve some dedicated configuration syntax for the Hakefile and would only generate the EMP flounder stubs where actually needed.

5.4 Channel termination

A lot of issues remain for proper cleanup of a communication channel after the channel is terminated. As this is currently not addressed in UMP and involves a lot of additional problems, we did not worry about it as well in this project.

5.5 System Discovery

At the moment, every machine which is part of the cluster needs to be setup with its machine ID and IP address in a static list. The list of participating machines is fixed at compile time.

This clearly could be improved to support some form of discovery of participating machines inside a given subnet or range of IP addresses. One solution could be the implementation of a dedicated daemon which announces a booting system to an existing set of cluster machines using broadcasting and registering of EMP channels for its monitor & name server on discovery. This would remove the static setup we have at the moment.

Chapter 6

Conclusion

With EMP we provide a new ICD for Barrelfish which is able to exchange messages over the network. Our current implementation uses the LWIP library for the network stack. The dependencies on LWIP are minimal which means that EMP could easily be changed to use a different library. We extended Flounder to generate stubs for EMP which enables us to use all the existing interfaces or newly written ones across multiple machines. The performance evaluation of EMP has shown the latency differences between cross machine and cross core messaging as well as the current achievable throughput of our system. We discussed different architectures for the inter-monitor and inter-name server communication, described our implementation and changes to Barrelfish which makes it possible to run multiple Barrelfish machines and have their dispatchers communicate not only cross core but also across machines. We enabled EMP in the spawn daemon which allows to spawn programs from a remote Barrelfish instance. We managed to point out some problems which would need to be addressed by future work to make the system more viable.

6.1 Future Work

As we pointed out in the Evaluation (Chapter 4) and the Known Issues (Chapter 5) the biggest flaw in EMP itself currently is the throughput which should be a lot higher.

Other than that we did not address any security issues that can arise from different monitors communicating with each other over a network. Also, we don't have any protocols that deal with failure of machines. This would require some additions to the current master slave implementation of the name server and monitor inter-machine communication. We decided to ignore capability transfer over EMP. Flounder will generate the stubs for this but as soon as the monitor is called to transfer the capability, we abort.

Appendix A

System Setup

To use EMP in a user program, some modifications to the barrelfish toolchain and a few configuration changes have to be made. This is due to the fact that EMP is not (yet) fully integrated into Barrelfish and it has some dependencies on other user-space libraries. Those modifications are already present in the corresponding files on our barrelfish branch.

A.1 Modifications to Haskell

A.1.1 hake/X86_64.hs

Replace

```
optInterconnectDrivers = ["lmp", "ump"],
optFlounderBackends = ["lmp", "ump"],
```

with

```
optInterconnectDrivers = ["lmp", "ump", "emp"],
optFlounderBackends = ["lmp", "ump", "emp"],
```

A.1.2 build/hake/Config.hs

Add the following lines

```
-- Maximum supported number of machines (used for EMP)
max_servers :: Integer
max_servers = 2
```

Replace

```
"MAX_CPUS=" ++ show max_cpus
```

with

```
"MAX_CPUS=" ++ show max_cpus,
"MAX_SERVERS=" ++ show max_servers
```

Replace

```
-- enable config flags for interconnect drivers in use for this arch
= [ Str ("-D" ++ d)
    | d <- ["CONFIG_INTERCONNECT_DRIVER_" ++ (map toUpper n)
    | n <- optInterconnectDrivers opts]
]
-- enable config flags for flounder backends in use for this arch
++ [ Str ("-D" ++ d)
    | d <- ["CONFIG_FLOUNDER_BACKEND_" ++ (map toUpper n)
    | n <- optFlounderBackends opts]
]
```

with

```
-- enable config flags for interconnect drivers in use for this arch
= [ Str ("-D" ++ d)
    | d <- ["CONFIG_INTERCONNECT_DRIVER_" ++ (map toUpper n)
    | n <- filter (\x -> x /= "emp") $ optInterconnectDrivers opts]
]
-- enable config flags for flounder backends in use for this arch
++ [ Str ("-D" ++ d)
    | d <- ["CONFIG_FLOUNDER_BACKEND_" ++ (map toUpper n)
    | n <- filter (\x -> x /= "emp") $ optFlounderBackends opts]
]
```

With this modification, the flounder stubs for EMP will be generated but the corresponding compile and linker flags will not be set. A cleaner integration of EMP to hake and flounder would specify some dedicated configuration syntax for the Hakefile.

A.2 Modifications to the user process' Hakefile

For any user process which wants to use EMP, the following must be present in its Hakefile

```
addLibraries = ["emp", "lwip", "contmng", "timer"],
addCFlags = ["-DCONFIG_INTERCONNECT_DRIVER_EMP", "-DCONFIG_FLOUNDER_BACKEND_EMP"]
```

A.3 Modifications to lib/emp/ip_table.c

To add machines, just modify the following part at the end of ip_table.c according to your needs.

```
// Static setup, add machines here...
// use emp_add_machine_ip([machine id],[system IP address]);
IP4_ADDR(&address, 10, 0, 2, 11);
emp_add_machine_ip(0, &address);
```

```
IP4_ADDR(&address, 10, 0, 2, 12);
emp_add_machine_ip(1, &address);
```

A.4 Modifications to menu.lst

An example menu.lst configuration to use EMP

```
timeout 0
title   Barrelfish
root    (nd)
kernel  /x86_64/sbin/elver loglevel=4 machineid=0
module  /x86_64/sbin/cpu loglevel=4 machineid=0
module  /x86_64/sbin/init

# Domains spawned by init
module  /x86_64/sbin/mem_serv
module  /x86_64/sbin/monitor

# Special boot time domains spawned by monitor
module  /x86_64/sbin/chips boot
module  /x86_64/sbin/ramfsd boot
module  /x86_64/sbin/skb boot
modulenounzip /skb_ramfs.cpio.gz nospawn
module  /x86_64/sbin/pci boot
module  /x86_64/sbin/spawnd boot
#bootapic-x86_64=1-15
module  /x86_64/sbin/startd boot

## For networking
module  /x86_64/sbin/lpc_timer boot
## For qemu, enable rtl8029
module  /x86_64/sbin/rtl8029 boot
## For real hardware, enable e1000n
# module        /x86_64/sbin/e1000n boot
module  /x86_64/sbin/netd boot

# General user domains
module  /x86_64/sbin/serial
module  /x86_64/sbin/fish
```

The important parts are

machineid=X This sets the machine ID of the current system. This ID is used to identify the system for cross-machine communication and must be unique within a group of communicating machines.

Some ″boot″ additions lpc_timer, the network card driver (rtl8029 for Qemu, e1000n for real hardware) and netd must have the ″boot″ argument added because of EMP's dependencies on these services.

Bibliography

[1] Barrelfish frequently asked questions. http://www.barrelfish.org/bffaq.html.

[2] lwip - a lightweight tcp/ip stack. http://savannah.nongnu.org/projects/lwip/.

[3] Adam Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001:20, SICS – Swedish Institute of Computer Science, February 2001. Master's thesis.

[4] Andrew S. Tanenbaum. *Computer networks*. Prentice Hall, 2003.