Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Bachelor's Thesis Nr. 24b

Systems Group, Department of Computer Science, ETH Zurich

A Routing and Forwarding Subsystem for a Multicore Operating System

by

Alexander Grest

Supervised by

Akhilesh Singhania, Prof. Timothy Roscoe

August 2011

inf | Informatik
Computer Science

Abstract

The architecture of computer systems is radically changing: core counts are increasing, systems are becoming more heterogeneous and the memory system is becoming less uniform. One of the proposed implication of this for operating systems design is the use of a message-passing primitive instead of shared-memory for communication. Message passing happens generally over direct point-to-point channels between processor cores which can directly exchange data. However, this may not be possible in machines with partial connectivity where not all cores can directly exchange data. Furthermore, it may be undesirable in cases where communication is expensive in latency or bandwidth.

In this thesis, we demonstrate a routing infrastructure for the Barrelfish operation system that provides multi-hop messaging between cores. This allows communication between all cores in machines with partial connectivity and optimizes resource utilization by multiplexing multi-hop channels over existing communication channels. We further present the appropriate abstractions and performance tradeoffs involved in multi-hop messaging inside a multicore machine.

Contents

Chapter 1

Introduction

1.1 Motivation

The architecture of computer systems is radically changing: core counts are increasing, systems are becoming more heterogeneous and the memory system is becoming less uniform [4]. A modern computer resembles undeniably a distributed system and exhibits features traditionally found in distributed systems, such as node heterogeneity, dynamic changes due to partial failures and communication latency.

One of the proposed implication of this for operating systems design is the use of a message-passing primitive instead of shared-memory for communication [2]. Traversing a shared data structure in a modern cache-coherent system is equivalent to a series of synchronous RPCs to fetch remote cache lines. In cases where communication is expensive (in latency or bandwidth), it is more efficient to send a compact message encoding a complex operation than to access the data remotely. A message-passing primitive can therefore make more efficient use of the interconnect, because it encodes high-level operations more compactly. Furthermore, the use of message-passing rather than shared data facilitates interoperation between heterogeneous processors.

Generally, message passing happens over direct point-to-point channels between processor cores which can directly exchange data. However, there are at least two motivations for extending this:

1.1.1 Partial connectivity

Most current multicore machines are fully connected via shared memory. This means that any core in the system can communicate with any other core in the system by using shared memory. Nevertheless, it is possible that not all cores can directly exchange data in modern hardware, illustrated by the following two examples:

- Consider a PCIe-based channel which connects a single core on an x86 machine to a single core of a Intel Single-Chip Cloud Computer. In this case, only the two connected cores can exchange data directly.
- If a single operating system image is operated on a cluster of machines, there is only an Ethernet-based channel between the core(s) where the network stack is running. In order to allow every core to communicate with every other core, the available link must be multiplexed.

These are two examples of current set-ups which lead to partial connectivity. Indirect message passing will allow applications to communicate in such environments without having to worry about the concrete path a message takes from one core to another.

1.1.2 Resource usage

Communication channels that allow message passing between cores might be expensive. Multiplexing multiple channels over a single channel will optimize resource usage, while at the same time increase messaging latency. For example, consider a system with four cores, as illustrated in figure 1.1: If we want to allow direct message passing between all cores, we have to create six communication channels. If we use indirect message passing, we can reduce the number of required communication channels to three.

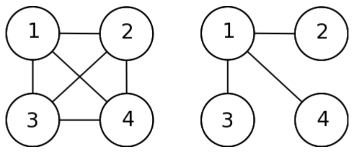

Figure 1.1: Communication between four cores

1.2 Aim

The aim of this thesis is to experiment with the design of a multi-hop channel that will allow indirect message passing. We will demonstrate the design of such a multi-hop channel and a routing infrastructure using the Barrelfish operating system. Moreover, we will identify the appropriate abstractions and performance trade-offs involved in multi-hop messaging.

1.3 Overview

This thesis is organized as follows: Chapter 1 has introduced the topic. Chapter 2 gives an overview of the Barrelfish operation system, focusing on the parts of Barrelfish relevant to message passing. This is mainly based on the literature [1], [2] and [3]. Chapter 3 presents the multi-hop interconnect driver, which allows indirect message passing in Barrelfish. Chapter 4 describes the integration of the multi-hop interconnect driver with the *Flounder*-stub compiler. Chapter 5 contains a performance analysis of the multi-hop interconnect driver. Finally, Chapter 6 gives an overview of possible directions for future work.

Chapter 2

The Barrelfish Operating System

Commodity computer systems resemble more and more networked systems and exhibit features traditionally found in such systems: node heterogeneity, dynamic changes due to partial failures and other reconfigurations and communication latency [2]. Barrelfish is a research operating system developed in cooperation between the Swiss Federal Institute of Technology Zurich (ETH) and Microsoft Research. It embraces the networked nature of the machine and rethinks operating system architecture using ideas from distributed systems.

Barrelfish is an implementation of the multikernel architecture. In a nutshell, the operating system is structured as a distributed system of cores which communicate using messages and share no memory. The multikernel architecture is guided by three design principles: Make all inter-core communication explicit, make OS structures hardware-neutral and view state as replicated instead of shared [3]. These principles allow the operating system to benefit from the distributed systems approach.

2.1 High level architecture overview

The structure of Barrelfish is depicted in figure 2.1 [3]. Each core runs a kernel which is called a ”CPU driver”. The CPU driver runs in privileged mode and enforces protection, performs authorization, time-slices processes, and handles interrupts, pagefaults, traps, and exceptions. It is single threaded, event driven and nonpreemptable.

On every core runs a distinguished user-mode *monitor* process. All inter-core coordination is performed by monitors. Monitors collectively coordinate system-wide state and encapsulate much of the mechanism and policy to be found in a typical monolithic kernel. On each core, replicated data structures, such as memory allocation tables, are kept globally consistent by means of an agreement protocol run by the monitors.

A process is represented by a collection of dispatcher objects, one on each core on which it might execute. Dispatchers on a core are scheduled by the local CPU driver, invoking an upcall interface that is provided by each dispatcher.

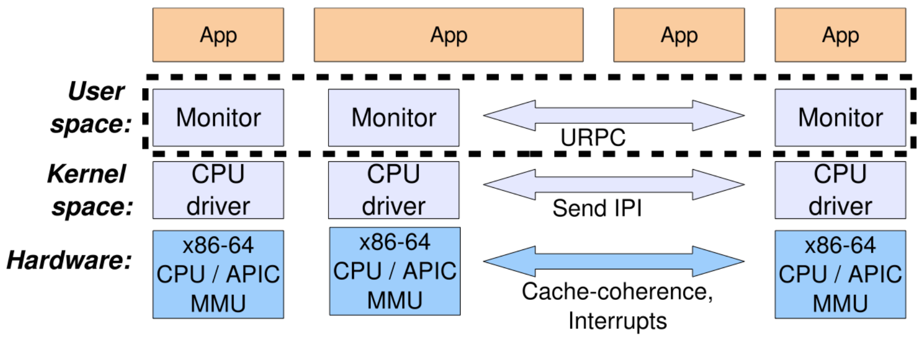

Figure 2.1: Barrelfish structure. Source [3]

2.2 Capabilities

The security model of Barrelfish is inspired by the seL4 microkernel. Access to resources is managed with capabilities. A capability represents some right in the system. The actual capability can only be directly accessed and manipulated by the kernel, while user level code has access to capability references only.

2.3 Inter-dispatcher communication

Communication in Barrelfish is not between processes but between dispatchers and hence cores. All inter-dispatcher communication occurs with messages. Messages are carried over *Interconnect Drivers* (ICDs), specialized messaging subsystems which carry small data units between dispatchers.

Interconnect drivers are highly optimized for particular hardware and do not expose a standard interface. Instead, interconnect drivers are abstracted behind a common interface, allowing messages to be marshalled, sent and received in a driver-independent way. As with conventional RPC systems, the interface for a particular communication binding is specified in an Interface Definition Language. A stub compiler, called *flounder*, compiles defined interfaces into a set of ICD-specific stubs.

2.3.1 Local message passing (LMP)

Communication between dispatchers on the same core is performed using LMP (local message passing). LMP uses endpoint capabilities for communication. The channel contains a reference to a remote endpoint capability (for the sender) and a local endpoint capability (for the receiver). When a message is sent, the sender invokes the remote endpoint capability with the message to send. This causes the kernel to copy the message into the associated endpoint buffer. The LMP interconnect driver is the only driver which is always present.

2.3.2 Inter-core user-level message passing (UMP)

The only inter-core communication mechanism available on some hardware platforms is cache-coherent memory. The current version of Barrelfish therefore uses a variant of user-level RPC (URPC) between cores: a region of shared memory is used as a channel to transfer cache-line-sized messages point-to-point between single writer and reader cores. The shared memory is split into a send buffer and a receive buffer. The sender writes into the send buffer, and the receiver reads from it. The receiver polls to determine whether new messages have been received.

As the performance of inter-core messaging is critical for a multikernel operating system, the implementation is carefully tailored to the cache-coherence protocol to minimize the number of interconnect messages used to send a message [3].

Chapter 3

The Multi-Hop Interconnect Driver

At present, all communication in Barrelfish happens over direct point-to-point ICD links. The multi-hop interconnect driver developed in this thesis gives applications the possibility to create a logical link between two dispatchers (called a *multi-hop channel*) which is multiplexed over existing ICD links.

A multi-hop channel can only be set up between two dispatchers running on different cores. It always leads through the two monitors running on each dispatcher's core. Between those two monitors the multi-hop channel can lead through an arbitrary number of additional monitors. We call all the monitors that lie on a multi-hop channel *nodes*. All the nodes of a multi-hop channel must be connected by means of other ICD-links (such as LMP or UMP ICD-links).

Once a multi-hop channel is set up, it can be used to exchange messages between the two dispatchers. The multi-hop channel transports messages by passing them to the underlying interconnect driver on each link between the nodes of the multi-hop channel.

The multi-hop interconnect driver consists of

- A mechanism to set up new multi-hop channels between dispatchers addressed by end-point identifiers
- A mechanism to send messages along a multi-hop channel
- A mechanism to receive messages from the channel

3.1 Design goals

3.1.1 Independence of the underlying interconnect driver

The multi-hop interconnect driver was designed to be independent of the type of the underlying ICD links between the nodes on the multi-hop channel. This means that it uses the common flounder interface supported by all ICDs when interacting with the underlying ICD link and uses no ICD-specific knowledge. This design involves a performance penalty: Interacting directly with the underlying ICDs instead of via the common flounder-interface would certainly perform better. Nevertheless, we chose this design, as it gives us more flexibility: The multi-hop interconnect channel can run over all present and future interconnect drivers in Barrelfish, as long as they support the common flounder interface.

3.1.2 Reliability

Interconnect drivers in Barrelfish generally provide a reliable messaging service: A message is delivered only once and each message sent is eventually delivered and its content is not corrupted. Furthermore, messages are delivered in FIFO order. The multi-hop interconnect driver is designed to provide a reliable messaging service in principle. However, contrary to the end-to-end argument, it does not provide any *end-to-end* reliability, but builds on the reliability provided by the interconnect drivers of the underlying links. We accept that the multi-hop interconnect driver can fail in case any of the interconnect drivers of the underlying link fail.

3.1.3 Resource usage

Because it is our goal to optimize resource usage (see section 1.1.2), the multi-hop interconnect driver is designed to perform considerably better in terms of resource usage compared to the scenario where we only use direct point-to-point ICD links. In particular, we save memory, because the multi-hop driver has a comparably small memory footprint.

3.2 Design overview

Messaging in Barrelfish is connection-oriented: messages are passed via an explicit binding object, which encapsulates one half of a connection, and such a binding must be established in advance. Therefore, we have decided to support only connection-oriented multi-hop messaging (for now). The multi-hop interconnect driver is designed in such a way that channel set-up is collapsed into the binding phase.

We use virtual circuit switching in order to multiplex multiple multi-hop channels over the available ICD links. Virtual circuit switching has several advantages over a packed-switched approach. It ensures that all messages take the same path and thereby FIFO delivery of messages (as long as the underlying ICD links provide FIFO delivery). Moreover, it allows to create per-circuit state on the nodes of a virtual circuit.

Each monitor maintains a forwarding table. For each multi-hop channel, entries are created in the forwarding tables at all the nodes of that channel. Messages that are sent over the channel are forwarded at each node according to its forwarding table. Those entries in the forwarding tables can be seen as per-channel created *hard* state: It is explicitly created at channel set-up and deleted at channel tear-down. Additionally to the entries in the forwarding table, per-channel created state includes bindings to the neighbouring nodes on the multi-hop channel.

In addition to the forwarding table, each node maintains a routing table. The routing table is used for channel set-up: If a node receives a channel set-up request, it determines where to forward the request with the help of its routing table.

The virtual circuit switching approach would also allow to reserve some resources on the nodes for each channel. Per-channel reserved resources could include buffer space to save received, but not yet forwarded messages, or bandwidth on the ICD links. This is potentially very useful for congestion and flow control. Note that we cannot simply drop messages in case of congested links, as we want to provide a reliable messaging service. As of now, we do not reserve resources on the nodes, but allocate required resources dynamically.

3.3 Additional monitor bindings

A multi-hop channel is multiplexed over the available ICD links. However, for each multi-hop channel, there will be two additional ICD links: Two additional LMP channels will be created between the client's dispatcher and the monitor running on its core and between the service's dispatcher and the monitor on its core. LMP channels are rather cheap - they do not require polling and require only a small amount of memory. Therefore, this does not compromise our goal of optimizing resource usage. Figure 3.1 shows an example set-up of a multi-hop channel with the two additional LMP channels.

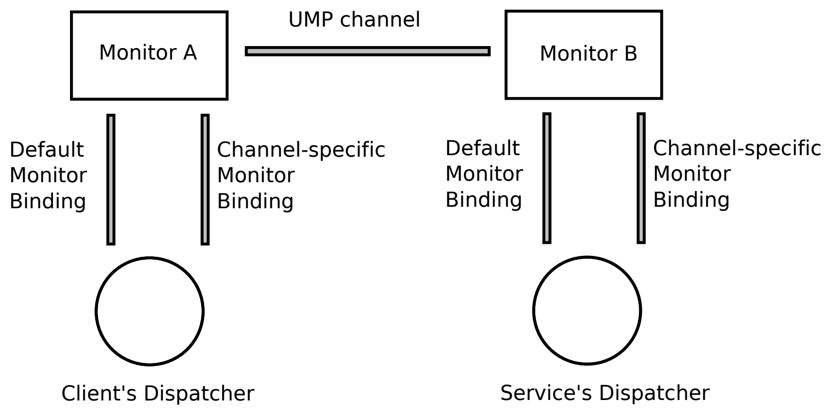

Figure 3.1: Basic set-up

Those additional channels are needed to ensure that the default monitor binding is not congested or even blocked by multi-hop messages. For example, suppose that a client's dispatcher receives a lot of multi-hop messages within a short period of time. The client reacts to this by allocating more memory. If multi-hop messages are sent over the default monitor binding, the message coming from the memory server will be blocked, therefore this will result in a dead lock. By creating new monitor bindings and not using the default monitor binding, we can prevent such a scenario.

3.4 Virtual circuit identifiers

Multi-hop messages carry a virtual circuit identifier (VCI). Virtual circuit identifiers allow nodes to identify the particular multi-hop channel a message belongs to. Each node on a multi-hop channel maintains a forwarding table, which maps VCIs to the next hop on that particular channel. A node forwards multi-hop messages based on this forwarding table. At channel end-points, a VCI allows to identify the binding belonging to the multi-hop channel the message was sent over. Virtual circuit identifiers are not only local to a specific link, but also to a direction on that link. Figure 3.2 shows an example assignment of VCIs.

We assign virtual circuit identifiers at random. At each node, we use a hash table to map virtual circuit identifiers to a pointer to the channel state. The use of a hash table allows efficient message forwarding. When a message arrives, it can be determined where to forward this message by means of a simple look-up in the hash table. The complexity of this lookup is linear in the number of virtual circuit identifiers that map to the same hash bucket (the number of buckets in the hash table is a compile time constant).

An attacker sending messages with manipulated virtual circuit identifiers may be able to send messages on channels not belonging to him. By assigning virtual circuit identifiers at random, we make it very unlikely for an attacker to find valid virtual circuit identifiers of channels not belonging to him.

This design requires that each node on a multi-hop channel tells its neighbours what virtual circuit identifier they should use for messages sent over that particular channel. This happens in the set-up phase of a multi-hop channel (see section 3.5).

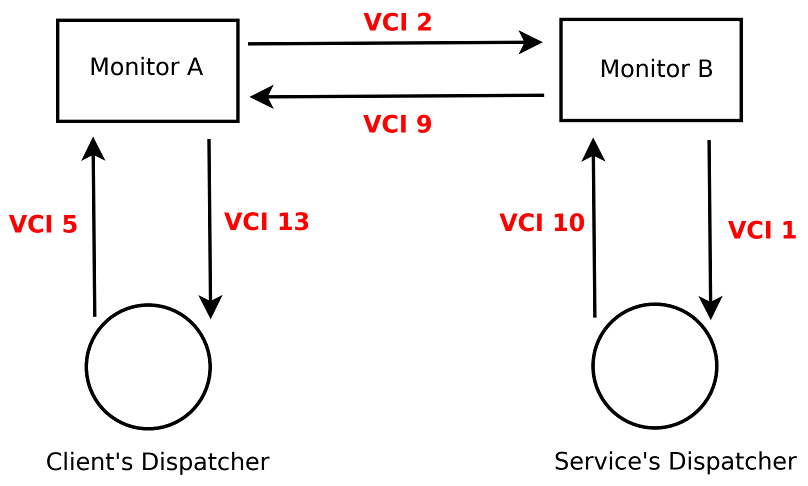

Figure 3.2: Virtual circuit identifiers

3.5 Channel set-up

If two dispatchers want to communicate with the help of the multi-hop interconnect driver, they have to create a multi-hop channel first. During channel-set up, one dispatcher must act as the client and the other as the server (however, once a channel is established, the communication process on both sides of the channel is indistinguishable).

The channel set-up process can be initiated by invoking the `multihop_chan_bind` function of the multihop interconnect driver. It has to be remarked that normally a user does not interact directly with the multi-hop interconnect driver, but only over the flounder generated stubs (see chapter 4).

multihop_chan.c

```
/**
 * \brief Initialize a new multihop channel
 *
 * \param mc  Storrage for the multihop channel state
 * \param cont  Continuation for bind completion/failure
 * \param iref  IREF of the service
 * \param waitset to use
 */
errval_t multihop_chan_bind(struct multihop_chan *mc, struct
    multihop_bind_continuation cont, iref_t iref, struct
    waitset *waitset)
```

The channel set-up process works as follows:

1. A client dispatcher initiates the set-up process by calling the bind function of the multi-hop interconnect driver. This function forwards the bind request to the monitor running on the client dispatcher's core. The bind request includes various parameters, including the *iref* of the service and the client's (ingoing) virtual circuit identifier.

2. The monitor running on the client dispatcher's core determines (from the iref) the core on which the service resides. It then forwards the bind request to another monitor, which is determined based on the routing table.

3. Monitors receiving the bind request check whether the service is running on the same core as they are. If so, they determine the local dispatcher which has exported this iref and forward the request to it. Otherwise, the bind request is forwarded to another monitor in the same way as in step two.

4. As soon as the service's dispatcher receives the bind request, it runs the user provided connection callback. Based on the return value of this callback, it either accepts the connection or rejects it. In any case, the bind reply is sent back to the monitor.

5. The monitor proxies the bind replay back to where it received the bind request from.

6. If the client dispatcher receives the bind reply, it will run the user provided bind callback.

In order to support setting up connections between dispatchers, the existing messaging interfaces between dispatchers and their local monitor, and between monitors has been extended. In particular, the following two messages have been added to the monitor and inter-monitor interface:

Interface monitor.if

```
call bind_multihop_service_request(uintptr service_id, uintptr
    sender_vci)

response bind_multihop_service_reply(uintptr sender_vci,
    uintptr receiver_vci, errval err)
```

Interface intermon.if

```
call bind_multihop_intermon_request(iref iref, vci_t
    sender_vci, uint8 core_id)

response bind_multihop_intermon_reply(vci_t receiver_vci,
    vci_t sender_vci, errval err)
```

As described in section 3.4, it is necessary that each node on the multi-hop channel tells its neighbouring nodes what virtual circuit identifier they should use for messages sent over that particular channel. Therefore, each message contains the virtual circuit identifier of the sender. The two response-messages additionally contain the VCI of the receiver. This allows the receiver of a response-message to identify the multi-hop channel the message belongs to.

3.6 Message forwarding

Once the multi-hop channel is set-up, messages can be sent in both directions. A message can be sent by invoking the `multihop_send_message` function of the interconnect driver.

multihop_chan.c

```
/**
 * \brief Send a multi-hop message
 *
 * \param mc pointer to the multi-hop channel
 * \param _continuation callback to be executed after the
   message is sent
 * \param msg pointer to the message payload
 * \param msglen length of the message payload (in bytes)
 *
 */
errval_t multihop_send_message(struct multihop_chan *mc,
    struct event_closure _continuation, uint8_t *msg, size_t
    msglen)
```

This function requires that the message payload is passed as one (char) array. If a user-defined message contains multiple arguments that are not stored in continuous memory locations, either the user-defined message must be split up in multiple multi-hop messages, or a new array must be allocated and all message arguments must be copied into the newly allocated array (see chapter 4 for a discussion).

In order to support sending messages, the existing messaging interfaces between dispatchers and their local monitor, and between monitors has been extended. Each multi-hop message contains a VCI, a field encoding the direction of the message and the message payload (as a dynamic array). Furthermore, it contains one field encoding message flags and another field used to acknowledge received messages. Those two fields are used for flow control (see section 3.10).

Interface monitor.if / intermon.if

```
message multihop_message(vci_t vci, uint8 direction, uint8
    flags, uint32 ack, uint8 payload[buflen]);
```

As a multi-hop channel allows to send messages in two directions, the direction field is needed to identify the direction of a particular message. Currently we assign direction ”1” to all messages going from the dispatcher who initiated the multi-hop channel to the other dispatcher, and direction ”2” for messages going the opposite way.

This definition of a multi-hop is motivated by the fact that it must be possible to transport an arbitrary message within one (or more) multi-hop messages. By using a dynamic array argument for the message payload, we can transfer data of an arbitrary size in a multi-hop message.

Internally, multi-hop messages are forwarded at every node of a multi-hop channel until they reach the receiver. We make sure that multi-hop messages cannot overtake other multi-hop messages at the nodes by enqueuing messages in the order they arrive and forwarding them in a FIFO order.

3.7 Capability forwarding

Because capabilities are maintained as references to per-core state in the CPU drivers, only the LMP interconnect driver which traverses kernel-mode code can directly deliver a capability along with message payload. In the multi-hop interconnect driver, capabilities travel out-of-band from other message payload.

multihop_chan.c

```
/**
 * \brief Send a capability over the multi-hop channel
 *
 * \param mc pointer to the multi-hop channel
 * \param _continuation callback to be executed after the
    message is sent
 * \param cap_state pointer to the cap state of the channel
 * \param cap the capability to send
 */
errval_t multihop_send_capability(struct multihop_chan *mc,
    struct event_closure _continuation, struct
    flounder_cap_state *cap_state, struct capref cap)
```

To send a capability, the monitor sends a `multihop_cap_send` message to its local monitor, containing the capability. The monitor determines whether the capability can be sent to the remote dispatcher. In gereral, capabilities referring to inherently local state (such as LMP endpoint) may not be sent, nor may capabilities that are currently being revoked. If the capability cannot be sent, a `multihop_cap_reply` message is sent back to the local dispatcher containing the error code. Otherwise, the capability is serialised and forwarded along the multi-hop channel.

The monitor running on the receiver's core reconstructs the capability from its serialised representation and forwards it to the destination dispatcher. This dispatcher identifies the binding to which the capability belongs and invokes a callback on that binding.

The capability sender only receives a reply message in case an error occurs. An error can occur if for example the capability cannot be sent or the receiver has no space left to accommodate an additional capability.

To support capability forwarding, the following two messages have been added to the monitor and inter-monitor interface:

Interface monitor.if

```
call multihop_cap_send(uintptr vci, uint8 direction, cap cap,
    uint32 capid);

response multihop_cap_reply(uintptr vci, uint8 direction,
    uint32 capid, errval err);
```

Interface intermon.if

```
call multihop_cap_send(vci_t vci, uint8 direction, uint32
    capid, caprep cap, bool null_cap);

response multihop_cap_reply(vci_t vci, uint8 direction, uint32
    capid, errval err);
```

3.8 Receiving messages

In order to receive messages sent over a multi-hop channel, message handlers must be registered with that multi-hop channel. In particular, three message handlers must be registered: one message handler for ”normal” messages, one handler for incoming capabilities and one handler for capability reply messages (that are sent in case an error occurred while sending a capability).

The flounder generated stubs for the multi-hop interconnect driver (see chapter 4) register those message handlers, not the application itself (normally).

multihop_chan.h

```
/**
 * \brief Set the message receive handler
 */
inline static void multihop_chan_set_receive_handler(struct
    multihop_chan *mc,
                   struct multihop_receive_handler rx_handler)
```

multihop_chan.h

```
/**
 * \brief Set the caps receive handler & the caps reply
     receive handler
 */
inline static void multihop_chan_set_caps_receive_handlers(
                   struct multihop_chan *mc, struct
                       monitor_cap_handlers cap_handlers)
```

3.9 Routing tables

The routing tables are used to determine where to forward a connection set-up request. Each monitor needs its own routing table. We currently support the automatic generation of routing tables for three basic modes of routing:

1. **Direct**: All set-up requests are immediately forwarded to the end-receiver.

2. **Ring**: We route over all cores of a system. Core i forwards a request to core $i + 1$ mod num_cores.

3. **Fat tree**: We route directly between the cores that are located on the same CPU socket. On each socket, we choose a ”leader” and route directly between all leaders. A set-up request for a core on a different socket is always forwarded over the local leader to the leader on that socket.

For the routing modes ”ring” and ”fat tree” we need information from the system knowledge base: We need to know the number of cores in the system for the ”ring” routing mode. For the ”fat tree” mode, we additionally need to know the number of cores per CPU socket (note that we assume here that sockets are continuously numbered).

We decided that there should be no direct communication between the monitor and the system knowledge base, because it is not always present. For some architectures, such as Microsoft’s experimental Beehive architecture or to a certain extend the Intel Single Chip Cloud Computer, the system knowledge base is not available. Therefore, a dependency of the monitor on the system knowledge base should be avoided.

For this reason, we decided to create a separate module, called the *routing table set-up dispatcher* (RTS) that talks to the system knowledge base and to the initial monitor (the monitor that is first booted). The routing table set-up dispatcher will retrieve the required information from the system knowledge base in order to construct the routing table. Once it has constructed the routing table, it will send it to the initial monitor.

The initial monitor will forward the (relevant parts of the) routing table to the other monitors once they are booted. This is necessary because we want to avoid having to create a channel between each monitor and the routing table set-up dispatcher.

It must be noted that the routing table set-up dispatcher can only generate the routing tables for the cores of a single system. It cannot handle set-ups like an Intel single chip cloud computer connected to a x86 machine over a PCIe-based channel.

3.10 Flow control

It is possible that one dispatcher on a multi-hop channel is sending at a faster rate than the receiving dispatcher can handle incoming messages and process them. Because we want to provide a reliable messaging service, we cannot just drop messages in such a case, but have to buffer them and deliver them eventually. To limit the space needed to buffer undelivered messages, we decided to implement a flow control mechanism within the multi-hop interconnect driver. The flow control mechanism allows the receiving dispatcher to control the transmission speed, so that it is not overwhelmed with messages.

We decided to use a credit-based flow control mechanism: The number of messages in flight at any given time is limited. Once a sender has reached this limit, he has to wait until he receives an acknowledgement that the receiver has processed previously sent messages. We call this limit the *window size*.

The flow control mechanism is completely transparent to applications. It is entirely handled by the multi-hop interconnect driver. On each message sent by a dispatcher over a multi-hop channel an acknowledgement for all messages previously received over this channel is piggy-backed.

If an application uses a one-way communication schema, i.e. one dispatcher is always sending while the other is only receiving, it is not possible to piggy-back acknowledgements on messages sent the other way. In such a case, the multi-hop interconnect driver sends a dummy message. A dummy message contains no message payload, but acknowledges all received messages. This approach ensures that acknowledgements are, whenever possible, piggy-backed on messages. Only if it is absolutely necessary, an acknowledgement is sent in its own message.

Chapter 4

Flounder integration

Flounder is a stub compiler which generates stubs for defined interfaces. To support multi-hop messaging, we created a new back-end code generator for the flounder stub compiler that generates code to use the multi-hop interconnect driver. Applications do not interact with the multi-hop interconnect driver directly, but only over the generated stubs. The stubs for the multi-hop interconnect driver have the exact same interface as stubs for other interconnect drivers. This makes application code independent of the interconnect driver used for communication.

The generated stubs can be seen as an ”adaptor” to the multi-hop interconnect driver. They translate calls to the common flounder interface to the interface of the multi-hop interconnect driver. Supported functionality mainly includes binding, sending and receiving of multi-hop messages and some control operations.

4.1 Binding

If two dispatchers want to communicate with the help of the multi-hop interconnect driver, they must acquire binding objects for each endpoint of the channel. In any binding attempt, one dispatcher must act as the client and the other as the service (however, once a binding is established, the communication process on both sides of the binding is indistinguishable). The binding phase is merged with channel set-up, i.e. a new multi-hop channel will be created during the binding process.

In order to initiate a binding, a client dispatcher calls the bind function for a given interface. Because Barrelfish features multiple interconnect drivers, the interface's bind function will have to decide which interconnect driver to use in order to establish the binding. Currently, it ”asks” the different interconnect drivers to establish a binding in a predefined order (for example, the LMP driver is always first). As soon as an interconnect driver manages to establish the binding, the binding process is finished. Should one interconnect driver fail, the next one in order is tried.

If an application wants to create a new multi-hop channel, it can pass the flag `IDC_BIND_FLAG_MULTIHOP` as an argument to the interface's bind function. This changes the order of the interconnect drivers: The multi-hop interconnect driver will come in second place, directly after the LMP driver. The LMP driver is first, because it is preferable to the multi-hop interconnect driver if client and service are running on the same core. If the multi-hop interconnect driver fails to establish a binding for some reason, the binding process continues as normal with the other interconnect drivers.

The result of the binding process on the client's and service's side is a binding object which is the abstract interface to the multi-hop interconnect driver for a specific interface type.

4.2 Sending messages

A message may be sent on the binding by calling the appropriate transmit function. We distinguish between user-defined messages and multi-hop messages. User-defined messages are those messages defined by the user in the interface. Multi-hop messages are messages that are sent over a multi-hop channel.

As pointed out in section 3.6, the multi-hop interconnect driver requires that the message payload is passed as one char array. If a user-defined message contains dynamic arguments (arguments whose size is only known at run-time), such as a string or a dynamic array, it is generally not possible to pass the message payload as one char array to the multi-hop interconnect driver. There are three possible approaches to send such a message:

1. Allocate a region of memory capable of holding all message arguments and copy the message arguments to this region. A pointer to it can then be passed to the multi-hop interconnect driver as message payload.

2. Split a user-defined message into multiple multi-hop messages. Each argument of the multi-hop message is transported in its own multi-hop message.

3. Use a combination of the above approaches. For instance, all fixed size arguments could be sent in one message, and each dynamically sized argument could be sent in an extra multi-hop message.

In comparison to approach 1, approach 2 saves the cost of allocating a region of memory and copying all the arguments of the message to that region. In exchange for that, it needs to split a user-defined message and transport it via multiple multi-hop messages. The preferable approach depends on the type of messages that are sent. However, we think that the performance penalty involved in sending each message argument in its own multi-hop message is not acceptable for most message types. Therefore, the flounder-generated stubs for the multi-hop interconnect driver use approach 1. Approach 3 might be a possible performance optimization, but is currently not in use.

4.2.1 Implementation

All message arguments are copied to continuous memory locations in order to send the whole user-defined message in one multi-hop message. When sending a user-defined message, we first calculate the size of its payload. The size of a message's payload is only known at compile-time if the message definition does not contain any dynamic arguments. Otherwise, the size of the payload has to be computed each time such a message is sent. After having computed the payload size, we allocate a memory region of that size and copy the message arguments to that region of memory. Finally, we pass a pointer to this memory region to the multi-hop interconnect driver.

We include the size of every dynamically sized argument in the message payload. This tells the receiver about the size of those arguments and allows him to retrieve them from the received message payload. Currently, we use 8 bytes to transfer the size of a dynamic argument. This ensures that we do not get an overflow. We account for those size fields when calculating the size of the message payload.

Capabilities are never sent as message payload. They are always sent out-of-band from "normal" message payload. A discussion of this can be found in section 3.7.

There is one issue regarding communication in heterogeneous systems of our implementation: To be consistent with the common flounder interface, we have to use a variable of type `size_t` to represent the size of a dynamic array. The type `size_t` is architecture dependent. On a 32-bit system it will likely be at least 32-bits wide. On a 64-bit system it will likely be at least 64-bit wide. If a dispatcher on a 64-bit system communicates with a dispatcher on a 32-bit system, this can lead to a problem: The dispatcher on the 64-bit system can potentially send dynamic arrays that are bigger than the dispatcher on the 32-bit system can receive. This is a problem of the current Barrelfish version and remains unsolved.

4.2.2 Send continuation

Each transmit function takes as an argument a pointer to a continuation closure. The closure will be executed after the message has successfully been sent. If another transmit function is called on the same binding before the continuation is executed, it will return the `FLOUNDER_ERR_TX_BUSY` error code, indicating that the binding is currently unable to accept another message. In this case, the user must arrange to retry the send.

The send continuation is the only way to know when a message has been sent over the multi-hop channel and it is safe to send the next message. Note that even if an application uses a ping pong communication scheme, i.e. it sends a message back and forth between two dispatchers, it is not guaranteed to not get a `FLOUNDER_ERR_TX_BUSY` error code, unless it serialises all sends with the continuation. This somewhat unintentional behaviour is caused by the fact that the multi-hop channel internally relies on other ICD-links to transport messages. The multi-hop channel itself uses send continuations on the underlying ICD-links to determine when it can accept another message. Those send continuations are always executed after a message is sent. Therefore it is possible (although unlikely) that a message is sent and the reply for that message is received, before the multi-hop channel can accept the next message.

4.3 Receiving messages

The flounder-generated stubs register a callback function with the multi-hop interconnect driver at channel set-up time in order to be notified when a message arrives. As we send a user-defined message within a single multi-hop message, we therefore also receive a user-defined message in one multi-hop message.

Upon receiving a multi-hop message, we have to extract the original user-defined message from it and pass it on to the user-provided receive handler. It is a fatal error if a message arrives on a multi-hop channel and the receive handler function for that message type is not set.

If the user-defined message contains dynamic arguments, we have to allocate space for each of those arguments separately and copy them from the received multi-hop message. This is necessary, because all dynamic message arguments are passed by reference to the user and become property of the user. The user must be able to free those arguments separately, therefore they must be copied to separately allocated memory. Fixed-size arguments are simply passed on the stack to the user.

Chapter 5

Performance Evaluation

In this chapter, we compare the performance of ”routed” communication over the multi-hop channel with the performance of regular, link-only interconnect drivers. In particular, we measure the message latency.

5.1 Test platform

We use the following system as test platform: The 2x2-core AMD system has a Tyan Thunder n6650W board with 2 dual-core 2.8GHz AMD Opteron 2220 processors, each of them with a local memory controller and connected by two HyperTransport links. Each core has its own 1MB L2 cache. All reported performance figures refer to this system.

5.2 Message latency

We measure the round-trip time (RTT) of messages between two cores, transported over different ICD-links. The benchmark application consists of a client and a server dispatcher. All the client does is to echo messages received from the server. The server sends messages to the client and measures how many clock cycles it takes until it receives the reply from the client. Before sending the next message, the server waits until the send continuation has fired at the server and client, indicating that the multi-hop channel is now ready to accept the next message. The client lets the server know when his send continuation has fired by sending an extra control message over an additional binding to the server.

Because we expect that the round-trip time is dependent on the type of the transported message, we use different message types. In particular, we use the following message types for our measurements:

Message name	Description
empty	Message with no payload
payload32_1	Contains one 32-bit argument
payload32_4	Contains four 32-bit arguments
payload32_8	Contains eight 32-bit arguments
payload64_1	Contains one 64-bit argument
payload64_4	Contains four 64-bit arguments
payload64_8	Contains eight 64-bit arguments
buffer1	Contains an dynamic array with a size of 1 byte
buffer100	Contains an dynamic array with a size of 100 bytes
buffer1000	Contains an dynamic array with a size of 1000 bytes

We have measured the round-trip time of those messages over LMP, UMP and the multi-hop interconnect driver. In case of LMP, both server and client were executed at the same core. In case of UMP and the multi-hop interconnect driver, the server was executed at core 0 and the client at cores 1, 2 and 3, respectively. All multi-hop channels lead through two hops, i.e. the two monitors running on the cores of the server and client.

In total, we have conducted each measurement 1000 times and discarded the first 100 results, therefore taking into account 900 measurements of round-trip times. The following table shows the median of the measured times, as well as the standard deviation:

		LMP	UMP	Multi-hop
Message type	Core	Cycles (σ)	Cycles (σ)	Cycles (σ)
	1		1618 (69)	22004 (1148)
empty	2	3325 (103)	1747 (67)	22091 (862)
	3		1745 (70)	22807 (1411)
	1		1566 (56)	22005 (1037)
payload32_1	2	3765 (21)	1692 (53)	22204 (829)
	3		1693 (54)	22938 (1371)
	1		1666 (54)	23597 (1496)
payload32_4	2	3616 (13)	1801 (55)	23901 (830)
	3		1796 (57)	24214 (970)
	1		1609 (60)	23690 (786)
payload32_8	2	3983 (16)	1738 (64)	24715 (785)
	3		1741 (66)	24608 (1039)

Message type	Core	LMP Cycles (σ)	UMP Cycles (σ)	Multi-hop Cycles (σ)
payload64_1	1	3549 (12)	1557 (58)	22518 (1125)
	2		1685 (57)	22754 (890)
	3		1691 (59)	24132 (1410)
payload64_4	1	3772 (12)	1587 (53)	24031 (785)
	2		1737 (72)	24534 (804)
	3		1732 (66)	24631 (958)
payload64_8	1	4085 (20)	2005 (77)	26597 (696)
	2		2253 (79)	27289 (700)
	3		2253 (77)	27150 (710)
buffer (1)	1	5865 (98)	2076 (78)	23564 (818)
	2		2244 (100)	23835 (825)
	3		2244 (98)	24134 (983)
buffer (100)	1	10607 (126)	23911 (242)	37755 (1014)
	2		23932 (2118)	38272 (1033)
	3		23973 (677)	38377 (949)
buffer (1000)	1	386475 (392)	41321 (259)	1437771 (38392)
	2		41600 (903)	1438911 (38631)
	3		42493 (628)	1479689 (75269)

The measured latencies are depicted in figure 5.1 on a logarithmic scale.

5.2.1 Discussion

We notice that routed communication over the multi-hop interconnect driver performs considerably worse than communication over direct ICD-links. In our test set-up, the multi-hop channel leads through two hops, i.e. the two monitors running on the core of the client and server. Each multi-hop message is transported twice over an LMP channel and once over an UMP channel. Hence, the best-case round-trip time equals the sum of twice the round-trip time of the LMP channel and once the round-trip time of the UMP channel.

The best-case performance is not achieved in practice. One reason for that is the fact that for each multi-hop message that is sent over the multi-hop channel, we need two context switches: After the application has sent the message, there is a context switch to the monitor (on the sender's core). The monitor forwards the message to the monitor running on the receiver's core. There, we need an additional context switch to the application. Figure 5.2

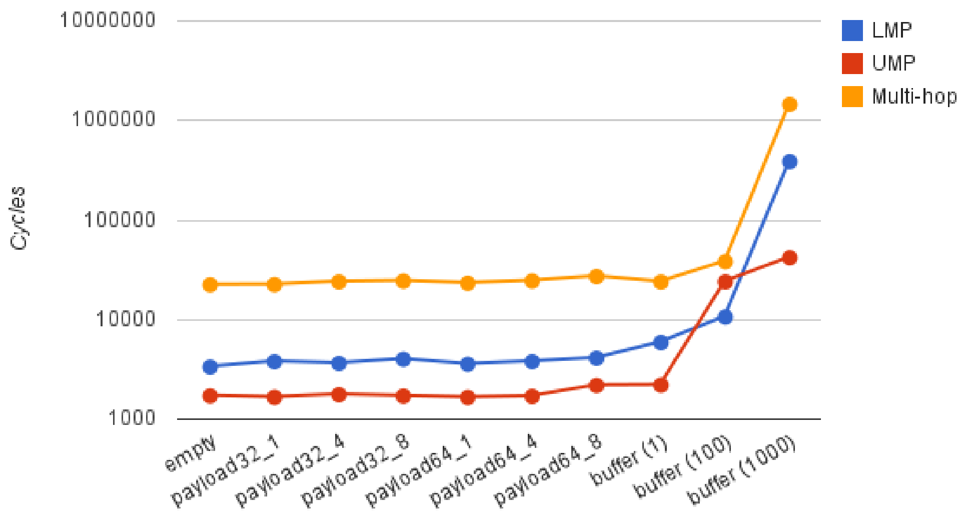

Figure 5.1: Median of measured round-trip times

shows a visual trace of one round-trip time measurement. The benchmark application is depicted in green, the monitor in blue.

In case of very large messages, the LMP interconnect driver fragments the message, resulting in even more context switches between the monitor and the application. This explains the big latency of the message containing 1000 bytes.

Another reason why the multi-hop interconnect driver performs worse than communication over direct ICD-links is the fact that both the LMP and UMP interconnect driver perform worse when transporting a dynamic array compared to transporting the same amount of data in fixed-size arguments. For example, consider the message "payload64_8" (containing only fixed size arguments) and the message "buffer (100)" (containing one dynamic array). Although the message "payload64_8" contains more data, its round-trip time is better than the round-trip time of "buffer (100)" by a factor of ten over the UMP interconnect driver. The multi-hop interconnect driver transports every message over underlying ICD-links in the form of a dynamic array (see section 3.6). Therefore the performance of the UMP and LMP interconnect drivers when transporting dynamic arrays is crucial to the performance of the multi-hop interconnect driver.

Finally, we have to take into consideration that the multi-hop interconnect driver copies every message argument before the message is sent and after

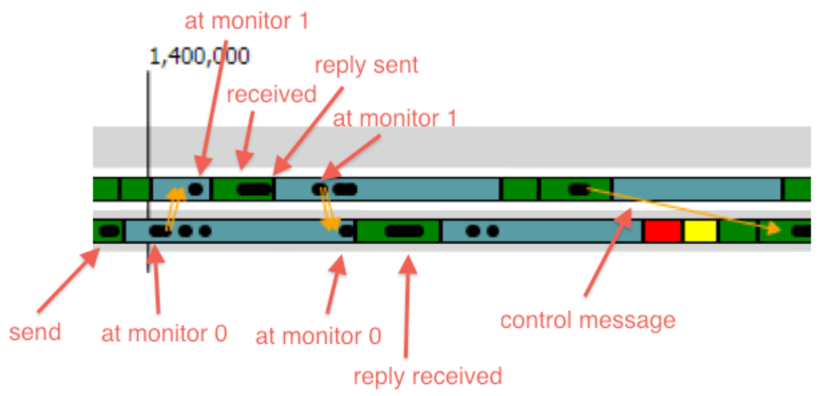

Figure 5.2: Visual trace of the latency benchmark

the message is received (a discussion of this can be found in section 4.2). Depending on the message size, this has a significant impact on performance.

We noticed during the performance measurement that scheduling has an impact on our measurements. Depending whether the monitor on core 0 is still running when the reply arrives, the reply can be forwarded immediately or has two wait until the monitor is scheduled again and therefore resulting in worse latency. Figure 5.3 shows a visual trace of a "unfortunate" scheduling situation.

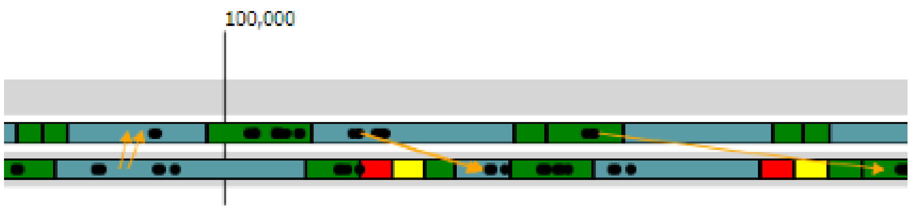

Figure 5.3: Unfortunate scheduling

To deal with this, we have modified the default polling cycles for the monitor. The default polling cycles determine how long the monitor polls its message channels before it yields control if no message has arrived. For the above measurements, we set this number to 20000.

5.2.2 Number of hops

The number of hops on a multi-hop channel has also an influence on the round-trip time. We measured the round-trip time of the multi-hop interconnect driver over two, three and four hops. The following table shows the median of the measured round-trip times, as well as the standard deviation:

	2-hop	3-hops	4-hops
Message type	Cycles (σ)	Cycles (σ)	Cycles (σ)
empty	22004 (1148)	25185 (563)	27871 (632)
payload32_1	22005 (1037)	25191 (686)	28017 (743)
payload32_4	23597 (1496)	26627 (844)	29705 (1018)
payload32_8	23690 (786)	27357 (759)	30700 (1613)
payload64_1	22518 (1125)	25774 (649)	28693 (722)
payload64_4	24031 (785)	27649 (763)	30794 (890)
payload64_8	26597 (696)	31117 (743)	35739 (774)
buffer (1)	23564 (818)	26990 (769)	30119 (861)
buffer (100)	37755 (1014)	62867 (278)	62651 (436)
buffer (1000)	1437771 (38392)	1501359 (39550)	1530413 (38304)

The measured latencies are depicted in figure 5.4 on a logarithmic scale. We notice that the number of hops on the multi-hop channel has a negligible impact on the performance of the multi-hop interconnect driver in our benchmark. We therefore conclude that the round-trip time is dominated by context switches and a transport over an LMP channel that occur only at the sender and receiver.

Naturally, the performance penalty involved in forwarding messages over additional hops depends on what is running on the cores of those additional hops. In our case, only the monitors were running on those cores, therefore messages were forwarded immediately. Figure 5.5 shows a visual trace of our benchmark when forwarding messages over four hops.

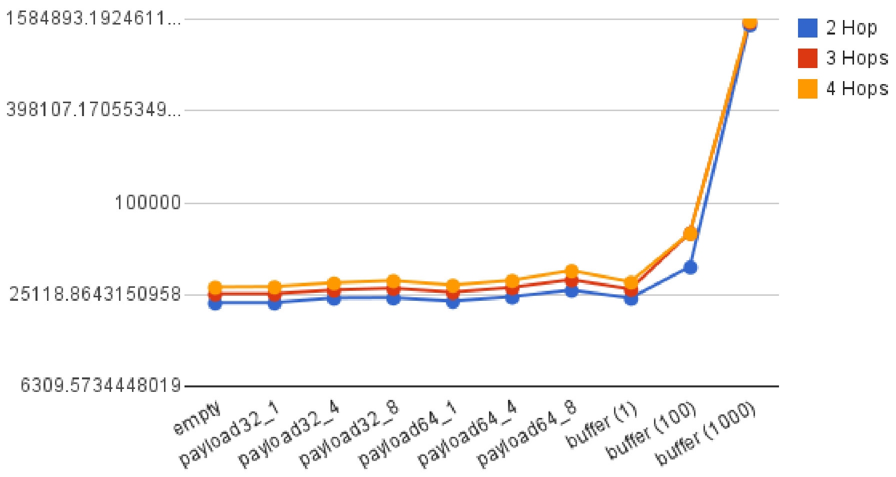

Figure 5.4: Median of measured round-trip times

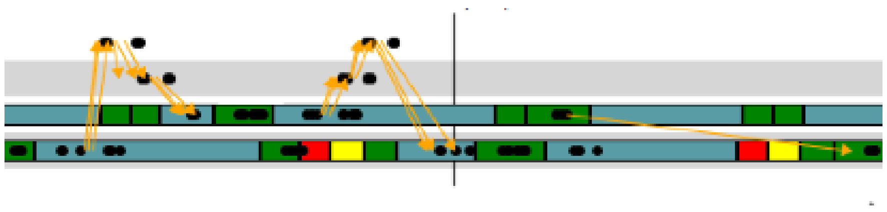

Figure 5.5: Multi-hop messaging over four hops

Chapter 6

Future Work

In this chapter, we will present some directions for future work.

6.1 Error handling

Currently, the multi-hop interconnect driver assumes that no node on a multi-hop channel will ever fail and that none of the underlying ICD-links will fail. This assumption is not quite true - for instance, an Ethernet-based ICD-link fails if the network cable is unplugged. The multi-hop interconnect driver should be extended to handle cases of partial failure by either trying to re-establish the multi-hop channel over a different route, or notifying clients about the disconnect.

6.2 Generation of routing tables

We support the automatic generation of routing tables currently only in three basic modes (see section 3.9). It is desirable to make the routing table set-up dispatcher more sophisticated. For instance, routing tables could reflect the congestion of links, therefore allowing us the create new multi-hop channels over less congested links.

Furthermore, we should support the automatic generation of routing tables in set-ups like an Intel single chip cloud computer connected to a x86 machine over a PCIe-based channel. The routing table set-up dispatcher currently can only generate routing tables for the cores of a single machine.

6.3 Group communication

Various parallel computing abstractions such as barriers require communication among a group of threads. When any thread enters a barrier, it waits for all other threads to enter the barrier as well before continuing. Various distributed communication abstractions such as achieving consensus also require communication among a group of nodes. A group of nodes wanting to come to agreement on some value need to communicate with each other.

Without multi-hop messaging, group communication is very expensive: It requires a fully connected network between the members of a group. Multi-hop messages makes group communication more efficient and better performing. Therefore, the multi-hop interconnect driver should be extended in order to allow group communication.

Bibliography

[1] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs, "Embracing diversity in the Barrelfish manycore operating system." in *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

[2] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs, "Your computer is already a distributed system. Why isn't your OS?" in *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verità, Switzerland, May 2009.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems" in *Proceedings of the 22nd ACM Symposium on OS Principles* , Big Sky, MT, USA, October 2009.

[4] Simon Peter, Adrian Schüpbach, Dominik Menzi, Timothy Roscoe, "Early experience with the Barrelfish OS and the Single-Chip Cloud Computer" in *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium (MARC)*, Ettlingen, Germany, July 2011.