ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

NATIONAL ICT AUSTRALIA

Andri Toggenburger

Modular Network Router for Microkernel-Based Embedded Systems

Master Thesis
November 7, 2006 to May 6, 2007

NICTA's Embedded, Real-Time, and Operating Systems Research Program, Sydney
Supervisor at NICTA Sydney: Dr. Ihor Kuz
Supervisor at ETH Zurich: Prof. Roger Wattenhofer

Abstract

Embedded systems are now ubiquitous and can be found in an abundance of application areas ranging from mobile phones to cars. To increase the reliability and thrustworthiness of these increasingly complex embedded systems, microkernel-based operating systems and component-based software engineering techniques are being used. CAmkES, a project of NICTA's ERTOS research program, has recently developed a component architecture for the microkernel-based operating system L4.

This report describes the design and implementation of a modular router for the L4 microkernel based on CAmkES. On the one hand, this included the proposition of a low overhead yet extensible and flexible CAmkES-based router architecture. On the other hand, patterns of how to apply CAmkES and its novel features to the development of embedded systems were researched.

The resulting modular router's performance depends on the configured level of protection between its components. The router induced delay as compared to a similar monolithic system is less than 5% if no protection between the different modules is desired. However, the performance penalty to pay for full protection between all modules is significantly higher.

Contents

Chapter 1

Introduction

1.1 Motivation

Embedded systems are now ubiquitous and can be found in an abundance of application areas ranging from mobile phones to cars. To increase the reliability and thrustworthiness of these increasingly complex embedded systems, microkernel-based operating systems and component-based software engineering techniques are being used. CAmkES, a project of NICTA's ERTOS research program, has recently developed a component architecture for the microkernel-based operating system L4. It allows the application of component-based software engineering to the development of embedded systems software.

Routers are embedded systems dedicated to the task of forwarding network packets as fast as possible from one network interface to another. A modular router architecture has the advantage of being extensible and flexible by providing support for the addition of new modules or the alteration of existing configurations. Furthermore, the robustness of modular routers benefits from the re-use of well-tested components reducing the amount of new, insufficiently tested code.

1.2 Task

The work presented in this report was conducted as part of a master's thesis. The primary goal of this master's thesis was the design and implementation of a modular router based on the CAmkES component architecture for embedded systems. On the one hand, this involved the development of network drivers, network stacks and any other required (operating system) functionality as reusable components. On the other hand, different approaches to building the modularized router were researched. This included investigating patterns of how to assemble the different components, experimenting with different types of connections between them and evaluating the resulting router by executing performance measurements on an embedded hardware platform.

As the described router is the first significant system based on the CAmkES architecture, another part of the research comprised of analysing the architecture's features and properties and proposing and implementing changes or extensions to it.

1.3 Overview

This report is organized in the following way: This chapter provides information about the background of this thesis project. This includes the different other projects that this project is based on, and the embedded hardware platform used to run the developed router. The design and the implementation of the modularized router for embedded systems is presented in Chapter 2 including a discussion of the designed architecture and how the resulting router relates to

other modular router projects. Chapter 3 analyzes and discusses the results of a variety of benchmarks carried out using the implemented router. As CAmkES forms an important part of the presented project, Chapter 4 discusses the suitability of CAmkES for a project like this and proposes additional features. Furthermore, features designed and implemented in CAmkES as part of this project are also explained. Finally, conclusions including feature work can be found in Chapter 5.

1.4 Terminology

- `Microkernel`: A microkernel is a minimal computer operating system kernel providing only basic operating system services (system calls), while other services (commonly provided by kernels) are provided by user-space programs called servers.

- `Embedded System`: An embedded system is a special-purpose system in which the computer is completely encapsulated by or dedicated to the device or system it controls. Unlike a general-purpose computer, such as a personal computer, an embedded system performs one or a few pre-defined tasks, usually with very specific requirements.

- `NAT`: Network address translation as defined in RFC 3022[31]. Its features include IP address translation, TCP/UDP port translation and ICMP query ID translation. Another feature is the translation of TCP/IP and UDP/IP packets embedded in ICMP error messages. These features enable multiple machines on the local network to access the global network at the same time by transparently sharing one global IP address. Consequently, the topology of the local network is hidden to the global network. Furthermore, the NAT component acts as a firewall by preventing all network traffic that originated on the global network, and was not requested from a local machine, from entering the local network.

- `NAT Router`: This expression refers to a system performing network address translation as defined in RFC 3022[31].

- `Access Router`: This expression refers to a NAT Router which is connected to the Internet through an ADSL or Cable connection. Access routers are embedded systems as they are completely dedicated to providing broadband access to the internet. Their field of application is home use.

- `IPC`: Inter-Process Communication is a set of techniques for the exchange of data between two or more processes. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls.

- `NIC`: Network Interface Controller.

- `RX queue`: Queue where received network packets are queued.

- `TX queue`: Queue where network packets ready for transmission are queued.

- Symbols used in the component and interface figures (see Figure 1.1).

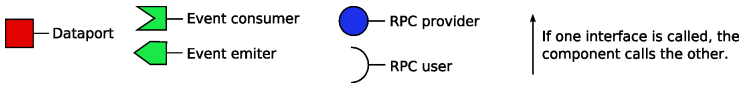

Figure 1.1: Symbols used in the component and interface figures: The captions refer to the description of the 3 CAmkES interface types in 1.7.3.

- DMA: Direct Memory Access. A dedicated controller supports the CPU by independently transferring data from the hardware over the bus to the main memory. The CPU is therefore available for other tasks at the same time. After a DMA transfer is complete, the CPU can access the transferred data at a previously defined memory location.

- `PIO`: Programmed Input/Output. In PIO mode, the CPU itself has to transfer the data from the hardware over the bus. This means, the CPU is not available for other tasks when it executes PIO transfers. The CPU becomes a bottleneck in a system using PIO especially if the hardware is slow and the CPU therefore has to waste cycles.

- `ADL`: Architecture Description Language: Used to describe a software architecture as a set of connected components in CAmkES.

1.5 L4

L4[23] is a family of second-generation microkernels, which is characterized by high performance, more than an order of magnitude better than its predecessors.
L4 is characterized by:

- Small size.
- Low IPC overhead.
- A small number of abstractions and fundamental mechanisms, and almost complete absence of kernel-defined policies. Memory management, protection management and process management are the responsibility of user-level servers.
- Interrupt handlers and device drivers at user level.

As such, L4 is almost a true microkernel. It violates the strict definition of a microkernel by containing a scheduler (and some scheduling policy) and typically drivers for serial ports and timers.

L4-embedded is the L4 version that was used for this thesis project. It is a L4 kernel that is especially tailored to the needs of embedded systems. More information about how L4-embedded supports the design of embedded systems and L4-embedded internals can be found in [11] and on the L4-embedded project's homepage[5].

1.6 Iguana

Iguana[18] is designed as a base for the provision of operating system (OS) services for embedded systems. Iguana runs in user space on top of the L4 microkernel and complements the underlying L4 API. It provides services virtually every OS environment requires, such as memory and protection management, thread management and a device driver framework.
Special features include:

- The memory and cache footprints of Iguana are kept small.
- Single address space layout to ease the sharing of data and to support fast address space switching on the ARM architecture widely used in embedded systems.
- Per-process protection domains. Every access of data in another protection domain is subject to access control.
- Capability-based protection management.
- Enables OO-style use of the L4 primitives.

Figure 1.2 shows an L4/Iguana based system. The L4 kernel runs in privileged mode in the kernel space while the services provided by Iguana, device drivers and applications run in unprivileged mode in the user space. Different applications and operating system servers, their data and threads are encapsulated by protection domains (PD). All accesses between different protection domains are subject to access control. Threads of a particular protection domain can access the ressources of another protection domain only if they hold the required capabilities. Capabilities to access a certain ressource can be explicitely passed from one protection domain to the other in order to grant access rights. The different protection domains' threads communicate via different IPC mechanisms provided by L4/Iguana including message passing and shared memory areas.

1.7 CAmkES

The CAmkES project[20] provides a component architecture for microkernel-based embedded systems. The router application described in this report represents the first major application designed and implemented using CAmkES. On the one hand, the features and properties of

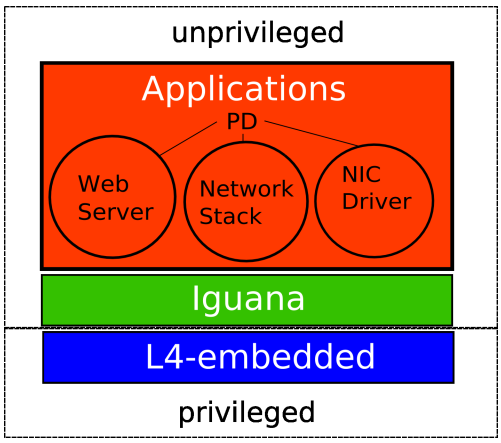

Figure 1.2: Iguana and applications running in unprivileged mode on top of the L4-embedded microkernel. Operating system components and applications are encapsulated by protection domains.

CAmkES shaped the resulting router in a major way. On the other hand, the insights gained during the router's development also led to the development of new features for CAmkES. This section describes the major features and properties of the CAmkES architecture. More information about the CAmkES design and how it relates to L4 and Iguana can be found in [21].

1.7.1 The CAmkES Project

The purpose of CAmkES is to provide support for developing embedded systems on top of microkernels. Since the underlying philosophy of microkernel-based operating systems is to componentised the OS by implementing its services as servers running at user level, it makes sense to apply component-based software engineering techniques to the design and development of these systems. The CAmkES approach to doing this involves providing a component architecture that supports the modeling of microkernel-based systems as collections of interconnected components. This architecture provides a component model, libraries of standard interfaces and component definitions, standard component implementations, standard services, and support for various architectural patterns and styles specifically suited for embedded systems.

1.7.2 Development of CAmkES-based Embedded Systems

As with general component-based software development, the development of CAmkES-based systems has four stages: design, implementation, deployment, and runtime. These are shown in Figure 1.3

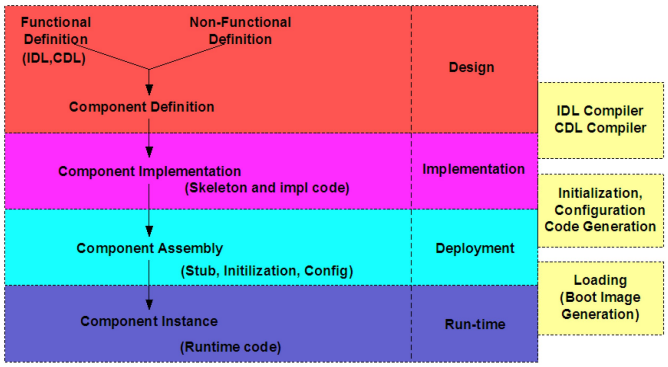

Figure 1.3: CAmkES Development Stages

At design time a CAmkES based system is defined using an interface definition language (IDL) and an architecture description language (ADL). The IDL is used to define interfaces through

which components communicate with one another. The ADL specifies the actual components, including which interfaces they provide and which interfaces they use. The ADL is also used to specify a complete component-based system, that is, the set of components in the system and the connections between the components. Finally, the ADL also provides configuration parameters to the defined components. Once all the components have been specified, a compiler generates header files and stub code from the IDL and ADL.

At the implementation stage the actual component code is written. The code makes use of the headers and stubs generated in the previous step.

At deployment time all of the code (both hand-crafted and generated) is compiled, linked and combined with the core runtime, the operating system and any non-CAmkES services to form a system boot image.

Finally, at run time, the boot image is loaded onto hardware (or a simulator), the component instances are created, connections between the components are initialised and the system is started running.

1.7.3 CAmkES Component Architecture

A component is a basic unit of encapsulated behavior. It is used to organize operations and data into interfaces that have well defined semantics and behaviors. During run-time, a CAmkES component maps to an Iguana protection domain. Thus components cannot access functionality provided by each other if not specified so at design time. Components expose interfaces that allow applications and other components to access their features. A component must also specify which external interfaces, that is, interfaces provided by other components, it will use.

CAmkES supports three types of interfaces:

- RPC interface: An RPC interface defines synchronous communication between components through remote procedure calls (RPC). A component can provide functionality as an RPC server or use functionality as an RPC client.
- Event interface: Used for asynchronous notifications between components. A component can emit (produce) an event or consume it.
- Dataport interface: Represents an interface that allows components to transfer data between each other (e.g. implemented as shared memory areas).

The CAmkES component model encapsulates communication between components in explicit architectural elements called connectors and connections. A connector defines the runtime interaction between a set of interfaces belonging to two or more components. A connection is an instance of a connector and connects the components together during runtime; it executes the functionality defined in the corresponding connector's implementation.

Connectors can be defined by the user and therefore the overhead of communication to be optimized since they can be tailored to a specific problem or scenario.

1.7.4 An Example Application in CAmkES

This section presents a simple example application that features two components based on the CAmkES component architecture. A server component implements the functionality to add two integer variables while a client component invokes the provided implementation.

During the design stage, the interface through which the components communicate is defined using the IDL:

```
interface Addition{
int add(in int x, in int y)
}
```

Furthermore, the actual components using this interface are declared in the ADL declaration part. Components that are active, which means that they have their own thread of control, are declared as control components:

```
import "Addition_interface.idl4";
component Server {
        provides Addition add;
}

component Client{
        control;
        uses Addition add;
}
```

In the ADL assembly parts, the instances of the components are defined. To assemble the whole system out of these instances, connections between the instances' interfaces are declared. A connection declaration consists of a connector to be used for this connection (e.g. `IguanaRPC`) and of the declaration of the two components' interfaces to be connected.

```
assembly{
        composition{
                component Server cx;
                component Client c1;
                connection IguanaRPC con1(from c1.add, to cx.add);
        }
}
```

At the implementation stage, the actual code for the server and the client components is written. This code makes use of the header and stub files that were generated out of the ADL and IDL specifications in the previous step.

The client component's code imports the generated header file <Client_add.h> that contains the functions declared by the corresponding IDL interface definition. The client's control thread can call these functions while the generated client stub code transparently communicates with the server providing the implementation.

```
#include <Client_add.h>
...
void run(void *args){
        int r = add_add(1,2);
}
```

On the server side, the server's code imports the header file <Server_add.h> that contains the function headers (as stated in the IDL declaration) to be implemented by the server.

```
#include <Server_add.h>
...
int add_add(int x, int y){
        return x + y;
}
```

At this stage, the definition and the implementation of the CAmkES bases application is complete. The code is compiled, linked and combined with the core runtime and operating system to form a system boot image.

At runtime, the system contains two components as shown in Figure 1.4. The client component calls the implementation of the interface `Addition` by using its generated client stub. The logic implemented by the `IguanaRPC` connector is executed when the client and the server component communicate with each other via the declared connection `con1`.

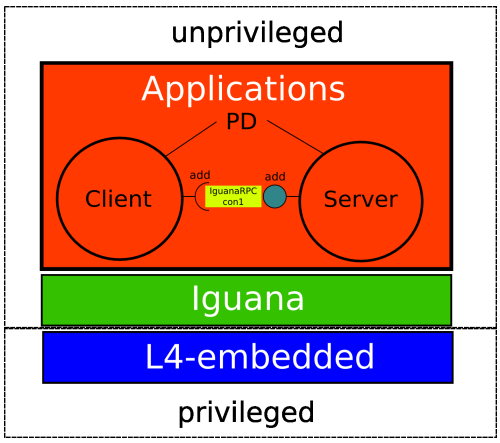

Figure 1.4: The defined application during run-time.

1.8 lwIP Network Stack

The lwIP Network Stack is a small independent implementation of the TCP/IP protocol suite that has been developed by Adam Dunkels at the Computer and Networks Architectures (CNA) lab at the Swedish Institute of Computer Science (SICS)[13]. The focus of the lwIP TCP/IP implementation is to reduce resource usage while still having a full scale IP stack. This makes lwIP highly suitable for use in embedded systems like the embedded router presented in this report.

1.9 Used Hardware

As a part of this project, an embedded hardware platform had to be chosen in order to run the router implementation. A Gumstix[3] machine was chosen due to the good experiences that several members of ERTOS program previously had made with it. As the goal of this project is to build a router, it is furthermore crucial that the chosen hardware can be extended with at least two NICs. A Gumstix extension board sporting two SMC91C111 ethernet controllers had come out shortly before the project commenced. Furthermore, a network driver for the SMC91C111 NIC had already been partially ported to L4/Iguana.

1.9.1 Gumstix Connex 400xm board

The `Gumstix Connex 400xm` board provides 64MiB of RAM and sports an Intel XScale/PXA255 processor that runs at clock rates of 200MHz, 300MHz or 400MHz. Intels XScale/PXA255 processor is compliant with the 5th generation of the ARM Architecture. The ARM Architecture is a 32-bit RISC processor architecture developed by ARM Limited[1] that is widely used in embedded systems due to its power saving features. The ARM architecture supports fast address space switching as implemented by L4/Iguana. It also provides a timer register that can be used to measure the execution time of critical program sections, such as e.g. interrupt handlers, to locate a bottleneck in a system.

A property that facilitates the development of an application for a Gumstix board is the possibility to upload boot images via Ethernet. This speeds up the process of deploying and testing software on this embedded platform. Furthermore, it is simple to connect a Gumstix machine to the serial port of a PC in order to print out debugging information.

1.9.2 SMC91C111 Network Interface Controller and Driver

Basic Gumstix machines can be extended by adding Gumstix extension boards. The Gumstix extension board netDUO-mmc contains two SMC91C111 network controllers.

The SMC91C111 network interface driver currently used in the router is based on a Linux implementation of a driver for NICs of the SMC91x family by Nicolas Pitre[25]. It was partially ported to L4/Iguana before this thesis commenced, however, features like controller initialization and DMA were still missing. There was also no support to run two network interface controllers at the same time in L4/Iguana.

Due to L4's microkernel architecture, drivers run in user-space like other servers. Interrupts are delivered to drivers as IPC messages (see [11]). A driver first must be registered with L4 in order to be notified if a certain interrupt was triggered. The hardware's I/O registers need to be mapped to the driver's memory area so that the driver is allowed to read the status registers of the device or write to the control registers.

Chapter 2

Design and Implementation of the Component Based Network Router

2.1 Router requirements

This section states the requirements for the component based network router. The router is targeted for embedded systems, so there are special requirements relating to the properties and challenges of these systems.

2.1.1 Functional requirements

The router's end-user functionality will be similar to the functionality of Access Routers (see 1.4). This means that the main function of the router will be to multiplex a single, global IP Address to several local IP Addresses. It can be used to share a modem connected to the internet between different machines.

The functional requirements are:

- Network address translation. The router will allow to share a single (global) IP Address transparently between multiple machines on the local network.
- Port forwarding, which allows a machine on the internet to connect to a specified port on a local machine by bypassing the dynamic network address translation.
- Administration interface to remotely configure router's settings. The interface will also allow querying of the router's status.
- DHCP client to automatically obtain IP Address, Default Gateway and Subnet Mask for the network interface, which is connected to the Internet.

2.1.2 Non-functional requirements

Security requirements:

- Complete protection between trusted and untrusted components by supporting memory protection between them. A trusted component could be e.g. a network stack component. An untrusted component could be e.g. a (downloaded) third party router extension component.

Flexibility requirements:

- Ability to upgrade parts of the router. This means new versions of single components can be installed without having to upgrade other components at the same time.
- Ability to add new functionality to the router by adding additional components to the system.

- Ability to reuse the existing router components in another application or configuration.

Performance requirements:

- The router induced delay has to be small enough so that the router can be used as a convenient access device to the internet. The targeted field of application is the same as the field of application of an Access Router.
- The router's throughput has to be high enough to support broadband internet connections. At the time this thesis was written, the fastest ADSL plan of the swiss internet provider Bluewin supported upload speeds of up to 0.5Mib/s and download speeds of up to 5Mib/s

Configurability requirements:

- The router's composition has to be configurable offline by editing its CAmkES ADL file.
- The router's NAT functionality has to be configurable online by using an Administration Console.

Compatibility requirements:

- The router has to be compatible with the standard internet protocols IP (RFC 791[28]), TCP (RFC 793[29]), UDP (RFC 768[30] and ICMP (RFC 792[27]).

Special requirements of Embedded Systems:

- The footprint of the router has to be small as embedded systems have limited memory.
- The functionality has to be implemented as efficiently as possible as embedded systems have limited processing power and power supplies.

2.1.3 Hardware requirements

- The router implementation will run on a Gumstix[3] board (`Gumstix Connex 400xm`: ARM XScale processor, 64MB RAM, 16MB Flash Memory, 400MHz).
- The network interface card used is a Gumstix `netDUO-MMC` card. It features two `SMC91C1111` Network Interface Controllers (10-100baseT) and two Ethernet ports.

2.1.4 Decisions leading to the stated requirements

Functional requirements

There was no purpose built router hardware available (including multiple NIC, fast bus/switching architecture and multiple processors). Given the available Gumstix machine, it was more appropriate to run NAT than a high performance routing algorithm. On the one hand, the Gumstix machine has only two network interfaces, which makes it impossible to reasonally test routing algorithms, which were designed for machines with multiple network interfaces. On the other hand, in a high performance router implementation, the bus of the Gumstix machine would pose a major bottleneck.
Furthermore, since the target hardware platforms for L4/Iguana and CAmkES are embedded systems and one kind of embedded system many people deal with on a daily basis is an Access Router, it was interesting to turn a (very small) Gumstix machine into such a device.

Hardware requirements

There are currently no L4/Iguana network drivers available for other hardware platforms than the chosen Gumstix machines. It would have taken a relatively long time to port network interface drivers of other hardware to L4/Iguana, which would have distracted from the actual focus of the thesis. Consequently, the stated Gumstix platform was chosen as a target system.
Furthermore, the Gumstix machine is very small, even smaller than most current Access Routers. This means that the router could be used in real life e.g., to replace an old desktop machine used as a NAT Router.

Non functional requirements

These requirements were chosen to cover as many of CAmkES' features and properties as possible. On the one hand, this leads to a router taking advantage of all the the mechanisms provided by CAmkES. On the other hand, it means that flaws and missing features in the CAmkES architecture will be discovered during design and implementation of the router.

2.2 The Router's Components

The router's main functionality is the forwarding, including NAT, of network packets either from the global NIC to the local NIC or vice versa. Other functionality includes providing an administration service for the NAT that is accessible through the network and a DHCP client. This divides the router into two planes. There is a forwarding plane that supports the fast forwarding of packets from one NIC to the other and provides a network stack to other components. The other plane is the application plane that interacts with the forwarding plane by querying and configuring the forwarding plane's components. Components of the application plane may also use the forwarding plane's network stack to consume network packets that are directed to the router itself and send out network packets on one of the router's NICs.

The router forwarding plane's components are based on the main OSI[33] network layers. These are physical layer, data link layer, network layer and transport layer. Each component represents one or more protocols of a certain OSI layer and is connected to components forming part of the upper OSI layer, the lower OSI layer or the same OSI layer. Some components might be part of different OSI layers at the same time as, for example a component implementing NAT needs to have access to network headers and transport headers at the same time. For reliability and protection management reasons, each NIC has its own set of components representing its local network stack. The router application planes's components form part of the OSI application layer and are clients of the forwarding plane's transport layer components. In addition to the implementation of certain networking protocols, the components also contain forwarding logic to forward network packets, by using CAmkES' communication infrastructure, to other components for further processing.

For an overview of the router's components and the two planes see Figure 2.1. The forwarding plane and the application plane containing the different components are colored in grey. The small boxes represent instances of the router's components and are arranged according to the OSI layer to which they belong. Arrows indicate the paths that network packets take between the components; they either are passed directly from one component to the other or are buffered in queues between the components. There are two NICs at the bottom of the network stack, one is part of the global network and one is part of the local network. The NAT component translates network packets in transit between the global network and the local network.

This strategy to divide the network stack into coarse grained components is beneficial to the component's communication overhead as the functionality each component executes becomes relatively expensive compared to the cost of the inter-component communication. Other approaches to component based routers, such as Click[19], use a very fine grained division of their functionality into components. However, the underlying CAmkES component architecture is geared towards components similar to servers in the microkernel context rather than towards support of library like components. Thus it primarily supports the design of software systems based on coarse grained components providing complex functionality and interfaces as opposed to fine grained components with very limited functionality. Despite its composition of coarse grained components, the resulting system is still configurable in a natural way by adding, removing or replacing certain network stack protocols, routing protocols or applications. For example, to extend the current router to a full scale router, more sophisticated routing algorithms and routing/forwarding tables could be added as application layer components. The forwarding plane's components could then query the routing table components in order to find the appropriate NIC to send out a network packet it is currently processing.

As shown in Figure 2.1, the flow of the network packets through the components of the router is in the order of the OSI Layers. This approach has the advantage that it helps define the basic interfaces between the components. A component of the network layer will expect to receive Ethernet frames from the data link layer components (e.g. network drivers), IP packets from other network layer components (e.g. routing components) and transport packets from transport layer components (e.g. UDP component).

The following subsections describe the functionality of the router's components and the CAmkES interfaces they provide and use. In addition to the components depicted in Figure 2.1, there are also helper components present in the router. These mainly consist of queues to buffer streams of network packets and memory allocators to manage memory areas shared by multiple components.

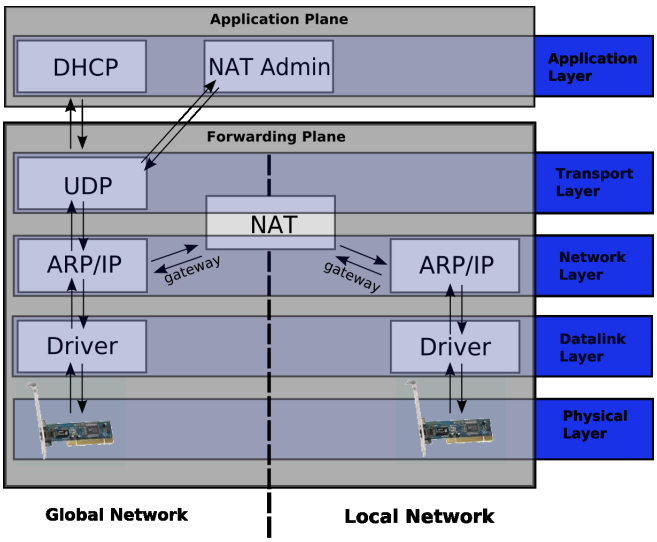

Figure 2.1: The modular design of the NAT router.

2.2.1 Network Driver

The Network Driver resides at the OSI data link and physical layers. It initializes and configures the underlying NIC's hardware and communication link depending on the parameters provided in the router's CAmkES ADL file.

The network driver handles interrupts delivered by L4 as IPC messages and calls the corresponding code in the driver implementation. When packets arrive from the network, the received Ethernet frames are enqueued in a reception queue (RX Queue). If Ethernet frames are available for transmission, it dequeues them from a transmission queue (TX Queue) and sends them to the hardware. The interfaces dealing with the queue are shown in Figure 2.2, 1.

Another role of this component is to provide an interface to other components, which allows them to query or set driver specific information. This information includes the associated NIC's MAC address, IP address, netmask and default gateway address. The actually network related IP information is stored in the driver for simplicity reasons. If this information changes during run-time, the network driver emits an event. This allows other network stack components to refresh their information about the associated NIC. To set up the NIC and network driver in a well defined way, a function to set up this driver from another component is also available. These interfaces are shown in Figure 2.2, 2.

In order to allocate memory to copy the Ethernet frame out of the NIC's memory to, a memory allocator service is used. This allocator service (see Section 2.2.4 allocates or deallocates chunks of memory in a (shared) memory section and makes them available to this component

(and other components if the memory is shared between them). Figure 2.2, 3 shows the component's memory allocation interfaces.

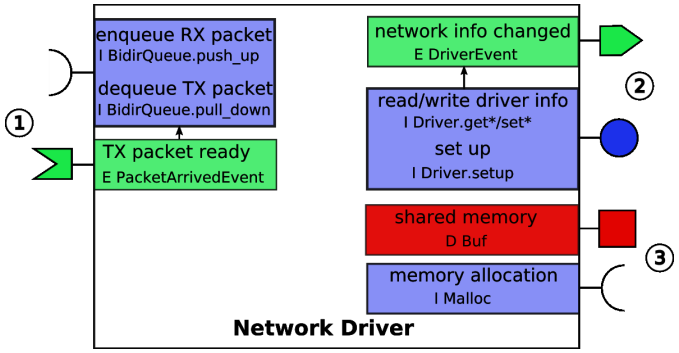

Figure 2.2: The network driver component and its CAmkES interfaces. Legend: see Figure 1.1

2.2.2 ARP/IP

The ARP/IP component implements the ARP and IP protocols of the OSI network layer (see ARP RFC 826[26]). ARP and IP are actually two separate protocols. However, as the IP protocol has to query the ARP component for each Ethernet frame to be sent to the network, these protocols were combined in one component. This increases the forwarding performance of the router by avoiding the overhead of an RPC invocation for each ARP query.

This component handles Ethernet frames after dequeuing them from the driver's RX queue using the interfaces shown in Figure 2.3, 1. Depending on the type of the network packet (ARP or IP) it is then fed to either the ARP protocol implementation or the IP protocol implementation.

The ARP protocol implementation maintains the ARP database and sends out ARP replies and queries to the corresponding network driver component by enqueuing Ethernet frames in the driver's TX queue whose interface is shown in Figure 2.3, 1. The ARP database is used to find out the MAC address that belongs to a certain IP address; this information is needed to create Ethernet frames out of IP packets.

The IP part of the component is connected to other components (except the network driver) by an RPC interface. This means that arriving packets do not have to be fetched actively by this component but are received as parameters of an RPC interface invocation. The same principle is used for packets that arrived on other NICs and were routed to the current network stack for delivery. The interface where these packets arrive (Figure 2.3, 5) is called the gateway interface as it is used to communicate with other networks.

If a received network packet's destination IP address belongs to the local network, an Ethernet frame is created (using the ARP database to look up the MAC Address) and the frame is enqueued to the driver's TX queue. However, if the packet's destination IP address is the address associated with this network stack itself, it is forwarded to the local transport layer components using the interface shown in 2.3, 5. Finally, if the packet's destination address is not part of the local network, it is sent to the gateway interface in order to be handled by the routing infrastructure of the router. This interface is shown in Figure 2.3, 4. Depending on the source and destination of the network packets, IP headers will be added or stripped and certain header fields may need to be edited or recalculated.
The ARP/IP component needs to know about data associated with the network driver (like IP Address, MAC address etc.). Therefore, it receives a notification if the driver changes its parameters during run-time. On reception of such a notification, the component can refresh the information it needs by calling the appropriate RPC interface shown in Figure 2.3, 2.

To allocate or deallocate memory, a memory allocator service is used. This allocator service allocates or deallocates chunks of memory in a (shared) memory section and makes them

available to this component (and other components if the memory is shared between them) through the dataport interface in Figure 2.3, 3.

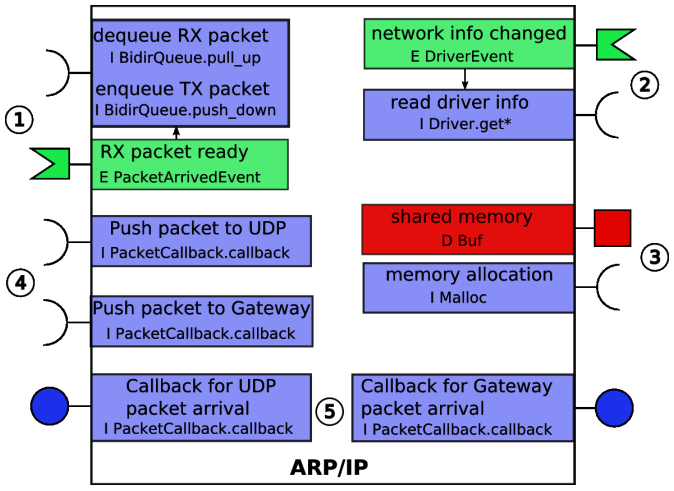

Figure 2.3: The ARP/IP component and its CAmkES interfaces. Legend: see Figure 1.1

2.2.3 UDP

The UDP component implements the UDP protocol at the OSI transport layer.

Applications using the UDP component can bind themselves to a specific IP:port combination by invoking the corresponding RPC interface provided shown in Figure 2.4, 5. If an application is bound to a certain IP:port combination, the UDP component enqueues all arriving packets matching this combination into the corresponding application's RX queue (each application has its own RX queue). Packets from the ARP/IP component are received by the UDP component as parameters of RPC interface invocations using the interface shown in Figure 2.4, 4.

If packets from an application arrive to be processed by the UDP component, they are dequeued from the appropriate application's TX queue (Figure 2.4, 1) before processing and forwarding them to the ARP/IP component (Figure 2.4, 4). Processing includes adding a UDP header and calculating the UDP checksum.

The UDP component needs to know about data associated with the network driver (IP address). Therefore, it receives a notification if the driver changes its parameters during run-time. Upon reception of such a notification, the component can refresh the information it needs by calling the appropriate RPC interface as shown in Figure 2.4, 2.

To allocate or deallocate memory, a memory allocator service is used. This allocator service allocates or deallocates chunks of memory in a (shared) memory section and makes them available to this component (and other components if the memory is shared between them). (Figure 2.4, 3).

2.2.4 Memory Allocator

The memory allocator component is used to manage (shared) memory areas forming part of a memory section provided to other components through a CAmkES dataport interface (Figure 2.5, 1).

It provides an RPC interface to other components so they can allocate and deallocate memory in the associated dataport (Figure 2.5, 2). Due to Iguana's protection domains, this is only possible if the client component using this functionality explicitely shares the corresponding managed

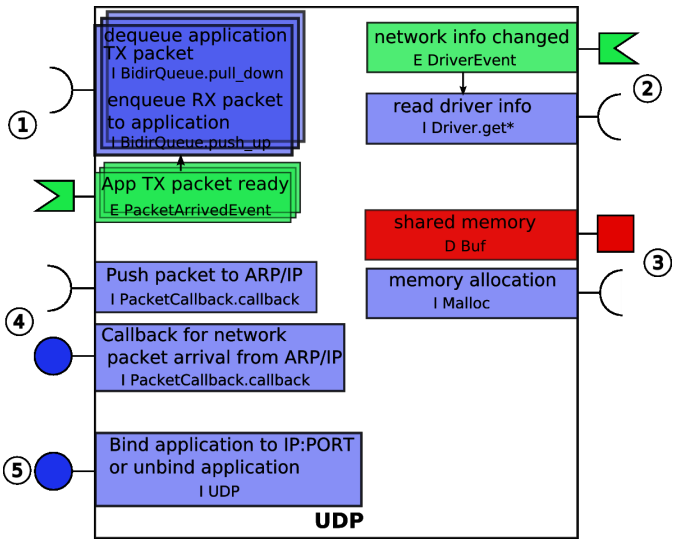

Figure 2.4: The UDP component and its CAmkES interfaces. Legend: see Figure 1.1

memory section with this component (has read/write rights to it).

The algorithms used for the memory allocation can be different depending on the component's implementation. A possible implementation divides the memory section into pre-allocated buffers of a fixed size. Another implementation supports the allocation of memory chunks of arbitrary size by using the K&R Malloc algorithm used in the L4/Iguana ANSI C library's memory allocation implementation. This allows for application-tailored optimization of the memory allocation algorithm used in a system simply by replacing an existing memory allocator component by an optimized one.

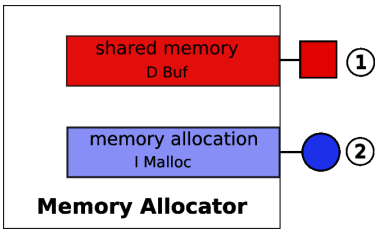

Figure 2.5: The memory allocator component and its CAmkES interfaces. Legend: see Figure 1.1

2.2.5 Bidirectional Queue / Copying Bidirectional Queue

A bidirectional queue component contains two FIFO queues queuing packets travelling in opposite directions. One queue enqueues packets flowing in the direction from the top of the network stack to the bottom of the network stack; it is called the TX queue (also referred to as the down-queue). Packets flowing in the direction from the bottom of the network stack to the top of the network stack are enqueued in the RX queue (also referred to as the up-queue). Components can enqueue and dequeue packets in either queue by making calls to the corresponding interfaces exposed by this component. (Figure 2.6,1).

A network packet is enqueued by passing its reference to one of the FIFO queues, which then stores the reference in its internal data structures. If the corresponding FIFO queue was empty before inserting the new packet, an event is emitted to notify the consumer component of the packet arrival by using the interfaces shown in Figure 2.6, 2 and 3. A component dequeuing a network packet from a queue receives the reference to the dequeued network packet. Since network packets are looked at as references to the memory area where their payload is stored,

a consuming component needs to have access rights to the memory area where the actual data resides. This means that the producer and consumer of a bidirectional queue need to share an appropriate memory area by defining a dataport connection in the application's ADL file.

Using the bidirectional queue, it is not possible to support total memory protection between consumer and producer components (as they need to share the memory area where the packets reside). To obtain total memory protection, a copying bidirectional queue is introduced. This component shares memory with both the consumer and the producer component through two separate dataport interfaces shown in Figure 2.7, 4 and 5. However, the producer and consumer components do not share any memory between each other. If the consumer dequeues a network packet, the copying bidirectional queue copies the network packet's payload from one dataport memory section (the producer's) to the other (the consumer's) and returns a pointer to the copy to the consumer. During this process, the copying bidirectional queue needs to allocate memory in the consumer's memory section and deallocate memory in the producer's memory section. This is accomplished by calling the allocation or deallocation functions of the memory allocator component managing the appropriate memory section by invoking the interfaces shown in Figure 2.6, 4 and 5.

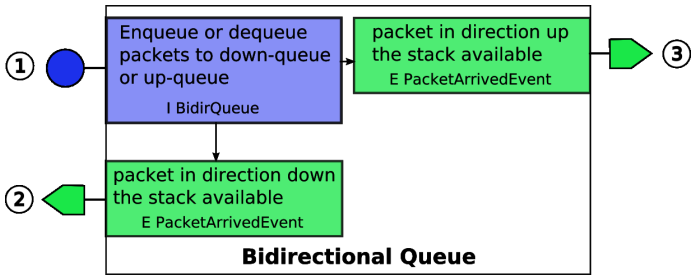

Figure 2.6: The bidirectional queue component and its CAmkES interfaces. Legend: see Figure 1.1

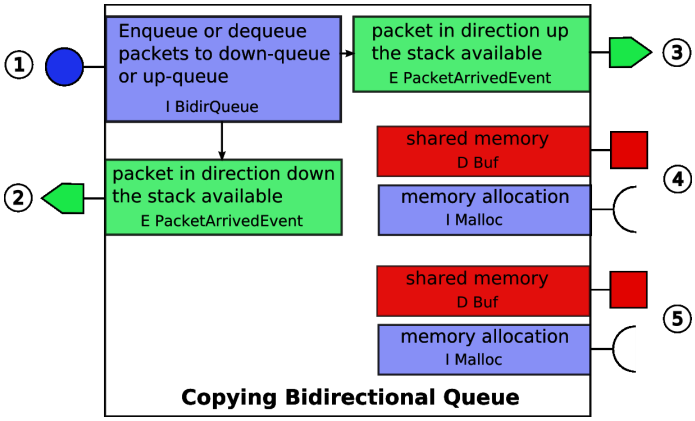

Figure 2.7: The copying bidirectional queue component and its CAmkES interfaces. Legend: see Figure 1.1

2.2.6 DHCP

The DHCP component is part of the application plane and implements DHCP Client functionality as defined in RFC 2131[12].

The DHCP component communicates with DHCP servers by enqueuing datagrams to be transferred to the network in its TX queue and by dequeuing received datagrams from its RX queue (Figure 2.8, 1). DHCP's transport protocol is UDP, so the DHCP component is a client of the UDP component, which enqueues packets directed to the DHCP component in the corresponding RX queue. In order to receive the appropriate network packets, the DHCP client uses its

UDP interface to bind to a specific IP:port combination using the interface shown in Figure 2.8, 3.

To allocate or deallocate memory, a memory allocator service is used. This allocator service allocates and deallocates chunks of memory in a (shared) memory section and makes them available to this component (and other components if the memory is shared between them) through the inferfaces in Figure 2.8, 4.

The DHCP protocol must knwo the MAC address of the NIC it is currently assigning an IP address to. In order to obtain this information, the DHCP component needs to be able to read the network driver's associated data. Furthermore, the DHCP component has to be able to set the IP address, the netmask and the gateway address of the corresponding network driver as it is the role of a DHCP client to automatically request network configuration data from a DHCP server and to alter the configuration of the network driver accordingly. The interface that is invoked during this process is shown in Figure 2.8, 2.

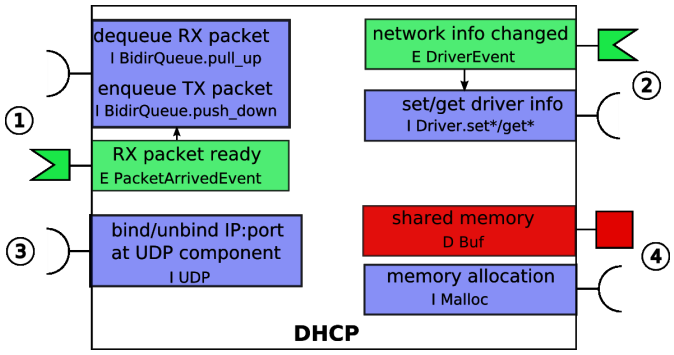

Figure 2.8: The DHCP component and its CAmkES interfaces. Legend: see Figure 1.1

2.2.7 NAT Admin

The NAT Admin component is part of the application plane and provides server functionality to configure the NAT component during run-time from a remote machine. Additionally, remote machines can query the status of the NAT component and request information about NAT statistics and NAT tables by using the NAT Admin component. A Java client was implemented in order to use this functionality (see appendix C).

This component implements a custom protocol running on top of UDP to communicate with an UDP client running on a remote machine. Request datagrams arriving from the client application are enqueued in the NAT Admin's RX queue and can be processed by using the interface shown in Figure 2.9, 1.

After processing a dequeued request datagram, the appropriate function provided by the NAT component is called by using the corresponding CAmkES RPC interface shown in Figure 2.9, 5. These functions are exposed by the NAT component to provide an interface for run-time configuration. The NAT Admin component then generates a reply datagram containing the results of the configuration method invocation to be sent back to the remote client. Finally, the reply packet is enqueued in the component's TX queue for delivery to the network.

Note that CAmkES does not support return types with variable lengths (e.g. arrays). The only way to return types of a variable length is to create a shared memory area between the caller and the callee by defining a dataport interface as shown in Figure 2.9, 5. The result of the function call therefore is copied to the shared memory by the callee and returned to the caller as a reference to the result data structure.
To allocate or deallocate memory, a memory allocator service is used. This allocator service allocates and deallocates chunks of memory in a (shared) memory section and makes them available to this component (and other components if the memory is shared between them).

(Figure 2.9, 4).

During this project, a Java client for the NAT Admin component was developed. It allows its user to configure and query the NAT component from the command line of any networked machine supporting Java.

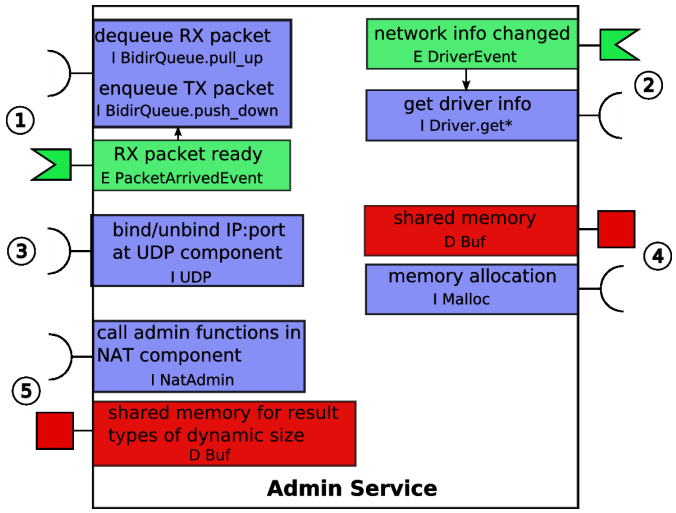

Figure 2.9: The NAT Admin component and its CAmkES interfaces. Legend: see Figure 1.1

2.2.8 NAT

The task of the NAT component is to translate the IP addresses and other data contained in IP packets in transit between the local network and the global network (as described in Section 1.4).

The NAT component receives IP packets from the local and global network stacks' gateway interfaces, which are shown in the overview of the rounter's components in Figure 2.1. These packets are received by the NAT component as parameters passed by reference in RPC interface invocations of the interfaces shown in Figure 2.10, 1 and 2. This usually happens when an IP packet's destination address is not part of the network stack's local network or if the network stack is configured to forward all packets to its gateway. Upon IP packet arrival, the NAT component alters the necessary fields in the IP header and the headers of the transport layer protocol packets (UDP, TCP, ICMP) contained in the IP packet's payload. Finally, the processed IP packets are sent to the gateway to the other network.

The NAT component needs to replace the source IP address of local packets bound to the global network by the IP address assigned to the global NIC. Consequently, it needs to be notified if the IP address of the global NIC changes in order to refresh it. (Figure 2.10, 3).

As opposed to other components, the NAT component does not have to allocate memory for packet headers as it only changes fields in already allocated IP packets. However, the NAT component needs to free the memory of packets that have to be dropped (e.g. packets from the global network that do not match any NAT table entries). Hence, the NAT component still needs to have an interface connecting it to a memory allocator component as shown in Figure 2.10, 4.

In order to allow other components to configure the NAT component during run-time and to query status information, it has to expose an appropriate RPC interface. The CAmkES implementation does not support types of variable length (e.g. arrays, strings) at the moment, so a dataport interface (shared memory) is used to pass these types between the caller and the callee by reference. (Figure 2.10, 5).

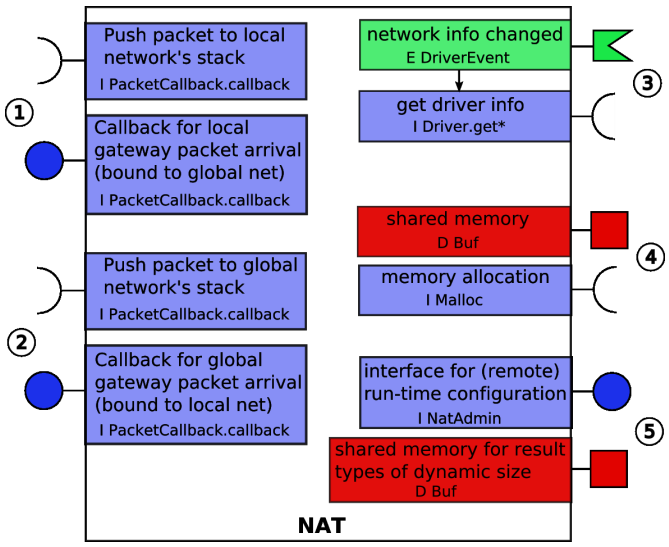

Figure 2.10: The NAT component and its CAmkES interfaces. Legend: see Figure 1.1

2.3 Assembly of the Router

This section explains in detail how the presented CAmkES components are assembled to form the router. A coarse overview was already given with Figure 2.1. However, Figure 2.1 does not include helper components like queues or memory allocators, nor does it specify the interfaces each component provides or uses and the connections between these interfaces.

In Figure 2.11, a detailed overview of the whole router is given. It includes all important components in the router including the interfaces they provide and the connections between pairs of these interfaces.

- Green lines refer to CAmkES event connections. The arrowheads on these lines point from the emitter of an event to its consumer.
- Red lines refer to CAmkES dataport connections. A dataport interface is usually used in conjuction with an RPC interface connected to a memory allocator component (managing the shared memory), hence there is only one line drawn for such a pair (for simplicity). The arrowheads point from the component using the memory allocator's functionality to the associated memory allocator component itself.
- Blue lines refer to CAmkES RPC connections. The arrowheads point from the component calling the interface's functions to the component providing their implementation.

Only the interface types and the pairs of interfaces that are connected together are defined at this point. The implementation of a particular connection in the actual system depends on the chosen connector for the according connection as explained in Section 1.7.3).

The dashed lines on Figure 2.11 divide the router into the forwarding plane and into the application plane (see 2.2). In between these planes are the copying bidirectional queue components (Figure 2.11, 3) which separate the two planes (to provide memory protection) and act as packet buffers between the network stacks and the applications.

All of the components of the forwarding plane share a managed memory section. A memory allocator component is used to allocate and free memory in this memory pool. Network packet related data is allocated in this memory pool and can be passed by reference from component to component. This allows for fast zero copy forwarding of network packets between the components of the forwarding plane (except for the copy out from the NIC's memory and the copy into the NIC's memory). (Figure 2.11, 2).

In order to buffer network packets that arrived from the NIC or that are bound to be sent to the NIC, there are bidirectional queues between the driver components and the rest of the system. (Figure 2.11, 1).

The application plane's application components are mutually protected from each other as each CAmkES component resides in its own protection domain. Each of the application components is connected to its own memory allocator component to allocate memory for outgoing packets or to deallocate the memory occupied by incoming packets (Figure 2.11, 4). This architecture easily allows one to connect an application component closer to the forwarding plane by connecting it to the forwarding plane's memory allocator component instead of to its own (so there are no more memory copies required between this application component and the forwarding plane). This will increase the speed of the system but decrease its security as a misbehaving application component could trash the shared memory of the forwarding plane.

2.4 Connecting the Components Together

The previous section showed how the different component's interfaces are linked together to assemble the router. This section explores how different implementations of these connections shape the actual router. The code implementing a connection is generated automatically by CAmkES depending on the connector that is chosen for it in the application's ADL file.

2.4.1 Connectors

A connector defines the semantics of a connection, i.e., how the two connected interfaces actually exchange their data. These are the connectors used in the router's ADL file:

- IguanaRPC: This is an RPC connector. The caller component sends its marshaled parameters to the callee component by L4 IPC. The callee implements a service loop thread that receives the IPC, unmarshals the parameters, calls the corresponding implementation, marshals the results and sends them back to the caller by L4 IPC.
- DirectCall: This is an RPC connector. The caller component has access to the function pointers of the connected interface's implementation in the callee component. The caller then calls the function directly through its function pointer. This is only possible if the callee component provides the caller access to the memory section where the actual function implementation resides, for example if both components reside in the same protection domain.
- IguanaSharedData: This is a dataport connector. It enables zero copy memory sharing between the connected components by allocating an Iguana memory section and giving all of the components read and write rights to it.
- IguanaAsynchEvent: This is an event connector. If a component emits an event to an event consumer, an L4 asynchronous IPC call is made to the consumer component. The consumer component implements a service loop thread that handles the asynchronous IPC message and delivers it as an event to the component's implementation.

Direct method invocations are significantly faster than L4 IPC calls. This means that RPC interface invocations through connections of type `DirectCall` are a lot faster compared to connections of type `IguanaRPC`. Consequently, `IguanaAsynchEvent` connections are also relatively slow compared to direct method invocations as there is an asynchronous L4 IPC call involved during the event delivery.

When choosing connectors, there seems to be a general trade-off between security and performance. For example, `DirectCall` is faster than `IguanaRPC` but there is less memory protection between the two connected components. This poses a security risk. More in-depth discussion of the different connectors' performance and their influence on the system's performance can be found in Section 4.2 and Chapter 3.

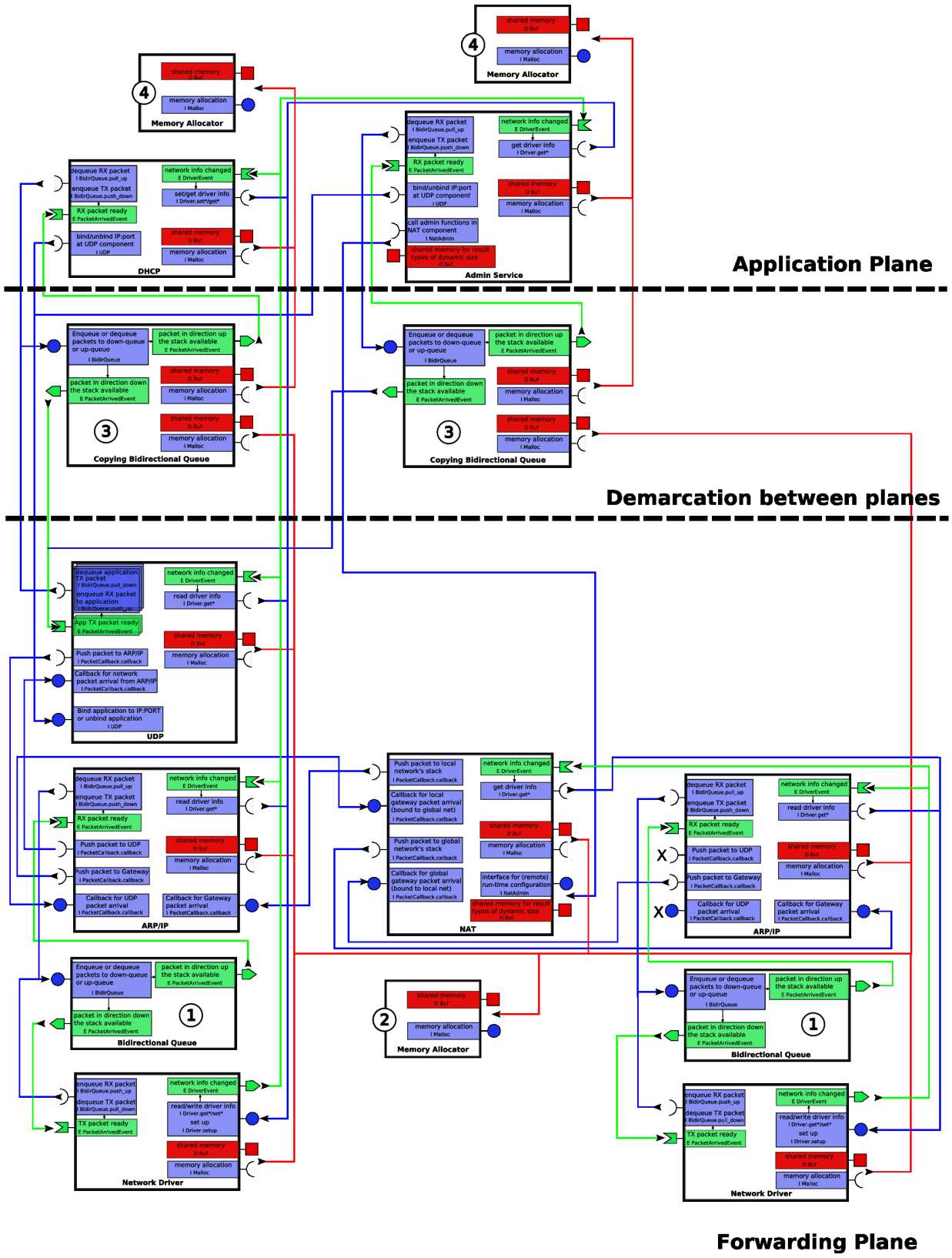

Figure 2.11: The assembled modular router. Legend: see Figure 1.1

It is not always straightforward to replace one connector type with another, for example, to replace `DirectCall` connectors with `IguanaRPC` connectors. As the `IguanaRPC` connector invokes calls via the kernel's IPC mechanisms, systems connected with this type of connector tend to be highly multi threaded. This is because every component has to implement its own

service loop thread to receive and handle the RPC related IPC messages arriving from the kernel rather than allowing the functions to be called directly. This is illustrated in Figure 2.12.

On the other hand, systems whose components are connected via `DirectCall` tend to be more monolithic as the thread making the interface invocation actually executes the provided implementation itself as illustrated in Figure 2.13.
As such, if there is a chain of method invocations across several components, this can lead to a redirection of the calling component's thread across multiple other components before returning. It is therefore not straightforward to replace `IguanaRPC` with `DirectCall` connectors as the possibility of deadlocks is high if we switch from a system using `IguanaRPC` to a system using `DirectCall`. However, the possibility of race conditions between multiple threads is high if we switch from `DirectCall` to `IguanaRPC`. Precautions like placing semaphores and mutexes at appropriate locations must be taken during the design and implementation time of a system, to ensure that replacing `IguanaRPC` by `DirectCall` and vice versa is possible as in the presented router.

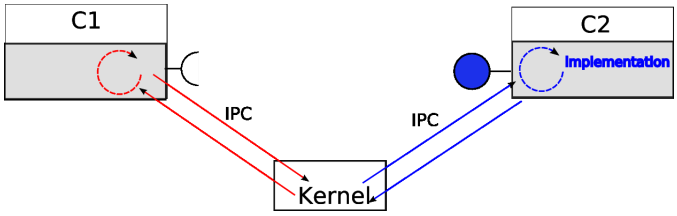

Figure 2.12: IguanaRPC connection. The function provided by C2 is executed by C2's thread (blue) while C1's thread (red) only marshals the parameters and unmarshals the results.

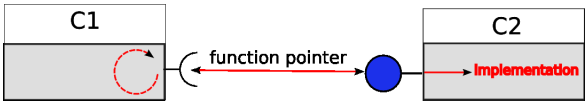

Figure 2.13: DirectCall connection. C1's thread (red) executes the function provided by C2 itself.

2.4.2 Different patterns of how to connect the router's components together

There are different patterns of how to connect the router's components together so that network packets can flow between them. The main patterns are:

Forwarding pattern: One way components can pass network packets to other components is by invoking a function provided by the consumer component. The packet is passed by reference as a function parameter (see Figure 2.14). Another pattern used to pass packets from one component to the other is the well known consumer/producer pattern using a queue between the consumer and the producer component. The consumer enqueues a network packet in a queue by invoking a push function. The queue emits an event to notify the consumer about packet arrival if it was empty before. The consumer's control thread can then dequeue (pull) a packet from the queue when it is ready as illustrated in Figure 2.15. The first approach to passing packets is more similar to synchronous communication between the components while the second approach is asynchronous and allows the producer thread to schedule packet dequeuing. Because of the indirection of the consumer/producer pattern, passing packets directly from one component to the other is much faster.

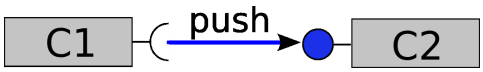

Figure 2.14: Method call to pass a packet from component C1 to component C2.

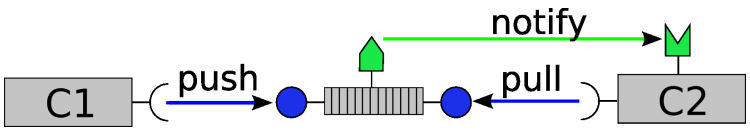

Figure 2.15: Consumer/producer pattern to pass a packet from component C1 to component C2. The queue is also a component.

Memory Sharing pattern: Components can share a memory area by defining a data sharing relation with each other using a dataport interface. This allows for zero copy passing of packets between two components by simply passing a reference to the network packet residing in the shared memory area as shown in Figure 2.16. If no memory sharing between two components is desired for security reasons, a network packet can be copied from the producer's memory area to the consumer's. A component separating the consumer and the producer does the copying between the two memory areas. It needs to have data sharing relationships defined with both the consumer and the producer component in order to be allowed to access their memory areas. This is shown in Figure 2.17. Zero copy passing of network packets is much faster than copying packets from one memory section to the other. However, it may pose a security risk to let two components share a memory area. This especially applies if one of the components is an untrusted component, for example, one that was downloaded from the Internet.

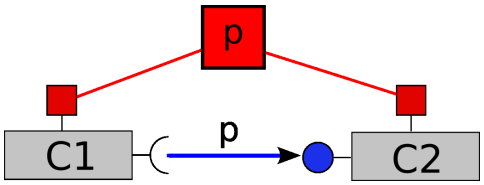

Figure 2.16: C1 and C2 share a common memory area. This supports zero copy passing of packets between the components.

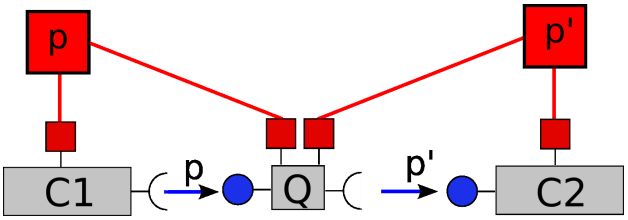

Figure 2.17: C1 and C2 do not share a common memory area. A third component in between them has to copy the data from one memory area to the other.

Connector induced patterns: Depending on the chosen connector, the semantics of a resulting system may change (see Section 2.4.1). Connectors are very powerful abstractions and replacing existing connectors by different ones can lead to changed communication patterns in the system. If DirectCall connectors are used, the resulting system and its ADL definition seem to be more control flow focuses. If IguanaRPC connectors are used, the resulting system and its ADL definition seem to be more data flow focused. DirectCall connectors explicitely transfer the control to the implementation of a called function while IguanaRPC connectors transfer parameters to the component that implements a called function.

A system's properties may differ depending on the patterns chosen to forward packets. There are two extreme systems:

- Queues between all components, no shared memory, only IguanaRPC connectors. This system will be very slow because of the indirection caused by the queues, the various

memory copies required and by using the kernel's IPC mechanisms to invoke RPC interfaces. However, the components are completely protected from each other as they do not share any data and use the kernel's IPC mechanisms to communicate. The resulting system is highly threaded and asynchronous as every component has its own thread of control and is able to independently schedule when to process (dequeue) a network packet.

- Interface invocation to pass packets directly to consumer, shared memory between all components, only DirectCall connectors. This system will be fast as the packets are passed directly to the consumer component, there are zero copies when passing a packet, and the components use function pointers to execute the functionality provided by another component. The components are not protected from each other as they share memory areas where the network packets reside and give each other access to the memory areas where the provided functions reside. The resulting system is single threaded because the use of function pointer invocations requires that the thread at the beginning of a chain of invocations executes everything.

2.5 Patterns Used in the Final Router's Design

The goal of the router's forwarding plane is to forward network packets as quickly as possible. Therefore, it was decided to let the components of the forwarding plane pass their packets directly by method invocations instead of using a consumer/producer pattern. Only the drivers are connected by queues to the rest of the forwarding plane because of the asynchronity caused by the interrupts they have to process. The `DirectCall` connector is used as it is significantly faster than IguanaRPC. All of the forwarding plane's components are in the same Iguana protection domain.

To protect the forwarding plane from potentially misbehaving applications or extensions, the border between the forwarding plane and the application plane completely separates components of the one plane from components of the other plane. This is accomplished by not having shared memory between application and forwarding plane components, by using `IguanaRPC` as a connector, and by putting queues in between both planes to enable asynchronous, indirect communication.

2.6 Implementation of the Router's Components

This section contains some more details about the implementation of the router's components.

2.6.1 Network Stack Components

The lwIP[13] network stack was used as the basis for the implementation of the different network stack components. It is tailored to embedded systems and promised to yield good results. Also, a small Iguana application using it was already present and running.

As the lwIP network stack is a monolithic library, some changes had to be made in order to be able to use it from within CAmkES components in a modular way. These changes mainly affected the calls to upper or lower network layers in the lwIP implementation. In a componentised network stack, the communication mechanisms provided by the framework have to be used. The modularization of the lwIP network stack mainly involved replacing the calls to upper or lower network layers in its implementation by function pointer calls. A component then sets these function pointers in the lwIP implementation to point to the methods that communicate with the corresponding other components.

A component receives a network packet either by dequeuing it from a queue or by receiving it as parameter of an interface invocation. The packet is then passed to a function of the lwIP implementation corresponding to the component (e.g., the `receive` function of a certain protocol). At the point in the lwIP implementation where the processed packet must be passed to an

upper or lower network layer, lwIP invokes the previously set function pointer, which causes the appropriate component method to be invoked. This method then normally passes the packet to the next network stack component using the CAmkES communication mechanisms.
Consequently, most of the network stack components contain a loop to service their input queues and a switching architecture to send the packets out to the appropriate next network stack component.

2.6.2 Network Driver Component

The Linux SMC91C111 network driver (as described in Section 1.9.2) was used as the basis for the network driver component.

Necessary modifications included porting the driver's physical interface initialization functions to L4/Iguana. Furthermore, the driver had to be extended to be general enough to support two NIC's in the system. This involved removing hard coded memory and I/O addresses from the code and adding support to initialize these fields from a CAmkES component. Also, the hardware details like the interrupt number and I/O memory area of the second NIC had to be retrieved.

During the first tests of the router, some bugs in the ported version of the driver were discovered. These mainly had to do with the fact that there was no more thread synchronization present in the driver implementation that was used as a starting point. The original Linux driver had to be consulted to add synchronization primitives at the right places in the L4/Iguana port.

Unfortunately, the current driver does not support DMA but only runs in PIO mode. This means that the CPU has to copy a network packet byte by byte out of the NIC's memory. The same also applies to data copied to the NIC's memory before being sent out to the network. Every such copy involves the overhead of configuring a data transaction and transferring the data over the bus.

2.6.3 NAT Component

The NAT component was implemented from scratch. It supports the translation of IP packets and their payload: TCP, UDP and ICMP packets. (ICMP translation is not explained in this section as it uses a similar principle as TCP/UDP translation).

Upon arrival of a network packet from the local or the global network, information consisting of the packet's source and destination IP addresses and source and destination ports are extracted. This information later is used to query or update the NAT lookup database whose data structures are shown in Figure 2.18. This database is composed of two (layered) hashtables and a set of NAT entries describing the translations the NAT currently has installed.

On the local side of the NAT, a 2-layered hashtable is used to look up translation information because two keys have to be considered for packets travelling from the local network to the global network. On the one hand, the source IP addresses of the hosts in the local network are the keys for the first hashtable layer as packets from different hosts have to be distinguished. On the other hand, the local source port address of the network packet is the key for the second layer hashtable. Both keys together identify a communication channel created by a specific machine in the local network. Upon arrival from a packet from the local gateway, the NAT component retrieves the NAT entry belonging to the corresponding keys. If no NAT entry is present, a new entry is allocated and inserted into the hash table.

An important field for the NAT translation is the unique global NAT port associated with each combination <local source IP:local source port>. This global port is used to multiplex the single global IP address of the router. When a new NAT entry is created, a new unique global NAT port is assigned and stored in the corresponding NAT entry. This unique port identifies the communication channel initiated by the machine on the local network. The destination IP address of the

packet (that is an address belonging to the global network) is also stored. In order support the inverse translation of packets received as a reply for the packet currently beeing sent out to the global network, the new NAT entry is also inserted into the global hashtable using the unique global NAT port as a key.

The translation process from packets in transit form the local network to the global network replaces the source IP of the network packet by the IP address of the router's global network interface. On the transport layer, the source port is replaced by the assigned unique global NAT port looked up in the corresponding NAT entry retrieved from the local hashtables. The translation process also involves a recalculation of the IP and transport layer checksums.

For packets in transit from the global network to the local network, the global hashtable is queried. If the destination port of the packet matches a key in the global hashtable, the corresponding NAT entry is retrieved. Only if the source IP address of the global packet matches the previously stored IP address of the global host is the packet forwarded to the local network. This prevents packets sent by global machines to random ports on the global router interface from entering the local network if they match a previously assigned unique global NAT port by chance. The translation process from the global network involves replacing the destination address and the source address with the previously stored information contained in the NAT entry as well as recalculating the checksums.

Machines on the global network can not initiate a connection to a machine on the local network hidden behind the NAT. In order to support local machines to provide server functionality accessible by machines on the global network, the port forwarding feature of NAT routers is used. A port forwarding declaration consists of a global port (e.g. port 80 for a web server) and a local IP address. Consider the port forwarding example 80->192.168.0.1. If a network packet directed to port 80 arrives on the global side of the NAT, it bypasses the usual NAT translation. Only the destination IP address is replaced by the address defined in the port forwarding declaration (192.168.0.1 in this example). This allows the local host 192.168.0.1 to provide web server functionality that can be accessed from the global network.

2.7 Discussion of the Router's Design and Implementation

2.7.1 Implementation of the Defined Functional Requirements by the Router

The NAT component presented in Section 2.2 implements the functionality required to share a single global IP address between machines on the local network. The NAT component also supports the forwarding of global packets sent to a given global port to a prespecified local machine.

The NAT Admin component provides server functionality to remote machines to configure the NAT component and query its state. This can be used e.g. to forward certain ports, retrieve the number of packets that were dropped by the router or to disable the network address translation.

To dynamically obtain an IP address in the global network, the DHCP component implements the DHCP client protocol and assigns the values obtained from the DHCP server to the corresponding components.

2.7.2 Extensibility and Configurability of the Router

As defined in the functional requirements (see Section 2.1), the router's architecture has to be flexible and configurable in different ways.

Note that CAmkES does not support dynamic creation of components and thus all router configurations are static and require recompilation, reload and a reboot. However, the addition of support for dynamic components will be added to CAmkES in the near feature.

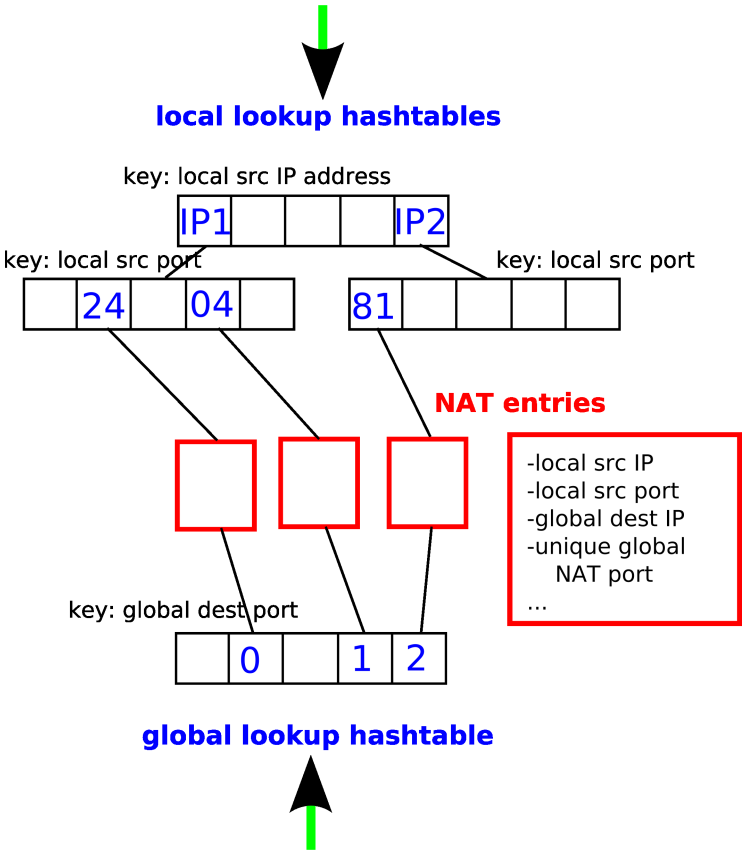

Figure 2.18: The hashtables used to look up a NAT entry containing the information for the network address translation.

The current design supports the upgrade or replacement of existing components. This is possible by implementing a CAmkES component that provides and uses the same interfaces as the old component. If a new component matches the interfaces of the old component, the component type in the application's ADL file can simply be replaced by the new type in order to get an updated router version. An example of such a component replacement would be to replace a NIC driver component by another driver implementation to support different NIC hardware.

It is straightforward to add additional functionality to the existing router. The application plane has specially been designed to enable the addition of untrusted applications or components. By defining the corresponding connections to other components in the application's ADL file, a new application component can be integrated into the system. If the new component is a trusted component, it can also be integrated into the forwarding plane to speed up the communication process with other components in the forwarding plane.

A special kind of extension component is a component that is actually added to the data path in the router. This means that network packets traverse the extension component on their way through the router. As a concrete example, a component implementing stateful packet inspection (SPI) could be added on the global side of the router between the ARP/IP component and the NAT component. This SPI component would block certain traffic from entering the local network by observing the traffic and detecting certain prohibited patterns in it. Another extension would be to add a component that provides TCP functionality.

Finally, all of the components are general enough to use them in a different setup. Especially the network stack components could be reused in other CAmkES based applications needing network support. The interfaces of the components are easy to use and a developer should have an application-tailored network stack based on the developed components up and running in a short time.

The router's NAT component is configurable online by using an administration service running as an application on it using UDP as a transport protocol. This service could easily be extended to also configure other components (like firewalls) online as well.
Also, by using appropriate attributes in the application's ADL file, the memory area, interrupts etc. associated to a NIC can be configured offline. Furthermore, it is possible to configure the behavior of the ARP/IP component by editing the appropriate attributes.

Another type of configuration refers to the security and performance trade off in the system. Faster (less secure) connections can be replaced by slower (more secure) connections between the components and vice versa. This is simply achieved by changing the connector definition in a particular connection declaration in the application's ADL file.

2.7.3 Security and Robustness of the Router

The router's architecture distinguishes between trusted and untrusted components. Untrusted components can be put into the router's application plane. If they crash, they can not crash other components in the system as each component in the application plane resides in its own protection domain. Components containing confidential information like passwords can also be put into the application plane so that no other component has access to the confidential data. The only way to communicate with such a component is to use the kernel's IPC mechanisms (as implemented in the `IguanaRPC` connector) and well defined RPC interfaces declared in the application's ADL file.

Security in the application plane is achieved by reusing the well-tested lwIP network stack rather than implementing an own network stack. A new network stack implementation would be more likely to crash as the implementation would contain more bugs than the mature lwIP code. Furthermore, existing l4/Iguana libraries were used whenever possible to reduce the amount of untested code in the system.

During execution of benchmarks and tests, the router's implementation seemed to be very stable despite its early development state.

2.7.4 Suitability of the Router for Embedded Systems

The router benefits of all of the properties features of CAmkES and L4/Iguana related to embedded systems. Furthermore, it also makes it possible to deploy the router on a variety of embedded platforms supported by L4/Iguana. The used network stack is a light weight implementation of the TCP/IP stack specially aimed at embedded systems. The implemented NAT component was designed to be light weight as well. This helps save memory and CPU load in the resulting router.

Memory Requirements of the Router

The memory requirements of the whole router application were investigated as part of this project. As a first part, the minimal amount of memory to load the router application (including L4/Iguana) is presented. As a second part, the run-time memory needs of the router are discussed.

The pie chart on Figure 2.19 shows the relative memory usage (including the `text`, `data` and `bss` segments) of the different router components and the L4/Iguana related code in the system. Note that some of the components occur multiple times in the assembled system. The slices representing L4/Iguana related code are pulled out.

The total memory usage of the whole system is approximately 2.2MiB. The router related code including all components occupies around 80% of these 2.2MiB (see Figure 2.19). The code

of the complete system can be stored in read-only memory (ROM) as no write access to it is necessary.

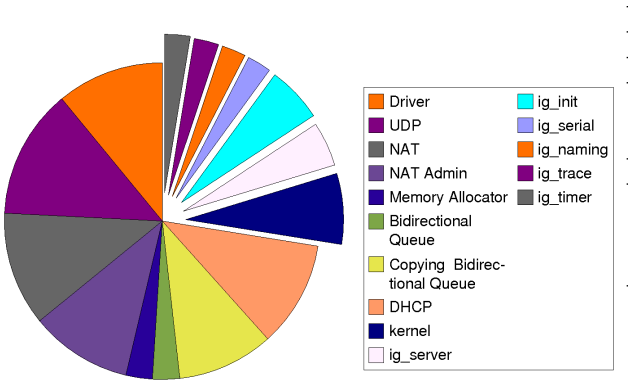

Figure 2.19: Code memory used by the router's different components (also shown is L4/Iguana related code).

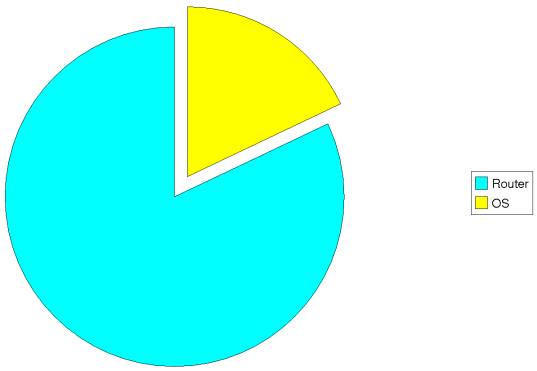

Figure 2.20: L4/Iguana related code uses about 20% of the 2.2MiB totally required for code.

To get an estimate of the amount of dynamic memory used during run-time, the functions `morecore` and `malloc` were instrumented. The memory that the router's and the system's components reserve during the startup of the router is about 41KiB. However, only about 7KiB of this memory finally is actually allocated by calls to `malloc`. Depending on how many packets are buffered in the router at a particular time (depends on the configured maximum queue lengths), this memory amount increases. If no packets have to be buffered in the router, this memory amount stays constant. Not included in these 41KiB is the memory used for local variables and other data allocated on the stacks.

The minimum hardware platform therefore needs to provide at least 2.2MiB of ROM and hundreds of kilobytes of RAM in order to be able to run the router.

2.7.5 Relation to other Component Based Router Architectures (Click, XORP and Scout)

There are no modular router architectures aimed especially at embedded systems. However, there are many different projects that address the general problem of designing and implementing an extensible yet fast modular router. Depending on the targeted environment and use of these routers, they sport different properties and features.

The Click Modular Router

The Click Modular Router architecture[19] is based on composing many very fine grained elements to produce a router that implements desired behavior. Packets are pushed by the device driver through a set of elements until they hit a queue where they wait until they are pulled from the queue by the next component. There are two main interfaces: a push interface to push packets to another element and a pull interface to retrieve packets from another element. This is similar to the components in the CAmkES based router that either forward packets directly from component to component (push) or dequeue packets buffered in queues (pull).

As the Click architecture uses very fine grained components (e.g., an element only increasing a single counter), it is different from the coarse grained design of the router presented in this thesis. The CAmkES component architecture is geared towards components that are similar to servers in the microkernel context and provide complex functionality. It is not appropriate to incorporate single Click elements in a CAmkES based application. However, it would make sense to wrap a whole Click system composed of several elements (e.g. implementing a certain network protocol) into a CAmkES component and be included in our router.

Click resides in the Linux kernel. A kernel thread runs the Click's router driver, which processes a task queue and executes each pending task in the router's components. Consequently, there is no protection between the different components; this means that the whole router crashes if a single component does. The security level for each component can not be configured in Click.

XORP - eXtensible Open Router Platform

The goal of XORP is to develop an open source router platform that is stable and sophisticated enough for production use, and flexible and extensible enough to enable network research.

The XORP[17] [32] design philosophy stresses extensibility, performance and robustness. For routing and management modules, the primary goals are extensibility and robustness. These goals are achieved by separating functionality into independent UNIX processes with well defined IPC interfaces between them.
For the forwarding path, the primary goals are extensibility and performance. Robustness here is achieved through simplicity and a modular design that re-uses well-tested components.

These XORP architecture goals share many similarities with the current CAmkES router's design.

First, the XORP components are relatively coarse grained. They provide complex functionality (like routing protocols) and are similar to servers in a microkernel context as they run in user space and communicate with the rest of the system by IPC. It should therefore be possible and rewarding to port XOPR components to CAmkES components and integrate them with the existing router components.
Applications (routing and management modules) run in different UNIX processes that communicate via well-defined IPC interfaces between each other and the part of the router that does the actual packet forwarding. This is similar to CAmkES components which run in separate protection domains and communicate using RPC.

As opposed to XORP, the actual routing process of the CAmkES router is also part of the forwarding path. The NAT component includes both routing and forwarding of packets at the same place for performance reasons. For more sophisticated routing protocols, like the ones implemented in XORP, this functionality could easily be moved to the application layer.
In the forwarding plane of our router, robustness is achieved in a way similar to that in XORP: simplicity and a modular design based on the well-tested lwIP network stack.
Another interesting fact is that the XORP project implemented a novel inter-process communication mechanism (called XRL) that allows processes on one machine to communicate with processes on other machines (similar to RPC). This enables the XORP router's processes to run in a distributed way on multiple machines. By implementing and using a special user defined

connector in CAmkES, this feature could also be enabled in a transparent way in the presented CAmkES router.

Finally, it is also possible to compile most of the XORP modules that normally run in different UNIX processes together to run them in a single UNIX process. Such a XORP router is faster but the crash of a single component then would crash the whole router. The process of moving all of the XORP modules into one process is equal to the process of replacing all `IguanaRPC` connectors between the application plane and the forwarding plane of the CAmkES based router by `DirectCall` connectors.

Scout

Scout[24] is a modular, communication-oriented operating system. Its major abstraction is the path: a linear flow of data that starts at a source device and ends at destination device. Paths are composed of stages, which are instances of modules. Each Scout module implements a well understood protocol, such as IP or TCP. Before a packet is sent through the router, Scout looks up the path for the packet by running a classification function in each module. The classification function decides to which subsequent module a packet has to be sent. Only after the whole path has been looked up, is a packet scheduled to run along the path. This allows Scout to manage all of the ressources that are used along a path. These include CPU, I/O bus, memory buffers and TLB.

Scout modules wrap single protocols in different modules. A packet is then forwarded along a path through these modules. This is similar to the CAmkES router. The main difference between Scout and the CAmkES based router is, that in Scout a complete path for a packet is looked up before the packet actually is sent. In the CAmkES router, the forwarding decision from component to component is made online, when a packet is in the system.

Scout seems to be more of an experimental platform as compared to Click and XORP. It is not clear how stages (instances of modules) relate to operating system abstractions like protection domains and processes and how the modules exchange data with each other.

A Model for Modular Router Architectures and how the CAmkES Router Relates to it

A comparison of different modular router architectures (including Scout and Click) is presented in [15]. This comparison is based on a simple model.
This model includes three different types of components in a system. All of these component types can change the state of the router.

- classifiers: forward packets to different modules depending on packet's properties
- scheduler: decides which packet to process next if there are several possibilities available (e.g. two queues to service)
- forwarder: simply forwards a packet from its single input to its single output

The main components in the CAmkES router fit to this model as follows:

- Driver: forwarder, as it forwards packets from the network to the RX queue and packets from the TX queue to the network
- ARP/IP: classifier, as it forwards packets either to the local stack or to the gateway. It is no scheduler as only one queue is serviced.
- UDP: classifier and scheduler. It forwards UDP datagrams to different clients depending on the destination UDP port. It also schedules processing of UDP datagrams arriving in different queues form different applications.
- NAT: forwarder. The NAT component forwards packets from one gateway to the other.

Generally, the current router's architecture does not pose any restrictions on how a component is composed of the three different types of the model. To generalize, a CAmkES router component can be a scheduler, a forwarder and a classifier at once. If we analize the UDP component we find that there is actually scheduler logic (decide on which packet to dequeue), forwarding logic (forward a dequeued packet to the next step inside the component itself) and classification logic (decide to which other component has the packet to be forwarded) in the one component.

Chapter 3

Performance Evaluation of the Router

This chapter presents and compares performance measurements of different router configurations. The first Section addresses the performance of the componentised network stack while the further Sections investigate the performance of the routing process including the NAT translation.
Standard benchmarks for routers include measuring the router induced delay as well as the achieved throughputs. The CPU load of the router during these benchmarks represents another important variable; it shows wether the CPU is the bottleneck during a test or if other aspects of the system may cause the results. Furthermore, a program that uses less CPU time is more energy efficient, which is important for embedded systems with limited power supplies.

3.1 Test Environment

In all of the following benchmarks, the router software was running on the following hardware (see also Figure 3.1):

`Gumstix Connex 400xm, ARM XScale/PXA255, 400MHz, 64MB RAM, netDUO-mmc extension board (10/100 Mib/s).`

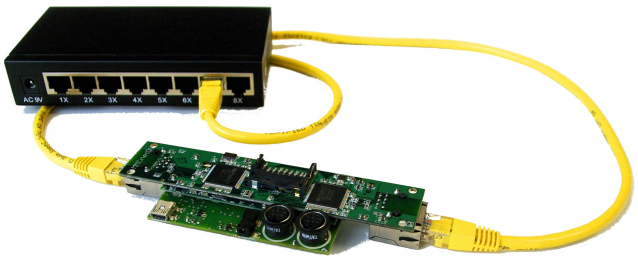

Figure 3.1: Gumstix Connex 400xm and netDUO-mmc extension board connected to a switch

Note: as stated in Section 2.6.2, the network driver used unfortunately does not support DMA and data sent from/to the NIC has to be byte-wise copied over the bus by the CPU. Also, there seem to be some I/O related problems in the network driver used as discussed in Section 3.4).

Depending on the actual benchmark scenario, different measuring tools were used. These tools are further explained in the Sections discussing the actual tests.

There is no practical built-in way to directly measure the CPU load in the L4/Iguana based router. To be able to still measure the CPU loads during the execution of the tests, we used the

following approach. A thread with the lowest priority is started at boot time of the router. The thread executes a loop where a counter in a register is increased. As the thread running the loop has the lowest priority, it is only executed if there are no other ready threads in the system. Consequently, the counter value is not increased if the CPU load generated by higher priority threads is 100%. To get an estimate for 0% CPU load, the rate at which the counter increases when no other threads are running in the system was measured.
During the benchmarks themselves, the value by which the counter increased during the test was divided by the test's duration to get the rate at which the counter increased. To finally receive the CPU load, this rate was divided by the pre-calculated rate for 0% CPU load and subtracted from 100%.

A useful feature of the XScale/PXA255 processor is its timer register. The value of this register is increased by one after a defined number of CPU cycles have passed. To measure the execution time of an interesting code Section, the value of the timer register is stored in a variable before the Section begins. After the code Section to be measured, the difference to the current timer register's value is calculated. By using the clock frequency at which the CPU was running during the measurement, the execution time of the measured code Section can be calculated.

3.2 Network Stack Delay Benchmarks

In this Section, measurements of the delay induced by the componentised network stack are presented. The delay is measured by using an UDP echo service on top of the router's network stack (see Figure 3.2). It works similar to the ICMP echo service (`ping`) but resides in the application layer and uses UDP as a transport protocol. The echo service receives UDP datagrams (echo requests) and sends them back to the sender (echo replies).

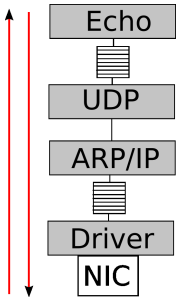

Figure 3.2: Network stack components with echo service running on top.

3.2.1 UDP Echo Delay as Measured from a Remote Machine

An UDP echo client sends UDP echo request packets to the router and measures the time until the echo reply arrives. This data shows the total echo delay including the delay caused by the router's network stack and the delay caused by the transmission over the LAN. As the delay depends on the network packet's size, measurements using different UDP packet sizes were carried out. As design under test, different versions of the router were used.
To compare these measurements to other network stacks, the same echo service was also implemented on top of a monolithic lwIP stack (Version `STABLE-1_1_1`) and as a Linux program (running on Linux for Gumstix as part of the Gumstix Buildroot[4], based on the Linux Kernel 2.6.18).

The results are plotted in Figure 3.3. For an explanation of the version numbers see the version history at appendix B. The data series shown in the graph are:

1. Router Version 0.9. All components are connected by queues. A network packet is copied when passed from component to component. Only IguanaRPC connectors are used.

2. Based on Router Version 0.9. There are only queues between the Driver and the APR/IP components and between the Echo and the UDP components. A network packet is copied when passed from component to component. Only IguanaRPC connectors are used.
3. Router Version 0.98. Design as described in Section 2.5. However, all DirectCall connections are replaced by IguanaRPC connections.
4. Router Version 0.98. The final design as described in Section 2.5.
5. Router Version 0.96. There are queues between the Driver and the ARP/IP components as well as between the Echo and the UDP component. There are no memory copies when passing a network packet up/down the stack and only DirectCall connectors are used.
6. Monolithic lwIP network stack. This is the standard lwIP network stack running as an L4/Iguana server. The NIC driver runs as a separate L4/Iguana server and passes packets by reference to the lwIP server. The UDP echo service runs on top of the lwIP stack in the same protection domain.
7. Linux for Gumstix (driver uses DMA).

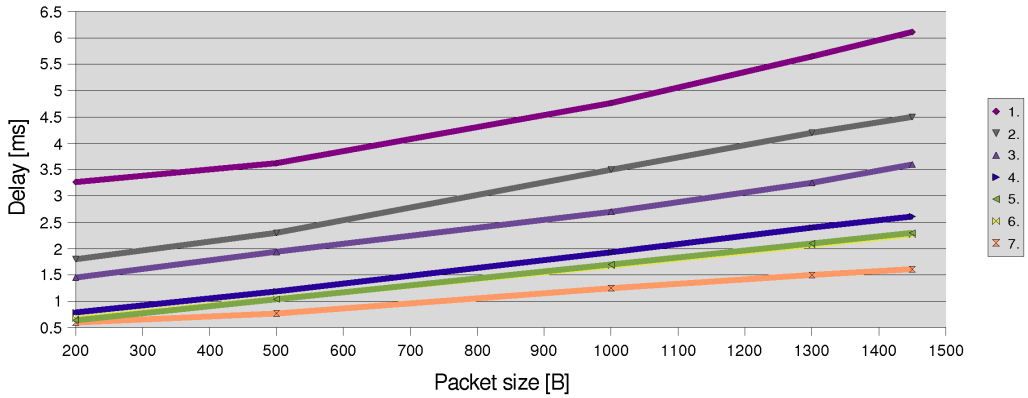

Figure 3.3: UDP echo delay (Return Trip Time).

Data series 5 is very close to the implementation using a monolithic lwIP stack (Data series 6) as it is optimized by not using IPC. Data series 4 shows that for a relatively small performance penalty, the system can be made more secure by implementing the final router design as described in Section 2.5. Data series 1-3 show that the delay caused by the system increases if DirectCall connectors are replaced by IguanaRPC connectors, network packets have to be copied from component to component, and queues are added in between the components. The echo service implementation running on Linux (Data series 7) is the fastest.

Since the network stack in the router is modular, it is expected that it would use more CPU time because of the overhead induced by the component architecture (such as marshaling/unmarshaling and too general code). To get an idea if the CPU is a bottleneck in the current router's network stack and is the cause of the increasing gap between data series 4 (final router design) and data series 7 (Linux echo service), the average CPU load during the echo service tests was measured. Figure 3.4) shows the CPU usage related to data series 4 and 7.

Data series plotted in Figure 3.4.

1. Router Version 0.98. Average CPU load while executing echo delay measurements.
2. Linux for Gumstix (`Kernel 2.6.18`). Average CPU load while executing echo delay measurements.

This plot shows that the cause for the increasing echo delay for larger packet sizes is not the router's CPU since the CPU usage decreases while the echo delay increases. Compared to Linux on Gumstix, the router has a lower CPU load for most of the packet sizes; Linux for Gumstix can process echos at full CPU load while the router's processor is not fully loaded.

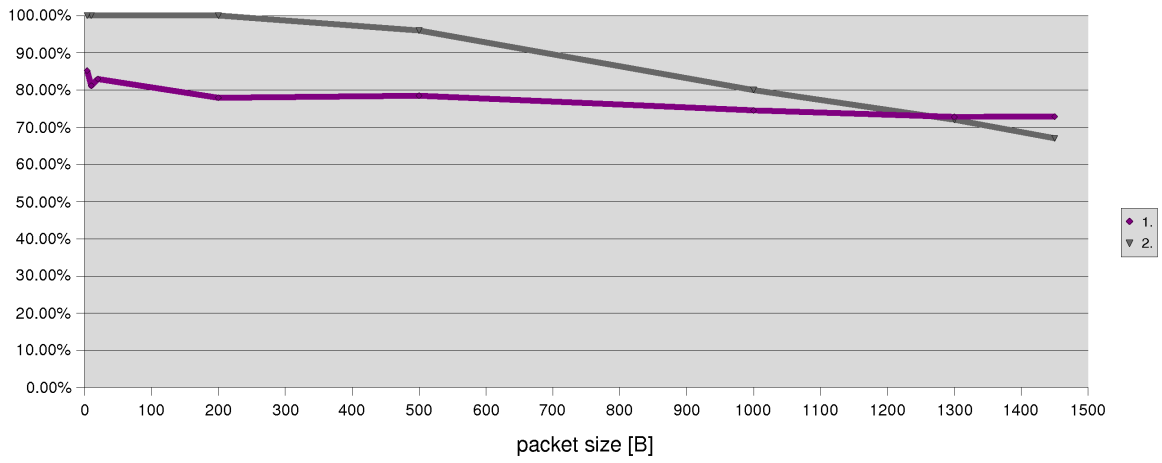

Figure 3.4: CPU usage during UDP echo delay measurements (see Figure 3.3).

3.2.2 Echo Delay Measured in Stack's Components

These measurements use the PXA255 timer register to measure the delay each component experiences between receiving an echo request packet from a lower network layer component until the packet's echo reply is sent to the lower network layer by the current component. This is the total time that the current component and all other components above the current component in the network stack spend processing the echo request (see Figure 3.5). These measurements depend on how fast packets are forwarded in the network stack from component to component and how much time each component takes to process a packet.

To measure this delay, the PXA255 timer register's value is stored in each component when an echo request packet arrives from a lower network stack component. When the same component sends the corresponding echo reply back to the lower network layer component, the difference between the current PXA255 timer register's value and the stored value is calculated to get the delay. These measurements can be turned on and off for each component by editing the `include/debugprint.h` file in the application's directory. The actual measuring points can be found in the corresponding component's implementation. There are measurements for different router versions.

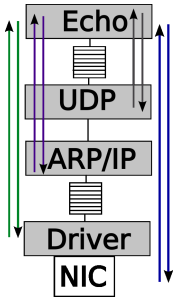

Figure 3.5: Delay measured in each network stack component (with PXA255 timer register).

Router Version 0.98, (see Figure 3.6)

1. Driver component. Measurement points are just before the driver copies the echo request via the bus from the NIC's memory to the main memory and just after the driver copies the echo reply to the NIC's memory via the bus (includes bus read/write).
2. Driver component. Measurement points are just after the driver copies the echo request via the bus from the NIC's memory to the main memory and just before the driver copies the echo reply to the NIC's memory via the bus (does not include bus read/write)

3. ARP/IP measurement points
4. UDP measurement points
5. Echo service measurement points

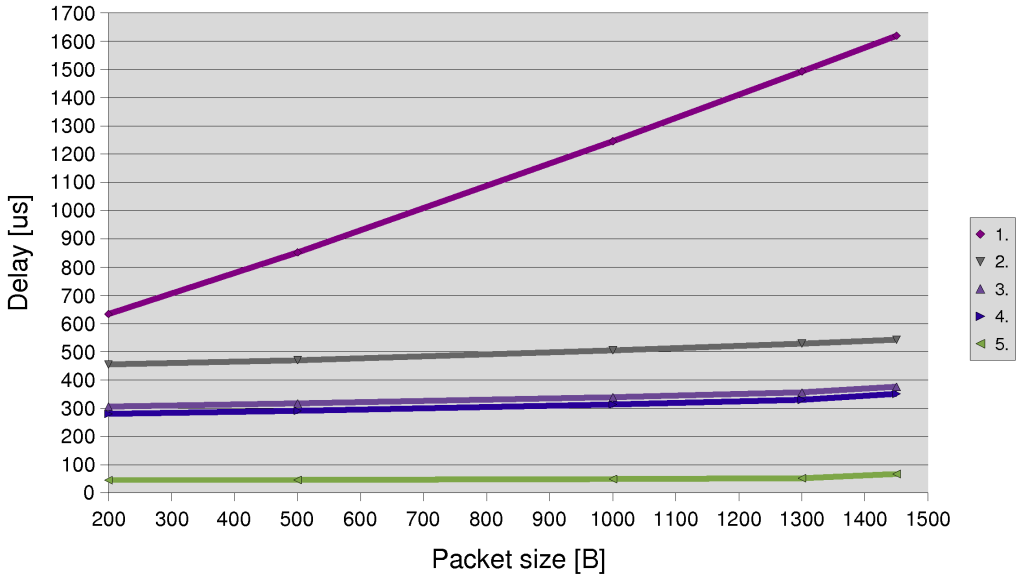

Figure 3.6: Echo Delay Measured in Stack's Components. Router version 0.98.

The gap between data series 1 and 2 is caused by the network driver while exchanging data with the NIC's memory over the bus. It is assumed that driver related problems are a main cause of the divergent data series 1 and 2. The gap between data series 2 and 3 is caused by the RX and TX queues between the driver component and the ARP/IP component. Data series 3 and 4 are very close together as there are no queues between the ARP/IP and the UDP components. As there is a copying queue between the Echo component and the UDP component that is connected with IguanaRPC, the gap between data series 4 and 5 is relatively large. It becomes larger as the packet size increases since there is more time spent on copying the data from the forwarding plane to the application plane.

Router Version 0.96, (see Figure 3.7)

1. Driver component. Measuring points are just before the driver copies the echo request via the bus from the NIC's memory to the main memory and just after the driver copies the echo reply to the NIC's memory via the bus (includes bus read/write).
2. ARP/IP measuring points
3. UDP measuring points
4. Echo service measuring points

To compare the final design to the most optimized router version (version 0.96), the same data was measured using router version 0.96. The results are plotted in Figure 3.7.
In Figure 3.7, the gap between data series 1 and data series 2 is caused by the delay induced by the queue in between the driver and the ARP/IP components as well as the time it takes to copy the network packet out of the NIC's memory. There is also a delay between data series 4 and data series 3 because the UDP component and the echo service are connected by a queue as well. The other delays on the plot are very small compared with router version 0.98 on Figure 3.6.
In total, this network stack is about 0.1ms faster than the one from version 0.98. This is because all of the components are connected by DirectCall connections. Also, the data series are nearly independent of the packet size (except for the driver) as there are no memory copies involved while the packet travels up and down the stack.

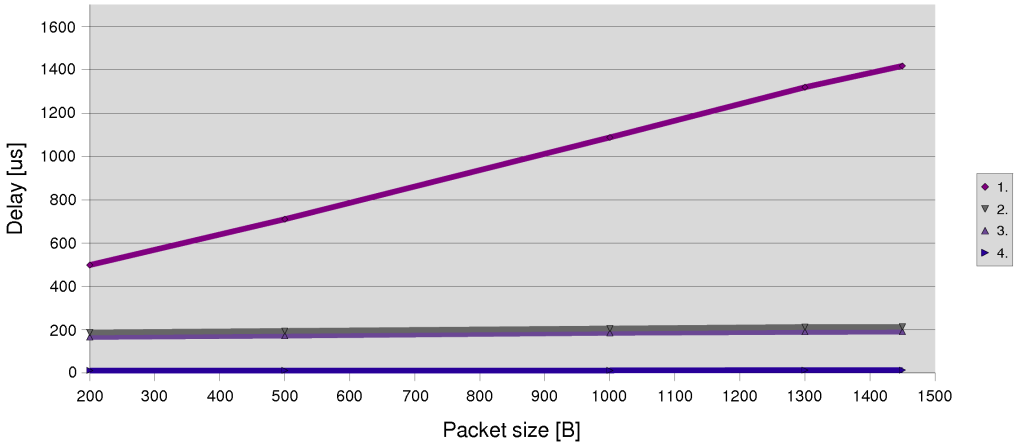

Figure 3.7: Echo Delay Measured in Stack's Components. Router version 0.96.

3.3 NAT Benchmarks

In this Section, benchmarks of the actual NAT routing process are presented. This includes the traversal of a network packet from the NIC where it arrives up the network stack to the ARP/IP component. Packets bound for another network are forwarded to the NAT component where the network address translation is executed. Finally, the packets travel down the network stack corresponding to the other NIC and are sent to the other network.

3.3.1 NAT Delay Benchmarks

To measure the delay that is caused by the router, ICMP echo request messages were sent from a machine in the local network via the router to a machine in the global network. The time between sending the echo request until receiving the echo reply includes the router induced delay caused by the network stack and the network address translation. As compared to the echo service running on top of the router's network stack, an echo request and the corresponding reply are written twice to the NICs and are also read twice from the NICs by the drivers.
The program used to acquire this data was `ping` in flooding mode (`ping -f -s host`). In the routing table on the machine sending out the echo requests, the gateway for the target machine of the echo request was set to the router's local IP address. In flooding mode, echo requests are sent out as fast as possible. In the end, the average packet delay is returned. As the delay depends on the packet size chosen with the `ping` parameter `-s`, measurements for different packet sizes were carried out.

ICMP Echo delay, Router Version 0.98, (see Figure 3.8)

1. Router Version 0.98, with all DirectCall connectors replaced by IguanaRPC connectors. The components are in different protection domains.
2. Router Version 0.98.
3. Router Version 0.98, with all DirectCall connectors replaced by IguanaRPC connectors. The components are in the same protection domain.

ICMP Echo, CPU usage, Router Version 0.98, (see Figure 3.9)

1. Router Version 0.98, with all DirectCall connectors replaced by IguanaRPC connectors. The components are in different protection domains.
2. Router Version 0.98.

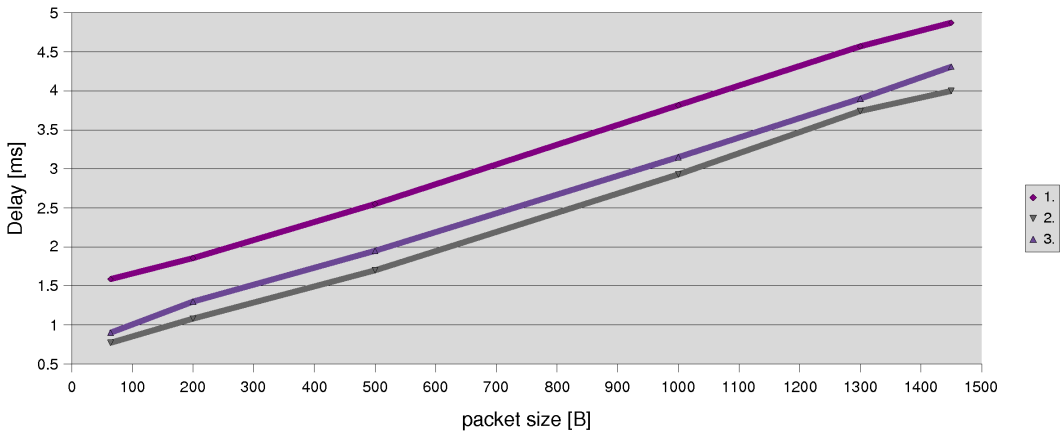

Figure 3.8: Delay between an ICMP echo request and the corresponding reply. ICMP requests were sent from the local network via NAT to the global network.

3. Router Version 0.98, with all DirectCall connectors replaced by IguanaRPC connectors. The components are in the same protection domain.

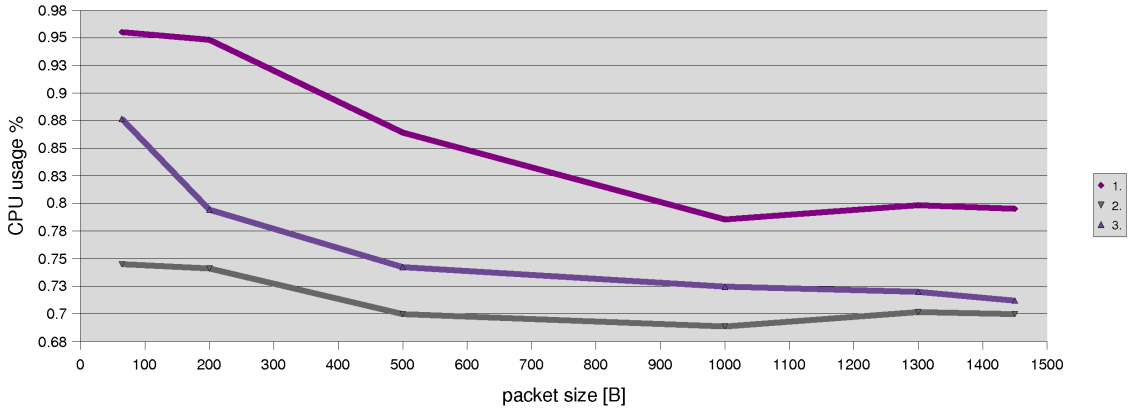

Figure 3.9: CPU usage during the ICMP echo test. ICMP requests were sent from the local network via NAT to the global network.

The final design using DirectCall connectors in the forwarding plane (Figure 3.8, data series 2) is faster than the same design where only IguanaRPC connectors are used (Figure 3.8, data series 3). If the components are also placed in different protection domains (Figure 3.8, data series 1), the delay increases again.
The same happens to the CPU loads measured in the corresponding systems (see Figure 3.9). The final design has the lowest CPU loads as DirectCall connectors are used in the forwarding plane. This shows that DirectCall connectors make a system faster and help save CPU time.

To see how the NAT logic in the router delays an ICMP echo packet, measurements using the PXA255's timer register were made. This time, the command "`ping -s[packet size] host`" was used. This is the standard ping command.
The timer register's value was stored when an ICMP echo packet arrived on the local NIC (just before sending it up the stack) and the difference with the current timer register value was calculated when the same packet was sent out by the other NIC (just before sending the packet to the NIC). The same procedure was carried out for the corresponding ICMP echo reply packets that arrived on the global NIC bound for the local network.
Figure 3.10 shows that ICMP echo request packets are delayed by about 200us while ICMP echo reply packets are delayed by about 300us. These delays do not include any NIC related instructions in the network driver. They only include the delay caused by the NAT and the traversal of the network stacks in the router's software from the local driver component to the

global driver component and vice versa. They are also independent of the network's speed. The data series shown in Figure 3.10 are:

1. Router Version 0.98, ICMP echo request, delay induced by NAT and router's network stacks.
2. Router Version 0.98, ICMP echo reply, delay induced by NAT and router's network stacks.

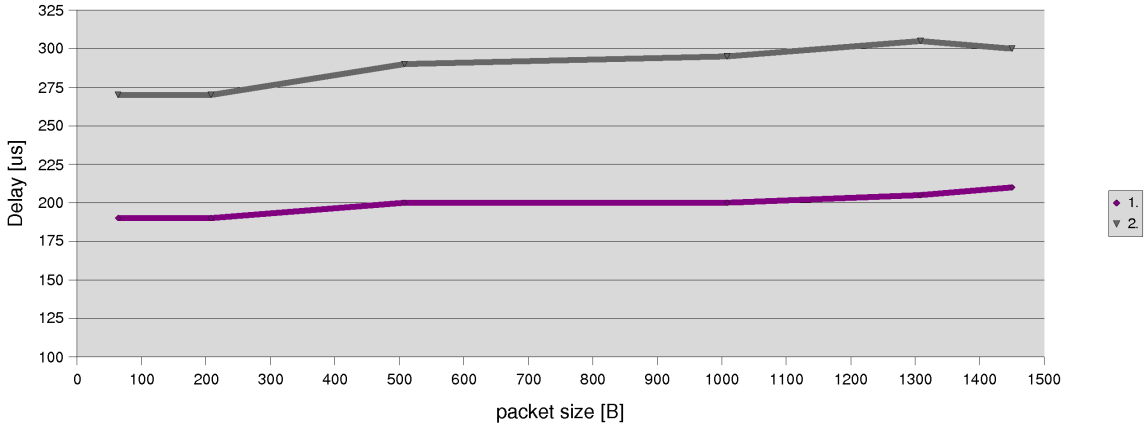

Figure 3.10: ICMP echo request and reply. Delay induced by the NAT and router's network stacks.

3.3.2 NAT Throughput Benchmarks Introduction

A standard benchmark for access routers is to measure the throughputs for UDP and TCP. This includes the upload from the local network to the global network as well as the download from the global network to the local network.
For TCP, the maximum throughput was measured since TCP supports flow control and congestion control and adapts the bandwidths of streams to the speed of the network.
However, for UDP, the stream qualities at certain stream bandwidths are interesing as UDP packets are sent out at a given rate regardless of the network speed. The UDP stream quality is the percentage of UDP packets that actually arrive at the destination. This gives an idea of how good the router handles streaming multimedia contents.

As a tool to run UDP and TCP benchmarks, `thrulay`[10] was used. Depending on wether upload or download was tested, a `thrulay` server was run on the local machine or on the global machine. For download tests, the `thrulay` server port had to be forwarded by configuring the NAT component accordingly.
In UDP mode, the `thrulay` client sends a stream of a given bandwidth to the `thrulay` server. The server then calculates the UDP stream quality by dividing the number of packets actually arrived by the number of expected packets.
In TCP mode, the `thrulay` client sends test data as a TCP stream to the `thrulay` server. At the end, the average throughput of the stream is calculated.

3.3.3 TCP Throughput Benchmark

This benchmark measures the maximum achievable TCP throughput for upload and download by using the router as a default gateway. For the download benchmarks, the router had to be configured to forward the port used by the `thrulay` server. The throughputs were measured for different packet sizes. The command used to start the `thrulay` client was "`thrulay -l[packet size] [host]`".

TCP throughput, absolute values, Router Version 0.98, (see Figure 3.11)

1. Router Version 0.98, Maximum TCP throughput download.
2. Router Version 0.98, Maximum TCP throughput upload.

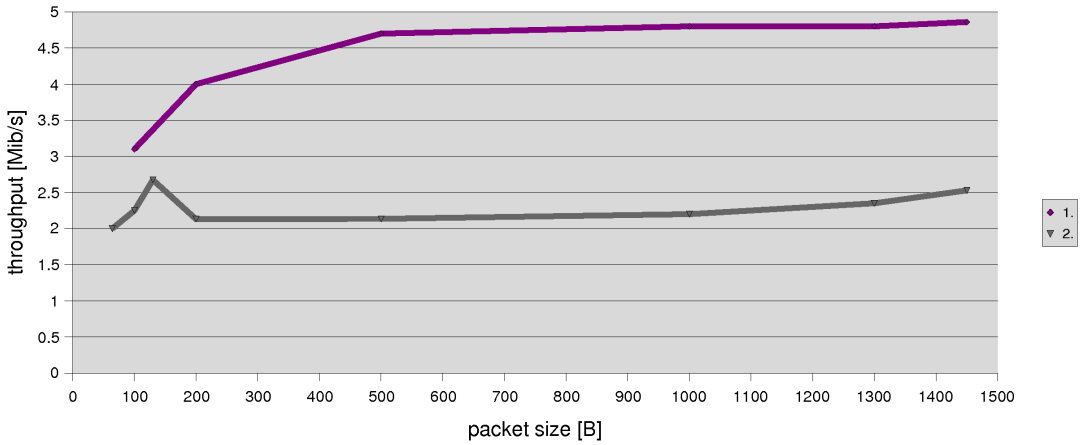

Figure 3.11: Maximum TCP throughputs for upload and download (router version 0.98).

CPU loads during TCP throughput test, Router Version 0.98, (see Figure 3.12)

1. Router Version 0.98, CPU usage download.
2. Router Version 0.98, CPU usage upload.

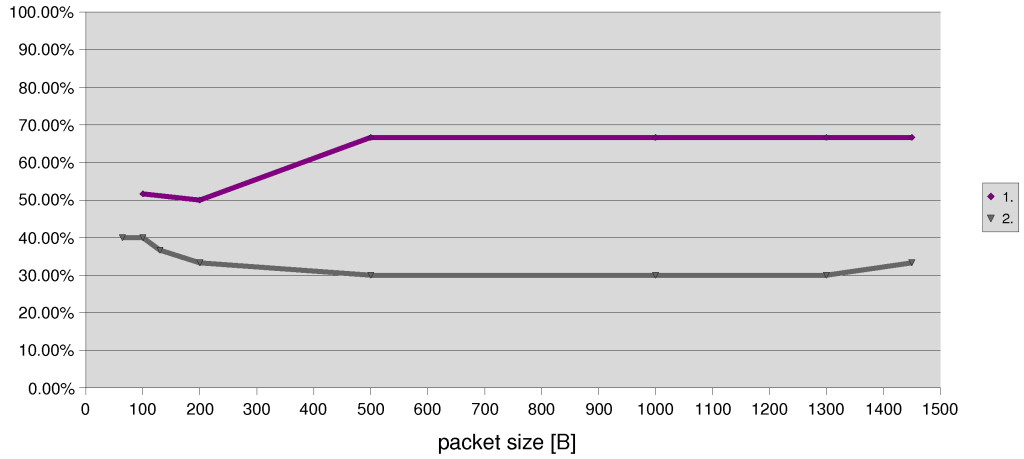

Figure 3.12: CPU loads during the TCP throughput test (router version 0.98).

As Figures 3.12 and 3.11 show, the throughput for download is higher than the throughput for upload. Both upload and download throughputs are smaller than expected. Theoretically, the throughputs should be closer to the maximum bandwidth that the NICs support, which is 100Mib/s. As Figure 3.12 shows, the bottleneck of the system is not the CPU as the CPU load never gets close to 100%.

3.3.4 UDP Throughput Quality Benchmark

This benchmark measures the UDP upstream and UDP downstream quality (rate of packets that arrived at the destination machine) when using the router as a default gateway (using 1450B UDP packets). For the downstream benchmark, the port used by the `thrulay` server had to be forwarded. The command used to start the `thrulay` client was `"thrulay -l[packet`

`size] -u[bandwidth] [host]"`.

UDP upstream and downstream quality, Router Version 0.98, (see Figure 3.13)

1. Router Version 0.98, Upstream quality.
2. Router Version 0.98, Downstream quality.

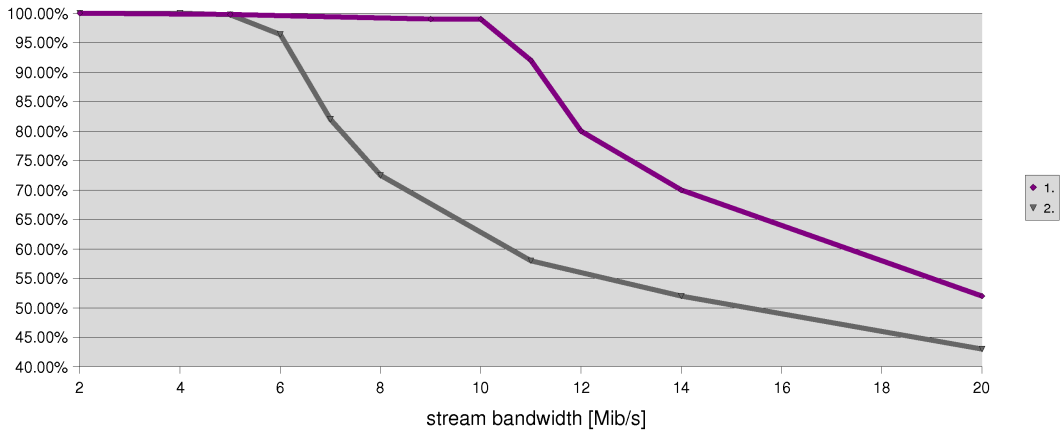

Figure 3.13: UDP stream quality (router version 0.98).

CPU loads during UDP stream quality benchmark, Router Version 0.98, (see Figure 3.14)

1. Router Version 0.98, CPU load upstream.
2. Router Version 0.98, CPU load downstream.

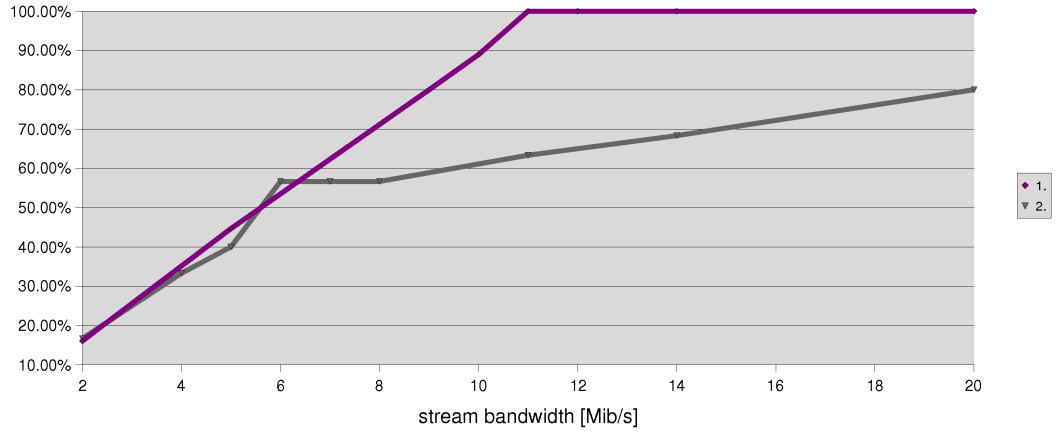

Figure 3.14: CPU loads during UDP stream quality benchmark (router version 0.98).

The drop of the UDP upstream quality (Figure 3.13, data series 1) at around 10Mib/s is most likely related to the CPU load reaching 100% at the same time (Figure 3.14 , data series 1). However, the UDP downstream quality drop at around 6 Mib/s (Figure 3.13, data series 2) is not related to the CPU load as the quality already gets worse at a CPU load of 55% (Figure 3.14, data series 2).

3.3.5 UDP Throughput Quality Benchmark: Overhead of Components in Different Protection Domains

The following plots compare the UDP stream qualities and CPU loads of the final router design and the router design where all components are in different protection domains and use Igua-

naRPC connectors (using 1450B UDP packets). This gives a good overview of the CPU overhead caused by having different protection domains and using IguanaRPC. The same benchmark as in Section 3.3.4 was carried out measuring the UDP upstream and downstream qualities as well as the CPU load during the tests.

UDP upstream quality, Router Version 0.98, comparison with router design where all components are in different Iguana protection domains and only use IguanaRPC connectors (see Figure 3.15).

1. Router Version 0.98, different protection domain, IguanaRPC, UDP stream quality upstream.
2. Router Version 0.98, UDP stream quality upstream.

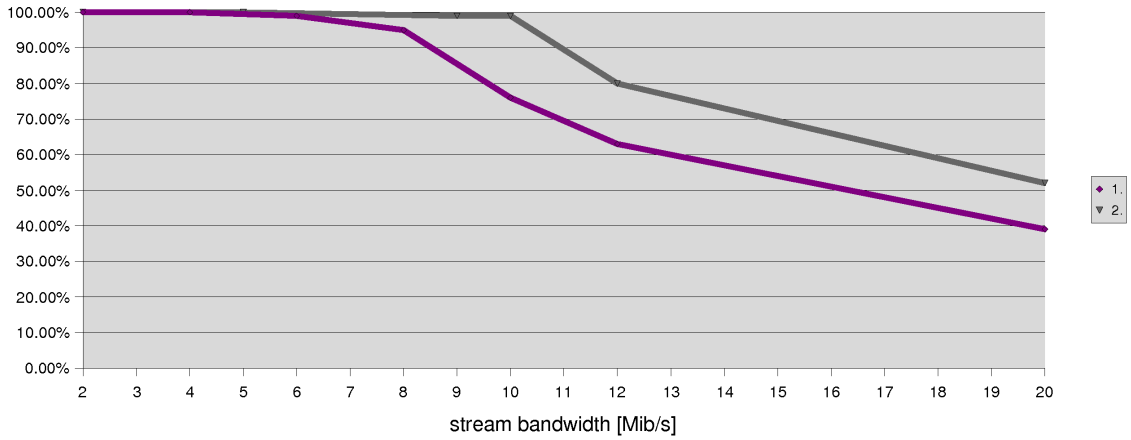

Figure 3.15: UDP upstream quality (router version 0.98 compared to design with components in different protection domain and only IguanaRPC used).

CPU load during UDP upstream quality measurements in Figure 3.15, Router Version 0.98, comparison with router design where all components are in different Iguana protection domains and only use IguanaRPC connectors (see Figure 3.16).

1. Router Version 0.98, different protection domain, IguanaRPC, CPU load upstream.
2. Router Version 0.98, CPU load upstream.

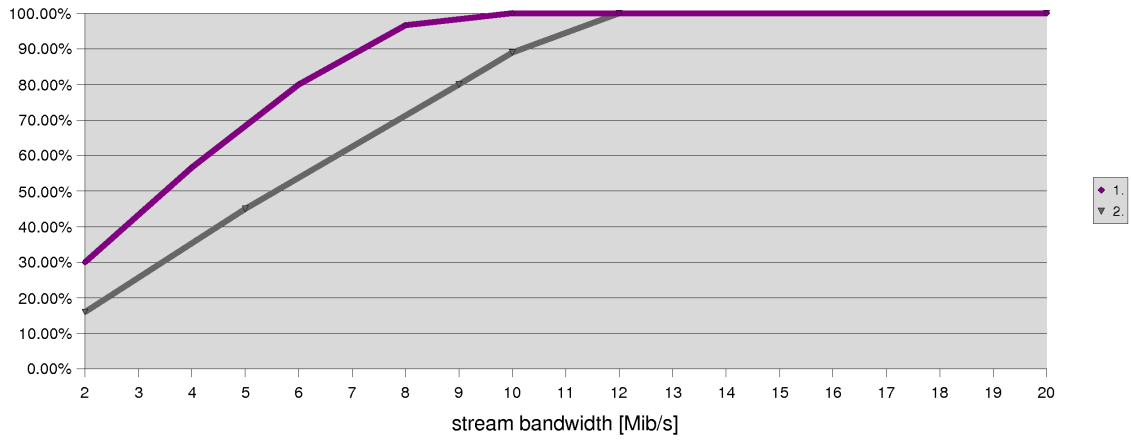

Figure 3.16: CPU load during UDP upstream quality measurements in Figure 3.15.

Figure 3.16 shows that the additional CPU load caused by the overhead of the inter protection domain IguanaRPC connectors is about 15% to 20% for UDP upstreams (Figure 3.16, data series 1 and 2). The upstream throughput penalty caused by this overhead is also approximately 20% (Figure 3.15 , data series 1 and 2).

UDP downstream quality, Router Version 0.98, comparison with router design where all components are in different Iguana protection domains and only use IguanaRPC connectors (see Figure 3.17).

1. Router Version 0.98, different protection domain, IguanaRPC, UDP stream quality downstream.
2. Router Version 0.98, UDP stream quality downstream.

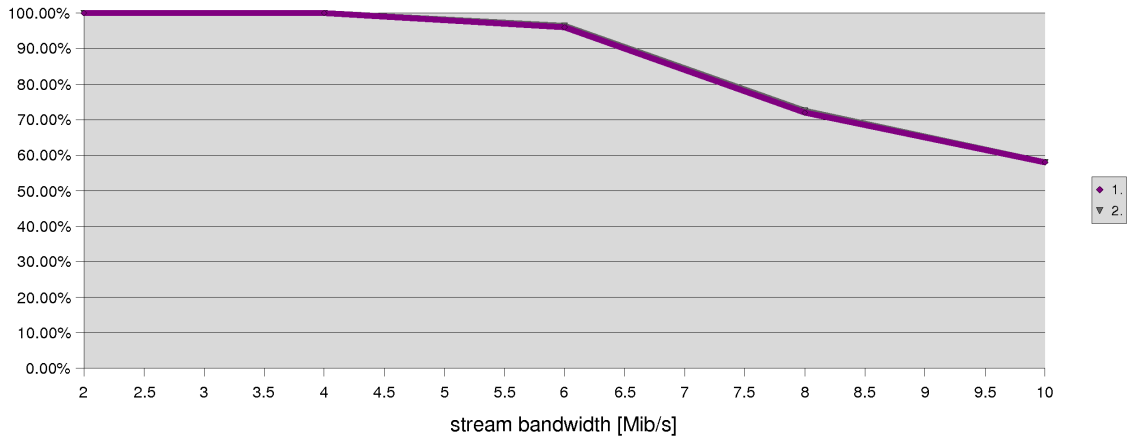

Figure 3.17: UDP downstream quality (router version 0.98 compared to design with components in different protection domain and only IguanaRPC used).

CPU load during UDP downstream quality measurements in Figure 3.17, Router Version 0.98, comparison with router design where all components are in different Iguana protection domains and only use IguanaRPC connectors (see Figure 3.18).

1. Router Version 0.98, different protection domain, IguanaRPC, CPU load downstream.
2. Router Version 0.98, CPU load downstream.

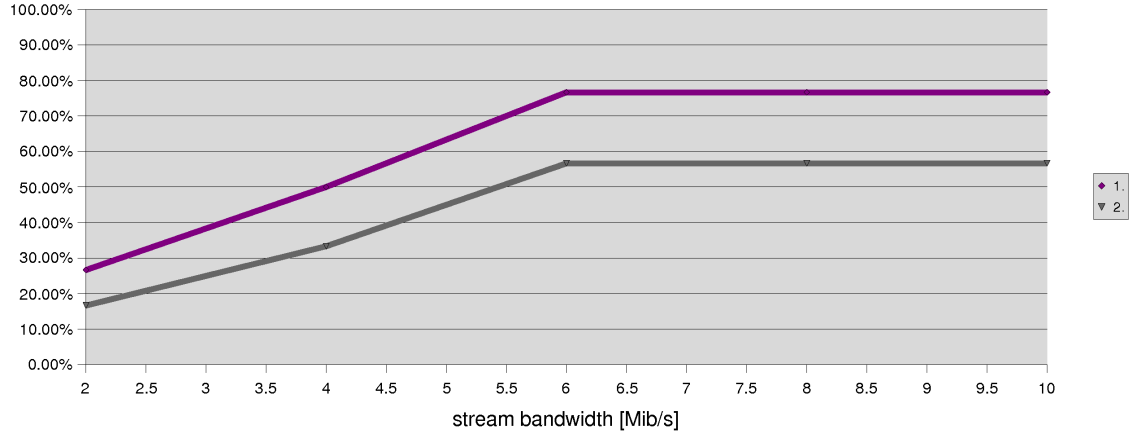

Figure 3.18: CPU load during UDP downstream quality measurements in Figure 3.17.

In Figure 3.17, both router designs have exactly the same UDP downstream quality for different stream bandwidths. The CPU load of the design where the components reside in different protection domains is about 20% higher than the CPU load of the router version 0.98 (Figure 3.18, data series 1 and 2). However, as the CPU load of both designs never reaches 100%, the UDP downstream quality stays the same even if there is a CPU load penalty to pay for the inter protection domain IguanaRPC connections.

3.4 Discussion of the Performance Measurements

3.4.1 Overhead of the Modular Network Stack

The results in Section 3.2 show that, depending on the configured security level, the delay induced by the overhead of the component architecture is less than 5% compared to a monolithic lwIP network stack.

However, the delay of the Linux network stack is, depending on the packet size, 5% to 30% lower than the delay measured with the most optimized router version 0.96. As the measurements in Figure 3.7 show, the delay measured by the stack's components from the network layer upwards is nearly independent of the packet size for router version 0.96 and is constant regardless of the packet size. This means, that 25% of this 30% difference between the Linux network stack and the router's network stack for large packets are caused by the network driver component. This assertion is backed by the fact that it takes up to 500ms to copy a 1450B network packet from the main memory to the NIC's memory as Figure 3.19 shows. Further, the delay measured in the monolithic lwIP stack, that uses the same driver as the router, is nearly the same as the one measured in the modularized network stack.

As Figure 3.4 shows, the presented network router does not run with 100% CPU load for all of the packet sizes while Linux on Gumstix takes advantage of more CPU time. These results show that the CPU in the router is not the bottleneck and that the packets are delayed somewhere else in the system. The router would still have CPU time available to process packets.

These facts lead to the conclusion that the network driver or the NIC hardware is responsible for a major part of the larger delay caused by the router's network stack compared to the delay induced by the Linux network stack. The problems that might cause these delays in the network driver or the hardware are discussed in Section 3.4.4.

3.4.2 Network Address Translation Performance

The delay that an ICMP network packet experiences while in transit between the local network driver and the global network driver is, as Figure 3.10 shows, between 200us for upload and 300us for download excluding NIC related instructions in the driver component. The return trip time inside the router therefore is about 500us and nearly independent of the ICMP packet size. However, the ICMP echo delay measured by a remote machine is between 0.7ms and 4ms for router version 0.98 as Figure 3.8 states; it depends significantly on the packet size. The CPU load is only about 70% at the same time. This means, the router has unused CPU capacity (see Figure 3.9) that could actually be spent to copy packets by PIO from the NIC or to translate between the global and the local networks.

Similar to the delay discussed in Section 3.4.1, the overall delay mainly seems to be caused by the network driver or the NIC hardware.

For a router that has all of its forwarding plane's components in different protection domains, the delay increases by nearly 1ms (see Figure 3.8) and the CPU load is also significantly higher as Figure 3.9 shows. If the components are in the same protection domain but still communicate via IPC, there is also a CPU load and delay penalty to pay. This is an example of the performance and security trade-off that has to be considered during the design of such a system.

Regarding the throughput benchmarks for UDP and TCP, the results are lower than expected. They should be closer to the 100Mib/s supported by the NIC. There should be enough CPU power as, except for the UDP upload benchmark in Figure 3.14, the CPU load never reaches 100%.

The UDP upstream speed in Figure 3.13 is related to the CPU load reaching 100% at the same time and would be higher if the CPU was faster or the network driver supported DMA. The CPU spends all of its time copying network packets by PIO from/to the NICs and would be available for the actual NAT process if a DMA controller executed this task in the background. However, for UDP downstream, the quality already starts to become worse when the CPU usage is only at about 60% (see 3.14). In this benchmark, the CPU load should theoretically go up to 100%. However, the packets never make their way up the network stack in the router but instead are dropped by the NIC because of reception overrun exceptions. Again, the network driver or the NIC seem to be the reason why the UDP downstream quality already starts to get worse at 6Mib/s. The task of finding the problem that causes the difference between the upload and the download stream quality were left for future work.

For TCP, the achieved maximum throughputs can be seen in Figure 3.11. Again, the throughputs are lower than expected as they should be closer to the 100Mib/s that the NICs support. It is not clear why the upload throughput is higher than the download throughput. Similar to the other benchmarks, the CPU load of the router never reaches 100% but stays as low as 30% for upload and 60% for download (see Figure 3.12). Theoretically, the CPU load should be close to 100%, especially as the network drivers only support PIO. This means, the CPU should be busy reading and writing to the NICs.

All in all, the performance of the router has potential for improvement. This mainly refers to the network driver components as most of the achievable performance actually gets lost before a network packet enters the router's logic. However, at the time this thesis was written, the router's performance was good enough to support the upload and download speeds provided by most of the ADSL and Cable Internet plans. For example, the fastest ADSL internet plan of the Swiss provider Bluewin included 0.5Mib/s upload throughput and 5Mib/s download throughput.

3.4.3 Trade-off Between Speed and Security

The benchmarks presented in this chapter reveal that there is a trade-off between performance and security in the router. If connections and patterns are used that are faster, they are usually less secure as the components need to share memory between each other. They have to provide access to the memory areas where the provided functions reside if a DirectCall connection is used. Components also need to share memory for the allocation of network packets if zero copy packet forwarding has to be supported. CAmkES provides ways to easily configure the security/performance trade-off from within the ADL file of an application.

In Figure 3.2, one can see that the delay caused by the componentised network stack increases when `DirectCall` connectors are replaced with `IguanaRPC` connectors, network packets have to be copied from one component to the other or queues are added between the different components. At the same time, the CPU usage increases in the system as Figure 3.9 shows. This is the price to pay for increased security.

Taking this into consideration, the final router design as presented in this report is divided into two planes. Components that reside in the forwarding plane only use the fastest connections but reside in one protection domain and therefore are not protected from each other as a result. The forwarding plane is protected from the application plane, but slower connections have to be used between the application plane and the forwarding plane. This represents a compromise between security and performance. The price to pay for this security feature is relatively small compared to the increased security, as this chapter shows.

3.4.4 Driver Components as Bottlenecks

As the discussion in Sections 3.4.1 and 3.4.2 shows, the driver components are a main bottleneck in the router implementation.
Some time during this thesis work was spent on tracking down the bottleneck in the driver implementation. However, as the focus of the thesis is not on low level I/O and low level driver implementations, the task of optimizing the drivers and solving I/O related problems was left for future work.

The communication of the router's drivers with the NICs mainly involves copying Ethernet frames over the bus between the CPU and the NIC's memory. Measurements of the time it takes to copy a frame out of the NIC's memory or into the NIC's memory revealed the copy durations showed in Figure 3.19. To measure these delays, the PXA255's timer register's value was stored before the instruction to read or write to the bus and the difference with the register's value after the bus transactions was calculated. No frames were forwarded to other components in the router. These delays are much higher than expected. For a frame of the size 1450 Bytes, the r/w delay has to be at most approximately 0.116ms to support a NIC speed of 100Mib/s in PIO mode. The measured delays are about 4 to 5 times higher than this expected delay. Taking the measured delays into consideration, the current driver only supports a maximum Ethernet frame arrival rate of about 2500 frames/s (frame size 1450B) if other interrupt processing related delays in the driver are ignored. This leads to a theoretical maximum reception throughput of only approximately 29Mib/s. Taking other delays in the interrupt delivery and handling process into consideration, this maximum throughput in reality is even lower.

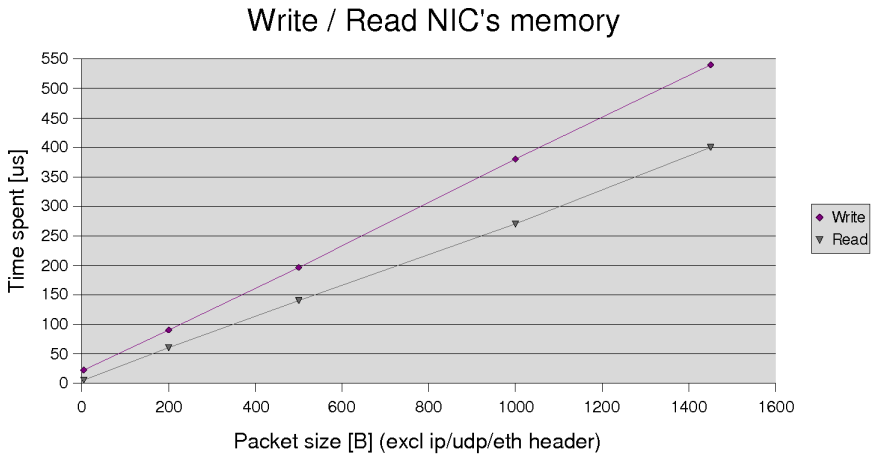

Figure 3.19: Duration in [us] of read/write to NIC's memory. The PXA255's timer register was used for the measurements (resolution approx. 0.3 [us]). The driver version was `andri.toggenburger@nicta.com.au-devel/driver-smc91x-devel-0.2`. The measurements only included the instructions that are used to read (`SMC_PULL_DATA`) or write (`SMC_PUSH_DATA`) to the bus. The variance of the data is 10 us^2. Used Hardware: Gumstix Connex 400xm, ARM XScale/PXA255, 400MHz, 64MB RAM, netDUO-mmc extension board (100 Mib/s).

As there are many RX overruns (a frame arrives at the NIC but the NIC is not ready to process it and consequently drops the frame) registered by the network drivers during the different benchmarks, the actual rates of RX overruns and processed Ethernet frames in the NICs were measured for UDP streams of different bandwidths (see Figure 3.20). The corresponding counters in the driver implementation were used to obtain this data. The Ethernet frames were copied out of the NIC's memory but were not forwarded to other components in the router.

Figure 3.20

1. Rate of processed packets.

2. Rate of RX overflows.

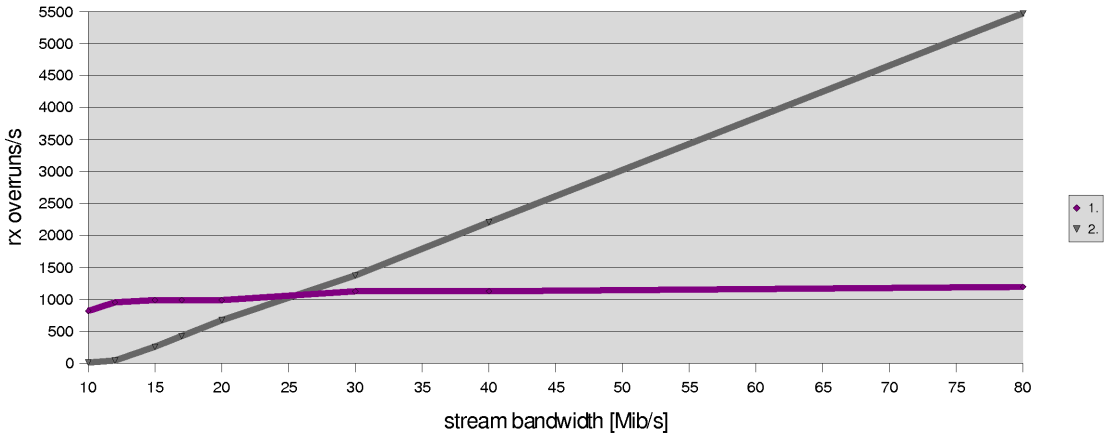

Figure 3.20: RX overruns and actually processed ethernet frames (frame size 1500B). The driver version was `andri.toggenburger@nicta.com.au-devel/driver-smc91x-devel-0.2`. The received Ethernet frames were not forwarded to other components in the router. Used Hardware: Gumstix Connex 400xm, ARM XScale/PXA255, 400MHz, 64MB RAM, netDUO-mmc extension board (100 Mib/s/s).

Figure 3.20 shows, that there is a maximum rate of frames a NIC can receive successfully. This rate is around 1100 frames per second (for 1500B frames). If the packet arrival rate exceeds this maximum rate, all other frames generate an RX overrun and are consequently dropped by the hardware. The measured maximum arrival rate corresponds with a throughput of approximately 14Mib/s. Note that during these measurements, the NIC only receives frames but does not send any out to the network.
Not a single RX overrun was registered when the instructions to actually copy a frame out of the NIC's memory were commented out.

Figures 3.19 and 3.20 suggest that the communication between the CPU and the NIC is a major bottleneck in the driver's implementation. Packets can not be copied fast enough from the NIC's memory to the main memory in order to support high bandwidths. As ethernet frames arrive at a higher rate from the network than the rate at which they can be copied out of the NIC's memory, the circular buffer in the NIC fills up and subsequent frames generate RX overruns, which lead to packet losses.

The task of solving this I/O related problem was, as stated before, left for future work. Some possible causes might include:

- Problem with thread priorities of driver.
- No DMA support, so CPU is still busy copying a frame while it actually should process the next interrupt.
- Bugs introduced when the driver was ported from Linux to L4/Iguana.
- Memory alignment problems or endianness problems.
- Wrong configuration of full-duplex, half-duplex or 10baseT/100baseT in NIC's physical layer.
- Wrong settings for collision avoidance and detection on physical layer.
- other hardware misconfigurations (see the SMC FAQ[9]).

The data sheet[8] and the FAQ[9] of the SMC network controller used provide more information about possible causes of bad NIC performance.

3.4.5 Comparison with the Stated Performance Requirements

The performance requirements in Section 2.1 can be fulfilled by the final router design. Its supported bandwidths are high enough to support the maximum speeds provided by most broadband plans at the time this report was written. The delays induced by the router are also small enough (between 1ms and 4ms for a `ping` with a packet size of 64B and 1450B respectively) that the router can be used as an access device to the internet instead of a Linux machine configured as a NAT router for example.
Furthermore, the router's performance has potential to be improved by solving the I/O related problems described in Section 3.4.4.

Chapter 4

CAmkES as a Component Architecture for the Development of a Modular Router

As CAmkES is the underlying architecture of the presented router, it shaped the design and implementation of the router in a major way. However, since the router was the first major application, missing features were implemented in CAmkES and insights into the development and implementation process of a project using CAmkES were gained.

4.1 Adding Support for User Defined Connectors

Each connection declaration in a CAmkES ADL file has an associated connector. The connector provides the communication logic and defines the run-time semantics of the communication channel defined in the connection declaration. The "user defined connectors" feature in CAmkES lets a designer implement their own logic, which is executed when two components communicate together via a corresponding connection. While it is easy to change the connector belonging to a certain connection by simply editing the connection declaration in the CAmkES ADL file, the implementation of the support for user defined connectors was missing in CAmkES when this thesis started and was added as part of it.

4.1.1 Motivation

During the design and implementation of the router, two major bottlenecks caused by the CAmkES architecture were detected. One bottleneck was the missing support for direct method invocations between the different components. Every method invocation involved the overhead of exchanging messages via IPC calls between the components as there was only the standard connector `IguanaRPC` implemented in CAmkES.

Another problem was the missing support for data sharing between more than two components. Only pairs of components could share a common memory area between each other. This meant that, for all processes where packets had to be transferred between 3 or more components, the payload of the packet had to be copied from one shared memory area to the other by the components. As various measurements showed, the overhead caused by the method invocations via IPC and the memory copies of whole packets had a major impact on the router's performance.

As part of this thesis, a connector based on method invocation by using function pointers was designed and implemented (DirectCall connector). To allow for data sharing between arbitrary numbers of components, the existing IguanaSharedData dataport connector was generalized.

During this work, insights into the architecture generating the code for the CAmkES component architecture were gained. This knowledge was used to design and implement the support for "user defined connectors" in CAmkES as described in the following sections.

4.1.2 Benefits

User defined connectors provide the following benefits as compared to a set of standard connectors:

- A designer can implement optimized connectors for a certain scenario.
- Connectors can be instrumented to e.g. make performance measurements in a system. They also can be instrumented to create log files of the performed RPC method invocations, emitted events and accessed memory locations.
- Queues, stacks and other data structures can be implemented as connectors rather than as components. This simplifies the ADL declaration of complex applications and makes it more readable.
- Extensions could be added to existing connectors. E.g. a connector could be extended to encrypt and decrypt network packets transparently between the components that it connects.

4.1.3 Design of the Support for User Defined Connectors

CAmkES uses a set of Python scripts and helper programs to generate the code for each component out of the ADL and IDL declarations and the component's implementation. An important tool is Magpie[6], which is an interface compiler for the L4 kernel. It supports template driven code generation.

First, Magpie creates an abstract syntax tree using the ADL and IDL declarations of the application. In a second step, the CAmkES Python scripts traverse this tree and create data structures for all of the components and connections in the application. These data structures are later used to generate code for each component by using a set of template files.

The CAmkES Python scripts that existed contained a lot of non generic and hard coded, connector specific, code. Also, the template files that were used to generate the code for the different connections and the components were not completely separated from each other. The templates used to generate code for the connections also contained code for the components and vice versa. However, connector specific templates have to be independent of other templates in the system so they can be added or removed like extension modules to CAmkES

Another missing feature of the existing implementation was its support for generic initialization code for connectors (and connections) of different types. Taking this into consideration, an architecture supporting generic initialization code, and that completely separates connector specific templates from the other connectors' and components' templates had to be developed.

The designed and implemented architecture is generic enough to support user defined connectors between 2 or more components. The connectors can implement their own connector and connection initialization protocols. The architecture is also symmetric, for example, it does not distinguish between users and providers in the RPC interface case. All it does is to pass the position of a component in the connection declaration to the corresponding connection's initialization function (e.g. first place, second place).

Code Generation Architecture

Figure 4.1 shows the code generation process in CAmkES. First, Magpie generates an abstract syntax tree out of the application's ADL and IDL files. This syntax tree then is processed by a set

of CAmkES python scripts that create data structures for the components and their connections.

In a further step, for each connection belonging to a certain component, server stub files and client stub files are generated. The code that is generated for a connection depends on the connector that is declared for this specific connection. Each connector implementation has its own set of template files that are automatically used by the CAmkES python scripts during the code generation.
The client and server stub code is generated by the following scripts (see Figure 4.1):

- RPC connections: The stub files are generated by the script `stub_runner.py`.
- Event connections: The stub files are generated by the script `event_runner.py`.
- Dataport connections: The stub files are generated by the script `dataport_runner.py`.

In order to initialise the connections during run-time, initialisation code for each connector a component uses is generated by the script `connector_runner.py` out of a set of connector specific templates. This code then sets up the environment (e.g. starts server threads waiting for IPC method invocations) of the connector and initialises all of the connections that use this connector. This consists of initialising the server and client stubs belonging to the specific connections. The used template files have to be named `<Connector_name>_connector.c/h` (see Figure 4.1).

There is a set of functions that has to be implemented for each connector in its templates. During the initialization of a component and its connections, these functions are automatically called by the component code generated by CAmkES in order to initialise the connector and all connections that use it:

- `<Connector_name>_connector_init(void *p, int index)`: Initialize connector wide things (e.g. start a thread listening for IPC requests or arriving events). Parameter `p` contains a generic environment while parameter `index` refers to the position of this component in the connection declaration.
- `<Connector_name>_instance_init(void *p, int index)`: Initialize connector related things in this component instance (e.g. call `__init` functions for each interface provided). Parameter `p` contains a generic environment while parameter `index` refers to the position of this component in the connection declaration.
- `<Connector_name>_connection_init_active(int con_id,L4_ThreadId_t con_init_loop_tid,objref_t data,int index)`: Implement connection initialization logic for the component that initiates this connection's initialization process (is the client in this connection's initialization process). Parameter `con_id` contains the global connection ID of the current connection. Parameter `con_init_loop_tid` contains the thread ID of the other component's (server) connection initialization thread. Parameter `data` contains a reference to the data section of this component (needs to be passed to other component to create Iguana sessions for memory sharing). Parameter `index` refers to the position of this component in the connection declaration.
- `<Connector_name>_connection_init_passive(int con_id,L4_ThreadId_t con_init_runner_tid,objref_t data,L4_Msg_t *msg,int index)`: Implement connection initialization logic for the component that is the server for this connection's initialization process. Parameter `con_id` contains the global connection ID of the current connection. Parameter `con_init_runner_tid` contains the thread ID of the other component's (client) connection initialization thread. Parameter `data` contains a reference to the data section of this component (needs to be passed to other component to create Iguana sessions for memory sharing). Parameter `index` refers to the position of this component in the connection declaration.

The code for the client and server stubs as well as the code for the connector and connection initialisation is finally placed in the directories of the corresponding components. After the code generation is finished, the generated C code can be compiled to build a boot image.

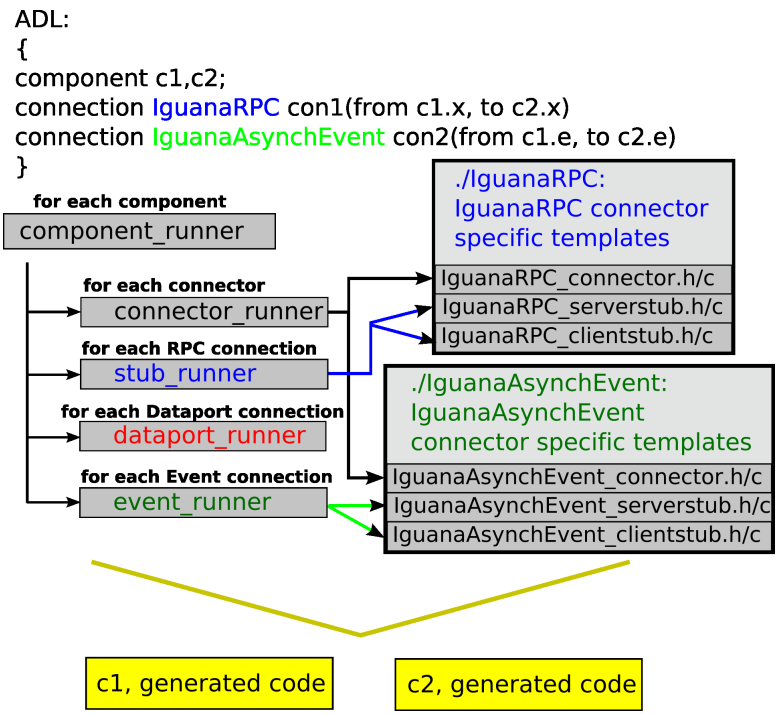

Figure 4.1: Generation of code using CAmkES user defined connectors. At the end, there is a directory for each component containing its generated code.

Connector and Connection Run-time Initialization

As an example of how components initialize their connections and connectors during run-time, the run-time initialization interaction between two components sharing an IguanaRPC connection together is depicted in the Figure 4.2. This also includes the startup of the components themselves.
This is an explanation of the different steps in Figure 4.2.

1. The CAmkES application loader thread starts up all of the components in the system as L4/Iguana servers. It also starts each component's `init_loop_thread` and `init_runner_thread`. The `init_loop_thread` is the server for connection initialization protocols while the `init_runner_thread` is the client for connection initialization protocols.

2. The `init_loop_threads` call the initialization function `<Connector_name>_connector_init` for each connector a component uses. In the example in Figure 4.2, it starts an IguanaRPC service loop thread that handles IPC method invocations. At the same time, the `<Connector_name>_instance_init` function is also executed to initialize the provided RPC interfaces (only the ones that use IguanaRPC as a connector).

3. The application loader sends information about each connection a component is part of to the corresponding component's `init_runner_thread`. This information contains the global connection ID of a given connection. Another data field is the `init_loop_thread` ID of the other component sharing this connection. Furthermore, the connector type that is used for the connection is also sent.

4. Using the received connection data, the `init_runner_threads` start to call the `<Connector_name>_connection_init_active` function for all of the connections the corresponding component has. This function implements the client part of the connection initialization protocol and IPC messages are exchanged with the `init_loop_threads` in the other component sharing a certain connection. The server part of the connection initialization protocol is implemented in the function `<Connector_name>_connection_init_active` that is called by the server's `init_loop_thread`. In the IguanaRPC example, the caller component receives the thread ID of the IguanaRPC service loop thread started in step 2. This is later used to make function invocations by using IPC messages.

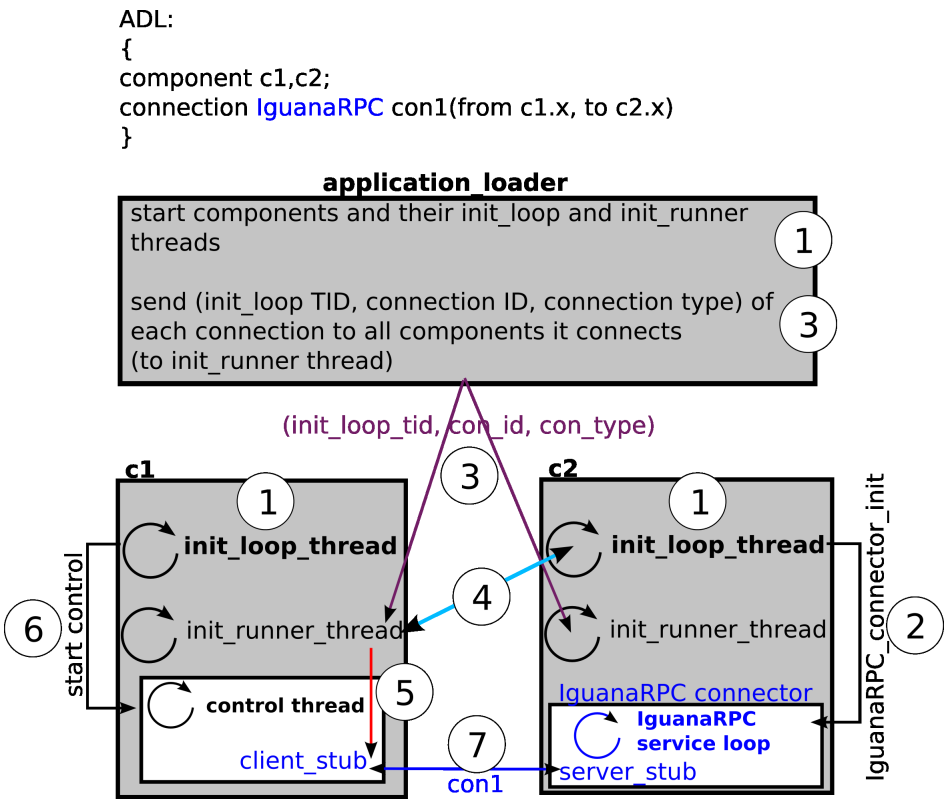

Figure 4.2: Startup, connector and connection initialization of a CAmkES application.

5. The data that was received from the `init_loop_thread` of the other component is stored in the client stub belonging to the just initialized connection. In this example, the thread ID of the IguanaRPC service loop of the other component is stored in the client stub. On the server side, the other component could store data in the server stub belonging to this connection if necessary.

6. After all components in the system are finished with initializing their connectors and connections, the application loader starts the control threads of the active components running. At this time, the whole system is up and running.

7. Now, as the system is up and running, component `c1` can make method invocations by using the functions provided by the client stub. The logic that is executed during the communication with component `c2` is the logic as defined in the IguanaRPC user defined connector templates.

4.1.4 Implementing a User Defined Connector

To implement a user defined connector, the template files mentioned in 4.1.3 need to be added to a directory named `<Connector_name>`. If a connector has to be made available for all CAmkES applications, this directory has to be in the standard CAmkES template directory. If only one application uses a particular connector, the connector's directory can be in the application's main directory. In order for the code generator to recognize the new connector, the new connector has to be declared in the file `./components/camkes/std_connectors.camkes`. While it would be better to declare the new connectors at a more appropriate place than the standard connector file, support for multiple connector definition files was left for future work.

4.2 Evaluation of Overhead Induced by CAmkES Connectors

4.2.1 Performance Comparison of 3 Different CAmkES Connectors

Given that the router's components are connected with IguanaRPC and DirectCall connectors, performance measurements for these connectiors were carried out. For connections of the type IguanaRPC, the two involved components do not need to be in the same Iguana protection domain. However, if the components are not in the same protection domain, another performance penalty has to be paid because of the more expensive cross-protection domain IPC.

Figure 4.3 shows a comparison of different connection types. The delay of a method invocation including 0 to 16 int parameters was measured using the PXA255 timer register on the hardware platform described in Chapter 3. The called function's implementation does nothing but add integers. The delay caused by the IguanaRPC connection is very high if the components are in different protection domains (data series 1) but is lower if the components are in the same protection domain (data series 2). The DirectCall connection is the fastest connection as it uses function pointers for the method invocation and does not have to send the parameters and the results as an IPC message (data series 3). However, if the components are in the same protection domain, a possibly misbehaving component can cause another component to crash by writing to its sensitive memory areas.

1. IguanaRPC connector, components in different protection domains.
2. IguanaRPC connector, components in the same protection domain.
3. DirectCall connector.

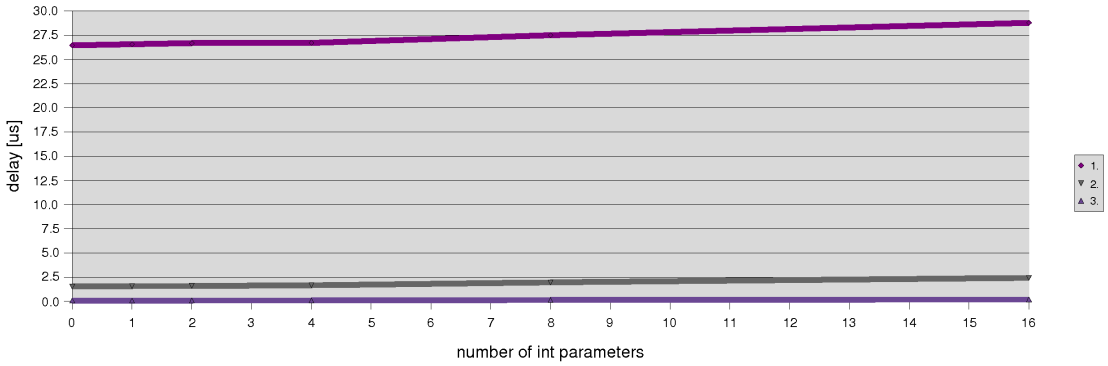

Figure 4.3: Comparison of the method invocation delay caused by different connectors.

4.2.2 Examination of the Overhead of the IguanaRPC Connector

The overhead induced by an IguanaRPC connector was measured using the instrumented `IguanaRPCMeasure` connector. This connector measures the duration of different stages in the process of invoking a function in another component by using the PXA255's timer register. As before, the hardware platform described in Chapter 3 was used to perform the measurements. The called function's implementation does nothing but add integers. The measurements were performed for 0 to 16 `int` parameters that were passed as function arguments.

Unfortunately, the resolution of the timer register is too low to measure the execution time of a short sequence of instruction like the marshaling or the unmarshaling in IguanaRPC. However, to get approximate values, the instructions in the interesting code sections to be measured were executed 10000 times in a row. This duration was divided by 10000 to get an approximation of the actual duration. Unfortunately, this is only a very rough approximation as cache misses and other system related delays that occur during an `IguanaRPC` invocation are not measured

this way. The measurements were carried out on the client side and on the server side of the interface invocation.
On the client side, there are three different stages in the communication process:

- Marshal function parameters
- Make IPC call and wait for reply from server
- Unmarshal the results

The duration of each of these stages can be seen in Figure 4.4. The duration of unmarshaling the result can be neglected compared to the durations of the other stages. The marshaling duration depends on the number of parameters that the function passes. The IPC call and the delay caused by the server cause most of the measured overall delay of an IguanaRPC method invocation.

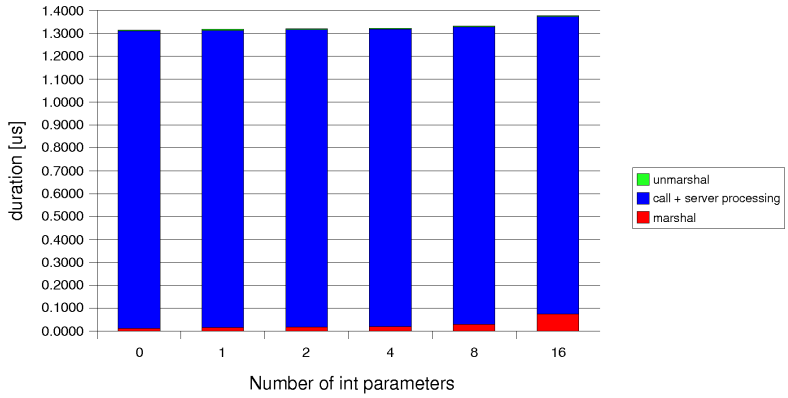

Figure 4.4: Durations of different stages in an `IguanaRPC` method invocation (client side).

On the server side, there are also three different stages in the communication process:

- Handle the IPC request message of the client (includes unmarshaling of the parameters)
- Dispatching of the RPC and executing the called function's implementation
- Send the reply by IPC to the client (includes marshaling of the result)

The duration of each of these stages can be seen in Figure 4.5. The majority of the time on the server side is spend to send the IPC reply message containing the function results back to the client. Handling the IPC request and dispatching the RPC take approximately the same time and depend on the number of received function parameters.

The results presented in this section show that the marshaling and unmarshaling of the function parameters and the result is small compared to the time it takes to make IPC calls and service arriving IPC messages. The marshaling and unmarshaling of the result is optimized as the result data structure is directly allocated in the IPC message buffer used to send the reply to the client. The function parameters, on the other hand are marshaled/unmarshaled in a more complicated way. As the time it takes to communicate via IPC causes the major part of the delay, there is only a low potential for any further optimizations of the `IguanaRPC` connector.

4.3 Influence of CAmkES induced overhead on Router's Performance

As Chapter 3 shows, the impact of the CAmkES component architecture's overhead on the router's performance depends on the chosen connector types and communication patterns. Generally, the overhead of connectors that involve the sharing of memory (to share variables or

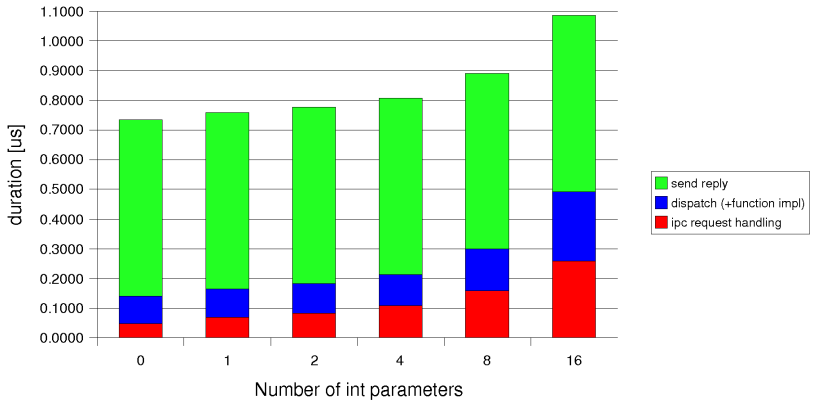

Figure 4.5: Durations of different stages in an `IguanaRPC` method invocation (server side).

to provide functionality) is smaller than the overhead of connectors using messages transported by IPC. However, the price to pay is lower security in the system as misbehaving components can trash the shared memory areas and cause other components to crash.

4.4 Assessment of the CAmkES Component Architecture

4.4.1 Positive and Negative Properties as Seen from a Developer's View

This reflects personal experience regarding the "look-and-feel" and the "ease of development" of using CAmkES as a component architecture.

Positive aspects include:

- Easy to implement first applications. During this thesis, the first test applications were up and running after a few days. The provided sample applications cover a wide range of CAmkES features.
- Easy to replace connectors with other connector implementations. This allows to configure the speed and security trade-off of an application in an easy way.
- User defined connectors allow the optimization of the component's communication, instrumentation of the communication logic and transparent addition of features like connection encryption in an easy way.
- Easy to replace different implementations of components with each other. For example, replace a driver component for a particular hardware with a driver component for a different hardware or replace different memory allocator implementations with each other to find the optimal one for a system.
- CAmkES code generation architecture is easy enough to be able to fix bugs in it or to add special or missing features required by an application (e.g. add interrupt handling support to it).
- Does not hide L4/Iguana but rather complements them. This makes it easy, for example, to encapsulate the implementation of a hardware driver into a CAmkES component.

Negative aspects include:

- CAmkES ADL files are hard to read as there are many connection declarations for more complex applications.
- CAmkES ADL files are hard to edit as there are no real error messages produced by the compiler if declarations are wrong. Combined with the fact that these files are hard to read, it takes a long time to track down an error in an ADL file.

- Poor support for complex data types. There is no support for types of undefined length (e.g. strings) at all as parameters for RPC calls.
- It can take several minutes to generate the code for a more complex application.
- No iterators for interfaces. In a component's declaration, each interface must have its own name. This makes it hard to read. Also, in the code using these interfaces, they have to be referred to by their names in a hard coded way.
- Not suitable to assemble applications out of fine grained components. CAmkES components are similar in size to servers in the microkernel context.

4.4.2 Generality of the Mechanisms Provided by CAmkES

In this section, the matching of the three other modular router architectures discussed in Section 2.7.5 (Scout, XORP and Click) to CAmkES components is investigated. This gives an insight into wheter CAmkES and its features are general enough to support different patterns of modular software development.

Scout

Scout *stages* and *modules* implementing different network protocols directly match to CAmkES components. Every CAmkES component would have to provide a classification function, so the path of a network packet through the router could be looked up before the packet actually enters the router's logic. The connections between the components could either be implemented as RPC method invocations by IPC or direct function calls. However, it is not straightforward to provide the guaranteed allocation of different system ressources to a certain packet's path through the router like in Scout. CAmkES is general enough to support the implementation of Scout. Wether a ressource allocation as in Scout is possible depends on the features of the underlying L4/Iguana.

XORP

XORP modules are UNIX processes behaving similar to servers in the microkernel context. Therefore, XORP modules directly match to CAmkES components connected by RPC or Event interfaces using IPC for message exchanges. By designing appropriate CAmkES connectors, it should also be possible to support XORP's transparent communication between components running on different machines. Similar to XORP, the components could either run in a single protection domain or in different protection domains. CAmkES would be an ideal platform to implement XORP.

Click

The Click elements do not really match to components in CAmkES as they are very fine grained. However, Click theoretically could be implemented as CAmkES components if necessary. Every Click element would be implemented as a single CAmkES component. An L4/Iguana thread would run over the task queue and call the different element's functions by using `DirectCall` connectors.
Such a system would contain a lot of components only implementing very limited logic. As the overhead of the generated code would be very large and the CAmkES ADL files would become unreadable because of the large amount of component and connection declarations, the CAmkES architecture is unsuitable to implement Click.

Results

As the previous discussion showed, CAmkES is general enough to theoretically support the implementation of all of the mentioned router architectures. However, it does not make sense to implement Click in CAmkES as Click elements are too fine grained compared to the more

coarse grained components that CAmkES is aimed at.

The supported interface types in CAmkES (RPC, Event, Dataport) and the possibility to implement user defined connectors to connect the components' interfaces make CAmkES very general and flexible.

4.4.3 Proposed Additional Features for CAmkES

The following is a list of features deemed missing during the router's development and that would improve CAmkES as a component architecture.

- There is no explicit support for interrupt handling available in CAmkES. It would make sense to add a new keyword to the ADL so components could declare their interest in receiving certain interrupts.
- There is no way to explicitely group a set of components together in a protection domain. The ADL should contain language constructs to explicitely group components in certain protection domains.
- A powerful additional feature for CAmkES would be to support hierarchies of connector. This would allow to define connectors as stacks of different other connectors or as bundles of other connectors.
- The support for data types of variable length (strings, arrays) as parameters in RPC interfaces is missing.
- Components can declare attributes in the ADL. A necessary feature would be to let connectors also declare attributes. These attributes could for example be used to set the thread priorities of the service loops of different `IguanaRPC` connector instances.
- The CAmkES ADL files are already hard to read for smaller applications. Therefore, a GUI implementation that allows to connect components together etc. and automatically generates an ADL file in the background would be beneficial to the development process of a CAmkES based application.
- Low overhead support for fine grained components (components that are similar to software libraries)
- Reflection for CAmkES, so the components can query information like their instance name and the connections they use. This would e.g. be useful for debugging purposes.

4.4.4 CAmkES to Build a Microkernel-Based Embedded System

Based on the experiences with CAmkES gained during the design and implementation of this project, it is believed that CAmkES will facilitate the development of microkernel-based embedded systems in a major way.
Examples of how CAmkES can be used to support the development of embedded systems:

- Easy assembly of whole microkernel-based (operating) systems out of pre-designed components. A component in this case would represent a server in the microkernel context. The connector type would be `IguanaRPC`.
- Easy assembly of servers in the microkernel-context based on pre-designed software modules. For example assembling a server that implements a network stack out of components that implement the different protocols. All of these components would be placed in one protection domain and be connected by `DirectCall`.
- Plug-and-play like assembly of experimental (operating) systems e.g., for analyzing and testing purposes.
- Easy to replace hardware specific code (e.g., a driver) with an implementation supporting other hardware or providing a software emulation of the hardware.

Chapter 5

Conclusions

This report presents the development of a ready-to-use modular router based on the CAmkES architecture for embedded systems. The proposed router architecture is extensible, flexible and especially tailored to embedded systems. The resulting software is the first CAmkES-based major application and runs on a real embedded hardware platform. CAmkES proved to be a very sophisticated architecture that supports an abundance of different patterns of how to build a modular system. The possibility in CAmkES to implement custom made logic for the connections between the system's components proved to be especially powerful. CAmkES will facilitate the development of microkernel-based embedded systems in a major way.

5.1 Self-Assessment

This section points out positive and negative aspects in the design and the implementation of the described router and the whole project.

Negative aspects include:

- The achieved throughputs of the router are low compared to the throughputs the NICs would support. The problem causing the low throughputs could be tracked down to the network drivers. However, the problem could not be solved in a reasonable time and therefore had to be left for future work. Another performance related problem is the missing DMA support in the system.
  In most of the measurements, the network drivers were the bottleneck. Therefore, it was hard to perform tests of the router's logic under heavy load as network packets simply were dropped by the NICs before entering the router's logic.

- More measurements using Linux for Gumstix would be necessary to compare delays and throughputs of the router's NAT functionality to another software architecture running on the same hardware. Unfortunately, it was not possible to set up the routing functionality in the Gumstix Linux distribution in a reasonable time. Many standard Linux programs like `route` are missing on the Linux for Gumstix and therefore have to be added manually. However, different approaches to compile NAT support into the Linux boot image for a Gumstix machine did not succeed.

- Regarding the modular network stack, only delay measurements were performed. It would also be important to measure the maximum throughput at which an application can send data out to the network and the maximum throughput at which an application can receive data.

- The presented router does not use a CAmkES feature that allows to group components together in a composite component. A composite component exports defined interfaces of its inner components and can be used like a standard component. Components of the network stack including their queues could have been grouped in a composite component to make it easier to understand the ADL definition of the whole application.

- The modularization of the lwIP stack could have been done in a more sophisticated and consistent way. At the moment, only the functions that are actually used by the router and its components are modularized. Furthermore, there would be architecturally better ways of modularizing lwIP compared to the current approach of actually replacing the code of all function invocations to upper or lower network layers by function pointer calls. The current way does not allow lwIP to be upgraded to a newer version, for example.
- A more generic architecture for the actual routing process could have been chosen. In the current design, the NAT component implements forwarding, packet editing and routing in one place. A more generic architecture would separate these three stages. One component would manage the forwarding of packets to an arbitrary number of NICs. During the forwarding, it would look up a route using an interface to a routing database. The entries in this database could either be configured manually or inserted dynamically by a component running a dynamic routing protocol. Before or after looking up a route, plugin components could be added. These plugins would allow, for example, NAT functionality to be implemented.

Positive aspects include:

- The implementation of the router is stable and the Gumstix machine running the router application has been successfully used to access the internet and its applications from a desktop machine by configuring the desktop machine's default gateway accordingly. The reliability of the router is a result of the decision to use already existing, well-tested code as often as possible.
- The router fulfills all of the requirements that were defined before the project started. It uses most of the features provided by CAmkES and runs on a real embedded hardware platform. It therefore is a good reference application for the CAmkES architecture.
- The design of the router distinguishes between trusted and untrusted components. Trusted components can be connected to other trusted components by using the fast `DirectCall` connector that was developed during this project. Untrusted components can be integrated into the system as components running in the application plane and communicating via IPC with other components. If an untrusted component misbehaves, it does not affect the rest of the system.
- Extensibility, flexibility and reusability are a key feature of the designed router architecture. The components provide simple and general interfaces. Therefore, components extending the router's functionality can be developed, implemented and inserted into the router in a short time. By choosing different connectors for the connections between the components, the performance/security trade-off of each connection can be configured in a flexible way. The existing network stack components, queue components and memory allocator components are general enough so they can be re-used in another application based on the CAmkES architecture.
- Performance measurements were carried out often and regularly as the project progressed. This allowed the comparison of the performance of different router versions between each other. As an interesting additional dimension to the measurements of the router's delays and throughputs, the CPU load in the system during the benchmarks was also measured. Finally, the overhead induced by the CAmkES architecture was examined by experimenting with different connectors and performing benchmarks of the resulting router systems and different stages in the communication process.
- The designed and implemented router was compared to other modular router architectures. This included pointing out similarities and differences between the different architectures as well as assessing the possibility to theoretically port components of another router architecture to CAmkES in order to be integrated with the current router.
- The experiences made during the design and implementation of a system with CAmkES led to an abundance of proposed features and patterns of how to build applications with CAmkES. During the project, new features were designed and implemented in CAmkES. This includes an architecture for user defined connectors, a `DirectCall` connector and an instrumented connector to measure the overhead of IPC method invocations.

5.2 Future Work

Interesting future work on this project could include:

- Solve the experienced I/O problems and add DMA support to the driver components. This possibly leads to a massively improved router performance. As the bottlenecks will move from the outer components to the components implementing the router's logic, this would also include optimizing the newly detected bottlenecks.

- Implement more sophisticated routing protocols and extend the router to support 3 or more NICs. This would require finding an appropriate hardware platform and porting drivers for the NICs to L4/Iguana. As a second step, a general routing architecture would have to be developed where components implementing different routing protocols can be added and removed on demand (similar to XORP). As a third step, actual routing protocols would have to be implemented in CAmkES. It should first be examined wether it is technically possible to directly port XORP components to CAmkES.

- Implement the remaining standard network stack protocols (e.g. ICMP and TCP) as CAmkES components so future CAmkES applications could use a complete network stack built out of CAmkES components.
  For TCP, this would also include the design of support for low overhead streams to be used by the application components. A promising approach might be to implement a connector that provides the basic stream operations like reading and writing bytes. This connector would connect an application component using TCP with a component implementing the TCP protocol.

- Improve the security of the system by adding support for transport layer security (TLS) protocols. TLS functionality could either be added as part of special connectors or be implemented in components dedicated to decrypting incoming network packets and encrypting outgoing network packets. An encrypted connection could be used to establish a secure connection between a remote machine and the NAT Admin component.

- Add components implementing static firewall functionality, statefull packet inspection or logging and packet sniffing functionality to the router. A firewall component would be useful to restrict internet access to a limited set of machines on the local network. Such a component simply would be plugged in between two existing components in the forwarding plane.

- Implement different traffic shaping concepts as components. This could include components to throttle the traffic generated by certain machines or to divide the available bandwidth equally between all machines currently using the router as a gateway to the global network. Furthermore, by adding other packet classifiers to the router, support for quality of service could be added.

- Add support for communication hardware other than ethernet controllers. This could include adding a component implementing a driver for a modem connected to the serial port. This component would have the same interfaces as the existing NIC drivers and would replace a NIC driver transparently.

5.3 Thanks

I would like to especially thank my supervisor Dr. Ihor Kuz for his friendly, prompt and competent support during this master's thesis and for his valuable comments during my work on this report. While I was working on my router project, he always made me feel like being a valuable member of the whole CAmkES project.

I would also like to thank Dr. Felix Rauch for supporting me during the organization of my thesis at NICTA and for always helping me whenever I needed any advice.

At ETH Zurich, I would like to thank Prof. Roger Wattenhofer for supervising my thesis from the ETH side.

Thanks also to Hans Dubach who helped me organize everything at ETH Zurich for this master's thesis abroad.

Finally, I would like to express my gratitude to NICTA and its ERTOS program. ERTOS made my master's thesis in Sydney possible and provided a great working environment in their Kensington Lab.

3

Appendix A

Thesis Project

A.1 Project Description

CAmkES[20], a project of NICTA[7]'s ERTOS[2] program, has recently developed and implemented a component architecture for its microkernel-based embedded operating system. The architecture allows the application of component-based software engineering to the development of embedded systems software, including operating system services such as file systems, network stacks, drivers, etc.

The first phase of the CAmkES project was the development of a low overhead component architecture. The next phase is to investigate the use of this architecture in building system software.

The proposed project forms part of this research, and involves the design and implementation of a significant system using the CAmkES component architecture. The project work will involve designing and implementing a network router (including drivers, network stack, and any other required operating system services) as a collection of interconnected components. The design should consider reusability and flexibility of the components so that they can be reused in different configurations and potentially lead to a dynamically reconfigurable router.

The implementation will run on our Iguana[18] L4[23]-based embedded operating system. Besides being a good test of the flexibility and suitability of the component architecture, the project will also involve analysing the resulting system (e.g., the understandability, ease of development, etc.), proposing changes or extensions to the architecture, and researching suitable design patterns and idioms for component-based embedded systems.

Depending on the progress made, possible further work may include comparing the resulting router to the Click modular router framework and consider, for example, whether click modules can be incorporated into the router as reusable components.

A.2 Project Plan

Starting date: November 7, 2006
Finishing date:May 6, 2007

November 2006

- familiarization with the CAmkES model and L4/Iguana
- familiarization with traditional router design
- router requirements, hardware platform choice
- start design of componentised router

- assessment: review of requirements, preliminary design and understanding of CAmkES and L4/Iguana

December 2006

- continue design of componentised router
- start implementation of router components
- assessment: review of final design

January 2007

- continue implementation of router components
- assessment: implementation progress

February 2007

- test implementation
- modify design and implementation as necessary
- assessment: review of implementation and test results

March 2007

- continue testing and modifying as necessary
- experiment with the changing of component-based designs, investigate flexibility of the model and the usefulness of the mechanisms provided by CAmkES
- assessment: suitability of initial design and proposed changes

April/May 2007

- wrap up work
- present work and results to ERTOS research group
- write a research report outlining the work. This research report will lead to a thesis
- assessment: assessment of presentation and report

Appendix B

Version History

Version 0.98 (andri.toggenburger@nicta.com.au–devel/camkes-project–router–0.98)

- Support for user defined connectors added.
- Ported all existing connectors to the user defined connector architecture.
- Implemented an instrumented version of the IguanaRPC connector (IguanaRPCMeasure) to measure the delay caused by different stages in the IguanaRPC connector.

Version 0.97 (andri.toggenburger@nicta.com.au–devel/camkes-project–router–0.97)

- Final design of the router (see 2.5). Protection between application plane and forwarding plane added.

Version 0.96 (andri.toggenburger@nicta.com.au–devel/camkes-project–router–0.96)

- Forwarding plane uses direct method invocation to forward packets (no queues in between the components). However, the driver components are still connected to the rest of the system by queues.
- The system uses only DirectCall connectors for connections where packets are forwarded.
- The application plane is connected to the forwarding plane through queues.

Version 0.95 (andri.toggenburger@nicta.com.au–devel/camkes-project–router–0.95)

- DirectCall connector implemented.
- IguanaSharedData connector generalized so 2+ components can share data.
- Replaced IguanaRPC connections between the forwarding components by DirectCall connections.
- All components share a common memory area to store network packets.
- Memory Allocator components implemented so the memory in the shared memory area can be dynamically allocated and deallocated by all of the components

Version 0.9 (andri.toggenburger@nicta.com.au–devel/camkes-project–router–0.9)

- First working version of the router.
- There are queues between all network stack components.
- Only pairs of components share common memory sections, so the network packets have to be copied every time they are passed to another (third) component.
- Only IguanaRPC connectors are used to forward network packets.

Appendix C

NatAdminConsole

The Java program `NatAdminConsole` is a console application that can be used to remotely configure the NAT component in the router and to query its status. It connects via UDP to the NAT Admin component running in the application plane of the router. The NAT Admin component is bound to the global network interface.

The parameters to start the application are `<IP address of router's global interface> <port used by NAT Admin server>`. The port where the NAT Admin server is listening is set to 3201 in the current implementation.

These are the commands supported by the console:

- N_PACKETS_FORWARDED_TCP: returns the number of the translated TCP packets by the NAT.
- N_PACKETS_FORWARDED_UDP: returns the number of the translated UDP packets by the NAT.
- N_PACKETS_FORWARDED_ICMP: returns the number of the translated ICMP packets by the NAT.
- N_PACKETS_DROPPED: returns the number of packets arriving from the global network that were discarded by the NAT.
- N_DYNAMIC_ENTRIES: returns the number of the dynamic entries in the NAT lookup tables.
- N_STATIC_ENTRIES: returns the number of configured port forwarding declarations.
- GET_GLOBAL_IP: returns the IP address of the global interface (assigned by DHCP).
- SET_ENABLE_NAT <true / false>: enables or disables the NAT. If it is disabled, all packets are dropped.
- SET_STATIC_ENTRY <localip> <globalport>: enables the forwarding of all packets directed to the port <globalport> to the machine with the address <localip> on the local network (enables port forwarding).
- DELETE_STATIC_ENTRY <localip> <globalport>: disables the forwarding of packets directed to the port <globalport> to the machine with the address <localip> on the local network (disables port forwarding).
- GET_STATIC_ENTRIES: print all defined portforwarding declarations.
- GET_DYNAMIC_ENTRIES: print the dynamic NAT lookup table used to translate packets between the local and the global network.

Appendix D

Gumstix - How To

In this section, the steps to connect to a Gumstix machine are explained. Furthermore, an explanation of how to upload a bootimage to a Gumstix machine via TFTP is given. This explanation assumes the following hardware to be present:

- Gumstix mainboard
- Gumstix ethernet extension board (1 or 2 NICs). The primary port has to be connected to a LAN.
- Gumstix serial extension board connected to a PC.

Connecting to the Gumstix via Serial

The program `kermit` has to be installed on the system first.

Then execute the following commands in a Linux console:

1. `su`
2. `kermit -l /dev/ttyS0`
3. issue the following commands in `kermit`:

```
set speed 115200
set reliable
fast
set carrier-watch off
set flow-control none
set prefixing all
```

4. issue the command `connect` in `kermit`
5. power up Gumstix machine

If everything works fine, the `uboot` boot screen of the Gumstix machine should be displayed in `kermit`.

Load Bootimages from a TFTP Server

Depending on your Linux distribution, either set up a `TFTP` server via console commands or by using a configuration menu (e.g. YaST in SUSE Linux). Then add the bootimage to be uploaded to the Gumstix to the files the `TFTP` server provides.

Now, the Gumstix has to be configured in order to be able to connect to a TFTP server. After powering up the Gumstix, hit any key to stop autoboot (2s limit). This loads an uboot menu.

Configure the IP settings of the Gumstix by setting the following environment variables (using the command setenv "<args>").

- ipaddr: set this to the desired IP address of the Gumstix network interface (e.g. 10.13.1.250).
- netmask: set netmask of the network the Gumstix has to be part of (e.g. 255.255.254.0).
- serverip: set IP address of the TFTP server to be used (e.g. 10.13.1.148).
- bootcmd: set this variable to "tftpboot a2008000 <bootimage filename (e.g. bootimg.bin)> ; go a2008000"

Example:

```
setenv ipaddr ''10.13.1.250'';
setenv netmask ''255.255.254.0'';
setenv serverip ''10.13.1.148'';
setenv bootcmd ''tftpboot a2008000 bootimg.bin; go a2008000''
```

As these environment variables are not saved in permanent storage, issue the command saveenv to permanently save the environment variables. The Gumstix is now set up for TFTP. In order to test if the Gumstix machine is able to connect to the TFTP sever, use the command ping.

Reboot the Gumstix machine. It should now load the defined boot image from the specified TFTP server and boot up.

Appendix E

UDP Echo Clients and Servers

UDP Echo clients and servers were implemented for different platforms. They implement functionality similar to the `ping` program but on the transport layer. An UDP echo service sends back the payload of a received UDP echo request to the sender.

Java UDP Echo Client / Server

This Java program implements an UDP echo client and an UDP echo server. The `main` function of this program is in the class `Test` and has to be started with the argument `<send>` in order to start the client or with the argument `<receive>` to start the server.

The UDP echo client measures the time it takes to send and receive a certain amount of UDP echo requests and replies. It waits for a pending echo reply to arrive before it sends out the next echo request. At the end, it prints out the total time it has taken to send and receive the configured number of echos. Parameters like UDP packet size, server IP, server port and the number of echos to be sent can be configured by setting the corresponding constants or variables in its implementation file `Sender.java`.

The UDP echo server simply sends back all of the received UDP packets to the sender. Its server port can be configured by setting the according constant in its implementation file `Server.java`.

CAmkES UDP Echo Server

The router's UDP echo service that implements UDP echo server functionality is, for simplicity reasons, implemented in the DHCP component and can be turned on or off by defining the according constant in the file `include/debugprint.h`. It is bound to the global router interface and listens for UDP echo requests on port 3200.

Linux for Gumstix UDP Echo Server

This is a C program that implements an UDP echo server. It listens on port 3200 for arriving echo requests and sends an echo reply back to the sender. Its implementation file is `udpecho.c`. In order for it to run on Linux for a Gumstix machine, it has to be compiled with the compilers supplied with the Gumstix Buildroot[4]

Appendix F

IDL and ADL Files of the Router Application

net.camkes

```
import "std_connector.camkes";
import "Driver_interface.idl4";
import "UDP_interface.idl4";
import "BidirQueue_interface.idl4";
import "Malloc_interface.idl4";
import "PacketCallback2_interface.idl4";
import "PacketCallback_interface.idl4";
import "NatAdmin_interface.idl4";

component Malloc{//k-d malloc
  provides Malloc mallocd;
  dataport Buf d;
}
component MallocPre{//pre-allocated
  provides Malloc mallocd;
  dataport Buf d;
}

component NatAdmin{
  control;
  dataport Buf d;
  dataport Buf data_nat_natadmin;
  uses BidirQueue q_udp_natadmin;
  uses NatAdmin nat;
  uses Malloc mallocd;
  uses Driver driver;
  uses UDP udp;
  consumes PacketArrivedEvent e_packet_up;
  consumes DriverEvent e_netinfo_changed;
}

component DummyCallback{
  //this is used to keep the compiler quiet for interfaces not used
  provides PacketCallback cb;

}

component CopyingBidirQueue {
        provides BidirQueue queue;
```

```
uses Malloc m_down;
uses Malloc m_up;
dataport Buf d_down;
dataport Buf d_up;
        emits PacketArrivedEvent q_e_up;
        emits PacketArrivedEvent q_e_down;
}

component BidirQueue {
        provides BidirQueue queue;
        emits PacketArrivedEvent q_e_up;
        emits PacketArrivedEvent q_e_down;
}

component DummyBidirQueue {
          //queue to store void * values. it is a double queue compromised of
          //an up-queue and a down-queue (e.g. up or down the stack)
  //it is used to keep the compiler quiet
        provides BidirQueue queue;
//event is triggered if queue was empty and a down packet or up packet arrived
        emits PacketArrivedEvent q_e_up;
        emits PacketArrivedEvent q_e_down;
}

component Driver{
control;
uses BidirQueue q_driver_arp;
consumes PacketArrivedEvent e_packet_down;
uses Malloc mallocd;
provides Driver driver;
dataport Buf d;
dataport Buf test;
emits DriverEvent e_netinfo_changed;
attribute string gpio_string;
attribute string memoryarea_string;
attribute string mac_string;
attribute string id_string;
}

component InitManager {
//initialise network cards, set static network info for local NIC
  control;
  uses Driver driver;
  uses Driver driver2;
  attribute string local_ip_string;
  attribute string local_netmask_string;
  attribute string local_gw_string;
}

component DHCP {
  control;
  uses Malloc mallocd;
  uses BidirQueue q_udp_dhcp;
  consumes PacketArrivedEvent e_packet_up;
  uses UDP udp;
  uses Driver driver;
  dataport Buf d;
  consumes DriverEvent e_netinfo_changed;
```

```
}

component ARPHandler {
  control;
  uses Driver driver;
  uses Malloc mallocd;
  uses BidirQueue q_driver_arp;
  uses PacketCallback2 gateway_callback_2;
  uses PacketCallback gateway_callback_1;
  uses PacketCallback to_udp_callback;
  consumes PacketArrivedEvent e_packet_up;
  provides PacketCallback fromgateway_callback;
  provides PacketCallback2 fromudp_callback;
  dataport Buf d;
  consumes DriverEvent e_netinfo_changed;
  attribute string mode;
}

component UDP {
         control;
 uses Malloc mallocd;
 uses BidirQueue q_udp_dhcp;
 uses BidirQueue q_udp_natadmin;
 consumes PacketArrivedEvent e_packet_down_dhcp;
 consumes PacketArrivedEvent e_packet_down_natadmin;
 uses  PacketCallback2 to_arp_callback;
 provides PacketCallback from_arp_callback;
 uses Driver driver;
 dataport Buf d;
         provides UDP udp;
consumes DriverEvent e_netinfo_changed;
}

component NAT {
  uses Driver globaldriver;
  uses Malloc mallocd;
  dataport Buf d;
  dataport Buf data_nat_natadmin;
  provides PacketCallback local_packet;
  provides PacketCallback2 global_packet;
  provides NatAdmin nat;
  uses PacketCallback to_local_callback;
  uses PacketCallback to_global_callback;
  consumes DriverEvent e_global_netinfo_changed;
}

assembly {
composition {
//processing stack components
component Driver driver1;
component Driver driver2;
component ARPHandler arp1;
component ARPHandler arp2;
component UDP udp1;
component DHCP dhcp1;
component InitManager initmanager1;
component NAT nat;
component NatAdmin natadmin;
```

```
component BidirQueue q_driver_arp;
component CopyingBidirQueue q_udp_dhcp;
component CopyingBidirQueue q_udp_natadmin;
component BidirQueue q_driver2_arp2;
component DummyCallback dummy_cb;

//Memory manager for forwarding plane
component Malloc m1;

//for DHCP app
component Malloc m2;

//for Nat admin app
component Malloc m3;

//***** Dataport connections ******
connection IguanaSharedData test1(from driver1.test, to driver2.test);

//** components to malloc m1
connection IguanaSharedData driver1_m1_data(from m1.d, to driver1.d);
connection IguanaSharedData arp1_m1_data(from m1.d, to arp1.d);
connection IguanaSharedData udp1_m1_data(from m1.d, to udp1.d);
//connection IguanaSharedData dhcp1_m1_data(from m1.d, to dhcp1.d);
connection IguanaSharedData nat_m1_data(from m1.d, to nat.d);
connection IguanaSharedData driver2_m1_data(from m1.d, to driver2.d);
connection IguanaSharedData arp2_m1_data(from m1.d, to arp2.d);
connection IguanaSharedData natadmin_m1_data(from m1.d, to
natadmin.d);
connection IguanaSharedData q_udp_dhcp_m1_data1(from m1.d, to
q_udp_dhcp.d_down);
connection IguanaSharedData q_udp_natadmin_m1_data1(from m1.d, to
q_udp_natadmin.d_down);

//components to malloc m2
connection IguanaSharedData dhcp1_m2_data(from m2.d, to dhcp1.d);
connection IguanaSharedData q_udp_dhcp_m2_data2(from m2.d, to
q_udp_dhcp.d_up);

//components to malloc m3
connection IguanaSharedData natadmin_m3_data(from m3.d, to natadmin.d);
connection IguanaSharedData q_udp_natadmin_m2_data2(from m3.d, to
q_udp_natadmin.d_up);

//** natadmin and nat
connection IguanaSharedData nat_natadmin_data(from nat.data_nat_natadmin, to
natadmin.data_nat_natadmin);


//***** function call connectors ****//


//** to driver1
connection IguanaRPC dhcp1_driver1_driver(from dhcp1.driver, to driver1.driver);
connection IguanaRPC udp1_driver1_driver(from udp1.driver, to
driver1.driver);
connection IguanaRPC arp1_driver1_driver(from arp1.driver, to
driver1.driver);
connection IguanaRPC initmanager1_driver1_driver(from initmanager1.driver, to
```

```
driver1.driver);
connection IguanaRPC nat_driver1_driver(from nat.globaldriver, to
driver1.driver);
connection IguanaRPC natadmin_driver1_driver(from natadmin.driver, to
driver1.driver);


//**to driver2
connection IguanaRPC initmanager1_driver2_driver(from initmanager1.driver2, to
driver2.driver);
connection IguanaRPC arp2_driver1_driver(from arp2.driver, to
driver2.driver);


//** to udp1
connection IguanaRPC dhcp1_udp1_udp(from dhcp1.udp, to
udp1.udp);
connection IguanaRPC natadmin_udp1_udp(from natadmin.udp, to
udp1.udp);


//** to malloc 1
connection DirectCall driver1_m1_malloc(from driver1.mallocd, to
m1.mallocd);
connection DirectCall driver2_m1_malloc(from driver2.mallocd, to m1.mallocd);
connection DirectCall arp1_m1_malloc(from arp1.mallocd, to
m1.mallocd);
connection DirectCall arp2_m1_malloc(from arp2.mallocd, to m1.mallocd);
connection DirectCall udp1_m1_malloc(from udp1.mallocd, to
m1.mallocd);
connection DirectCall q_udp_dhcp_m1_mallocdown(from q_udp_dhcp.m_down, to
m1.mallocd);
connection DirectCall nat_m1_malloc(from nat.mallocd, to m1.mallocd);
connection DirectCall q_udp_natadmin_m1_malloc(from q_udp_natadmin.m_down,
           to m1.mallocd);

// to malloc 2
connection DirectCall dhcp1_m2_malloc(from dhcp1.mallocd, to
m2.mallocd);
connection DirectCall q_udp_dhcp_m2_mallocup(from q_udp_dhcp.m_up, to
m2.mallocd);

//to malloc 3
connection DirectCall natadmin_m3_malloc(from natadmin.mallocd, to
m3.mallocd);
connection DirectCall q_udp_natadmin_m3_mallocup(from q_udp_natadmin.m_up, to
m3.mallocd);



//** connections between queues and components
connection DirectCall driver1_q_driver_arp_bidirqueue(
           from driver1.q_driver_arp, to q_driver_arp.queue);
connection DirectCall arp1_q_driver_arp_bidirqueue(
           from arp1.q_driver_arp, to q_driver_arp.queue);

//DHCP Client
connection IguanaRPC udp1_q_udp_dhcp_bidirqueue(from udp1.q_udp_dhcp, to
```

```
q_udp_dhcp.queue);
connection IguanaRPC dhcp1_q_udp_dhcp_bidirqueue(from dhcp1.q_udp_dhcp, to
q_udp_dhcp.queue);

//Nat admin server
connection IguanaRPC udp1_q_udp_natadmin_bidirqueue(from udp1.q_udp_natadmin, to
q_udp_natadmin.queue);
connection IguanaRPC natadmin_q_udp_natadmin_bidirqueue(
          from natadmin.q_udp_natadmin,to q_udp_natadmin.queue);

//to nat administration interface
connection IguanaRPC natadmin_nat_natadmin(from natadmin.nat, to
nat.nat);


//**
connection DirectCall driver2_q_driver2_arp2_bidirqueue(from
driver2.q_driver_arp, to q_driver2_arp2.queue);
connection DirectCall arp2_q_driver2_arp2_bidirqueue(from
arp2.q_driver_arp, to q_driver2_arp2.queue);


//**** event connectors ****//

//** queues to components dequeueing packets
connection IguanaAsynchEvent q_driver_arp_driver1(from
   q_driver_arp.q_e_down, to driver1.e_packet_down);

connection IguanaAsynchEvent q_driver_arp_arp1(from
   q_driver_arp.q_e_up, to arp1.e_packet_up);


       //dhcp client queue
connection IguanaAsynchEvent q_udp_dhcp_udp1(from
   q_udp_dhcp.q_e_down, to udp1.e_packet_down_dhcp);

      connection IguanaAsynchEvent q_udp_dhcp_dhcp1(from
   q_udp_dhcp.q_e_up, to dhcp1.e_packet_up);

       //natadmin server queue
connection IguanaAsynchEvent q_udp_natadmin_udp1(from
   q_udp_natadmin.q_e_down, to udp1.e_packet_down_natadmin);

      connection IguanaAsynchEvent q_udp_natadmin_natadmin(from
   q_udp_natadmin.q_e_up, to natadmin.e_packet_up);

   //**
      connection IguanaAsynchEvent q_driver2_arp2_driver2(from
   q_driver2_arp2.q_e_down, to driver2.e_packet_down);
      connection IguanaAsynchEvent q_driver2_arp2_arp2(from
   q_driver2_arp2.q_e_up, to arp2.e_packet_up);



//** driver1 to stack components
 connection IguanaAsynchEvent driver1_dhcp1_e(from
   driver1.e_netinfo_changed, to dhcp1.e_netinfo_changed);
       connection IguanaAsynchEvent driver1_udp1_e(from
```

```
   driver1.e_netinfo_changed, to udp1.e_netinfo_changed);
       connection IguanaAsynchEvent driver1_arp1_e(from
   driver1.e_netinfo_changed, to arp1.e_netinfo_changed);
       connection IguanaAsynchEvent driver1_nat_e(from
   driver1.e_netinfo_changed, to
   nat.e_global_netinfo_changed);
connection IguanaAsynchEvent driver1_natadmin_e(from
   driver1.e_netinfo_changed, to natadmin.e_netinfo_changed);

   //** driver2 to other components

connection IguanaAsynchEvent driver2_arp2_e(from
   driver2.e_netinfo_changed, to arp2.e_netinfo_changed);



//*****callbacks between components (to fwd packets)
connection DirectCall arp1_nat_callback(from arp1.gateway_callback_2, to
nat.global_packet);
connection DirectCall arp2_nat_callback(from arp2.gateway_callback_1, to
nat.local_packet);

//dummy connections (to deal with problem in camkes which doesnt allow
//a component to implement several interfaces of the same type
connection DirectCall arp2_nat_callback_2(from arp1.gateway_callback_1, to
nat.local_packet);
connection DirectCall arp1_nat_callback_2(from arp2.gateway_callback_2, to
nat.global_packet);


connection DirectCall nat_arp1_callback(from nat.to_local_callback, to
arp2.fromgateway_callback);
connection DirectCall nat_arp2_callback(from nat.to_global_callback, to
arp1.fromgateway_callback);

connection DirectCall arp1_udp1_callback(from arp1.to_udp_callback, to
udp1.from_arp_callback);
connection DirectCall udp1_arp1_callback(from udp1.to_arp_callback, to
arp1.fromudp_callback);

connection DirectCall arp2_udp2_callback(from arp2.to_udp_callback, to
dummy_cb.cb);

      }

      configuration {
//forwarding activated (so it forwards packets to gateway)
               arp2.mode = "forward";

//forward everything (except packets to port 67, 3200, 3201 on
//connected netif)
arp1.mode = "nat_global";

//parameters for driver1
driver1.memoryarea_string = "67108864"; //=="0x04000000"
driver1.mac_string = "10:10:10:10:10:1";
driver1.gpio_string = "36";
driver1.id_string = "1";
```

```
//parameters for driver2
driver2.memoryarea_string = "134217728"; //=="0x08000000"
driver2.mac_string = "10:10:10:10:10:2";
driver2.gpio_string = "27";
driver2.id_string = "2";

//parameter for initmanager (used to set static ip/gw/netmask of
//local driver (driver2))
initmanager1.local_ip_string = "192.168.0.1";
initmanager1.local_gw_string = "192.168.0.1";
initmanager1.local_netmask_string = "255.255.255.0";
}

}
```

Driver_interface.idl4

```
interface Driver {
int setup();
int getIpAddress();
int getHwAddressLength();
int getMac(in int i);
int getGwAddress();
int getNetmask();
void setNetworkInfo(in int ip, in int gw, in int netmask);
};
```

UDP_interface.idl4

```
interface UDP {
int abind(in int clientid, in int c_ip, in int c_port);
int aunbind(in int clientid, in int c_ip, in int c_port);
int aconnect(in int clientid, in int s_ip, in int s_port);
};
```

BidirQueue_interface.idl4

```
interface BidirQueue {
int push_up(in void *x);
int push_down(in void *x);
void *pull_up(void);
void *pull_down(void);
};
```

Malloc_interface.idl4

```
interface Malloc {
void *malloc(in int size);
void free(in void* p);
void *realloc(in void *ptr, in int size);
};
```

PacketCallback_interface.idl4

```
interface PacketCallback {
void callback(in void *p);
```

```
};
```

PacketCallback2_interface.idl4

```
interface PacketCallback2 {
void callback(in void *p);
};
```

NatAdmin_interface.idl4

```
interface NatAdmin {
int n_packets_forwarded_tcp(void);
int n_packets_forwarded_udp(void);
int n_packets_forwarded_icmp(void);
int n_packets_dropped(void);
int n_dynamic_entries(void);
int n_static_entries(void);
int get_global_ip(void);
int set_enable_nat(in int flag);
int set_static_entry(in int ip, in int port);
int delete_static_entry(in int ip, in int port);
void *get_static_entries(void);
void *get_dynamic_entries(void);
};
```

Bibliography

[1] ARM Limited.
http://www.arm.com/.

[2] Embedded, Real-Time, and Operating Systems Research Program.
http://www.ertos.nicta.com.au.

[3] Gumstix - Way Small Computing.
http://www.gumstix.com

[4] Gumstix Buildroot - Create Linux Boot Images for Gumstix
http://docwiki.gumstix.org/Buildroot

[5] The L4-embedded Project
http://www.ertos.nicta.com.au/research/l4/

[6] Magpie - Interface Compiler for L4.
http://www.ertos.nicta.com.au/software/kenge/magpie/latest/

[7] National ICT Australia - Australia's ICT Research Centre of Excellence.
http://www.nicta.com.au.

[8] SMC LAN91c111 Data Sheet.
http://www.smsc.com/main/datasheets/91c111.pdf

[9] SMC LAN91C111 Frequently Asked Questions.
http://www.smsc.com/main/catalog/lan91c111faq.pdf

[10] thrulay, network capacity tester.
http://shlang.com/thrulay/

[11] Carl van Schaik, Embedded Real-Time and Operating Systems Program (ERTOS), "NICTA L4-embedded Kernel Reference Manual", Sydney, November 2005.

[12] R. Droms, "Dynamic Host Configuration Protocol", RFC 2131, March 1997.

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

[14] Adam Dunkels, "Desing and Implementation of the lwIP TCP/IP Stack", Swedish Institute of Computer Science, February 2001.

[15] Yitzchak Gottlieb and Larry Peterson, "A Comparative Study of Extensible Routers", Department of Computer Science, Princeton University, 2002.

[16] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, Pavlin Radoslavov, "Designing Extensible IP Router Software", International Computer Science Institute, University College, London, 2005.

[17] Mark Handley, Orion Hodson, Eddie Kohler, "XORP Goals and Architecture", ACM SIGCOMM Hot Topics in Networking 2002.

[18] Gernot Heiser, "Iguana User Manual", Embedded Real-Time and Operating Systems Program (ERTOS), Sydney, April 2005.

[19] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek, "The Click Modular Router", ACM Transactions on Computer Systems 18(3), August 2000, pages 263-297.

[20] Ihor Kuz and Yan Liu and Ian Gorton and Gernot Heiser, "CAmkES: A component model for secure microkernel-based embedded systems", Journal of Systems and Software, Volume 80, Issue 5 (May 2007).

[21] Ihor Kuz, "CAmkES Core Design and Iguana Implementation", NICTA's Embedded, Real-Time, and Operating Systems Research Program, Sydney, 2006.

[22] Ihor Kuz, "L4 User Manual - NICTA L4-embedded API", NICTA's Embedded, Real-Time, and Operating Systems Research Program, Sydney, 2005.

[23] Jochen Liedke, "On μ-Kernel Construction", 15th ACP Symposium of Operating System Principles (SOSP) December 3-6 1995, Copper Mountain Resort, Colorado.

[24] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, John H. Hartman, "Scout: A Communications-Oriented Operating System", Department of Computer Science, The University of Arizona, 1994.

[25] Nicolas Pitre, "Linux Driver for SMSC's 91C9x/91C1xx single-chip Ethernet devices". Copyright (C) 2003 Monta Vista Software, Inc.

[26] D. Plummer, "Ethernet Address Resolution Protocol", RFC 826, November 1982.

[27] J. Postel, "Internet Control Message Protocol", RFC 792, September 1981.

[28] J. Postel, "Internet Protocol", RFC 791, September 1981.

[29] J. Postel, "Transmission Control Protocol", RFC 793, September 1981.

[30] J. Postel, "User Datagram Protocol", RFC 768, August 1980.

[31] P. Srisuresh, K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", RFC 3022, January 2001.

[32] XORP Project, "XORP Design Overview", International Computer Science Institute, Berkeley, 2007.

[33] Hubert Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", IEEE Transactions on Communications, Vol. Com-28, No. 4, April 1980.