ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Masters Thesis

The ANA Project

Development of the ANA-Core Software

September 21, 2007

Ariane Keller

Supervisor: Prof. Dr. B. Plattner
Advisor: Theus Hossmann, Dr. Martin May, Dr. Rainer Baumann

Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

In recent years, the Internet has reached enormous popularity, but at the same time its weaknesses become evident: Huge efforts have to be taken to manage all participating nodes, and the infrastructure is not suitable to integrate emerging network paradigms such as sensor networks or delay tolerant networks. Therefore several research projects try to reinvent the Internet and provide a network architecture which is better armed for future needs.

This thesis is situated in the ANA project which builds an autonomic network based on a clean slate approach. The ultimate goal is to develop a novel autonomic network architecture that enables flexible, dynamic, and fully autonomic formation of networks.
In an ANA node two main components can be identified: The *MINMEX* and the *Playground*. The Playground hosts the elements (*functional blocks*) providing networking functionality. The MINMEX ties these functional blocks together and dispatches the data amongst them.
In this Masters thesis we have developed the communication infrastructure between the MINMEX and the functional blocks. Due to this infrastructure ANA developers will never have to care about how packets get transferred between the different functional blocks.
The infrastructure is implemented for Linux kernel 2.4 and 2.6 as well as for the Linux user space. This allows to run ANA in different environments, reaching from embedded devices to personal computers, without the need of changing anything on the given operating system.
In a second phase we have designed a bootstrapping process. This bootstrapping process can be used by any functional block in ANA, regardless whether it implements a high level application or a low level communication facility. With the help of this bootstrapping process different ANA nodes can start to communicate together.
In order to illustrate the operation of the ANA core software as well as the bootstrapping process we have implemented the first application running on ANA: A chat program.

Acknowledgments

With this Masters thesis I complete my studies in Information Technology and Electrical Engineering at the Swiss Federal Institute of Technology (ETH) Zurich.

A number of people have contributed to this thesis and I would like to express my gratitude to them.

I would like to thank Prof. Dr. Bernhard Plattner for the supervision of my Masters thesis.

I would like to thank my advisors Dr. Martin May and Theus Hossmann. I am very thankful that Martin had that much confidence in me and let me work in this fascinating project. I thank Theus for the time and effort he put in supporting me with this thesis.
Special thanks to Dr. Rainer Baumann. He knew my interests and arranged this exciting Masters thesis for me. During my thesis he aided me with his experience in holding presentations and writing reports.
I'm thankful that Rainer and Martin encouraged me to stay at ETH after my Masters thesis is completed. I'm looking very much forward to this experience.

I would like to thank Thomas Steingruber for equipping me with all the necessary hardware, regardless whether it was a mouse, a switch or a third computer. His door was always open and he fullfilled my wishes always straight away.

I would like to thank Dr. Christophe Jelger and Ghazi Bouabene from the University of Basel for all the inspiring discussions and the great collaboration during the development of the ANA prototype.

Zurich, September 2007

Ariane Keller

Contents

List of Figures

1 Introduction

1.1 Motivation

Today, the Internet has reached enormous popularity. Mostly due to the enormous amount of applications provided and the ease of use for the end-user. But as the number and diversity of network capable devices increase and the applications make higher demands on the underling network, some drawbacks get apparent:

- For small devices (e.g. in sensor networks) it is not feasible to provide the whole TCP/IP stack, since it consumes a lot of resources. Therefore, it is desirable to let them communicate using a simpler protocol. But unfortunately the Internet requires the IP protocol for every participating node.
- With the number of participants in the Internet the management overhead increases. If one system administrator is assumed per 100 nodes, 10 million system administrators are needed for 1 billion nodes. The introduction of ubiquitous computing and sensor networks may increase the number of nodes beyond 1 billion. Therefore system administrators as well as users may be swamped with the increasing system complexity [29].
- The success of the Internet made it a valuable goal for attackers: Viruses, worms and Denial-of-Service attacks are well known problems. The Internet was not designed with security in mind, but as an open system with distributed control and mutual trust [1]. Therefore it is not possible to protect the network against malicious users without changing the architecture of the Internet significantly [2].
- Firewalls as well as NAT enabled routers pose difficulties for some legitimate applications like VoIP. They block incoming connections, either because the specified destination port is blocked, or because the specified port is not yet known by the NAT device. Firewalls may have to be reconfigured, in order to allow incoming traffic to some applications. And nodes behind a NAT device have to initiate all connections. This is difficult in case of VoIP because the caller is never known in advance.
- Many applications have some Quality of Service (QoS) requirements. But the Internet does not provide any help for QoS. In order to provide optimal performance some applications (e.g. video conferencing) need to estimate the available bandwidth, latency etc. If many applications perform these measurements actively (e.g. by sending probe packets), the network load increases unnecessary [4]. It would be preferable that the network itself provides some QoS mechanisms as well as information about the network status.

To eliminate these drawbacks the core network infrastructure needs to be changed fundamentally. But unfortunately the infrastructure of the Internet is not likely to change, as several examples (RSVP, IPv6 etc.) have shown in the

past. One major problem are the Internet Service Provider (ISP) which are very conservative in deploying anything that does not lead to a financial benefit [5].

Reducing the enormous management overhead would be a motivation for ISPs to introduce a new networking architecture. Such a future network architecture should resolve all drawbacks from the current Internet, and it should be flexible enough to integrate new solutions instead of putting new functionality on top of the network architecture.

In the next few paragraphs we describe some current research areas which are targeted to overcome the drawbacks of the Internet architecture. There exists already a variety of proposals for new network architectures. They can be coarsely divided into two parts:

1. *Internet patches*: All proposals in this category are based on the Internet architecture. They provide extensions or modifications of the current architecture.

2. *Clean slate approaches*: A clean slate network architecture is one that does not depend on the current Internet.

[7] summarizes these approaches and concludes that many approaches deal with the separation of *identifiers* and *locators*, contrarily to today's Internet where the IP address is both, an identifier and a locator.
Another important topic is *end-to-end connectivity*. If end-to-end connectivity is provided at higher layers, the network does not need to provide global reachability. Independent addressing realms can become part of the network architecture and global reachability becomes a property which can be switched on and off.
Another research field concerning new network architectures is *active networking*. In active networking one comes away from the principle that switches have to be simple devices. Instead, they become active and execute code. There are two kinds of active networks:

- *Strong active networks*: in a strong active network the end user inserts programs to the network which will be executed by the routers.

- *Moderate active networks*: the programs to be executed are provisioned by the network operator, only the data to be processed is inserted by the end user.

It is evident that the properties of an active network can be changed easily, and therefore it is much more flexible than the current Internet.

Another research field are *autonomic networks*. The aim of such network architectures is to create self-managing networks to overcome the rapidly growing complexity of the Internet and other networks and to enable their further growth, far beyond the size of today [42]. In an autonomic network the human interaction with the network is as small as possible. The network should be able to manage itself with the help of policies specified by its operators and users. To achieve this goal IBM has defined in 2003 the following four functional areas [45]:

- *Self-Configuration*: Automatic configuration of components.
- *Self-Healing*: Automatic discovery, and correction of faults.
- *Self-Optimization*: Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements.
- *Self-Protection*: Proactive identification and protection from arbitrary attacks.

Clearly, these are challenging goals. But an autonomic network architecture can greatly facilitate the maintenance and the incorporation of new features in the future Internet.

1.2 The ANA Project

The ANA (Autonomic Network Architecture) project [35] tries to build an autonomic network based on a clean slate approach. It is a European Union funded project in "Situated and Autonomic Communications" [9]. The ANA project has started in January 2006 and will last until end of 2009. Universities, research institutes and industry partners from Europe[1] are participating in this project.

The ANA project is divided into the following workpackages:

1. Architecture: architecture of autonomic networks, implementation of the "ANA core software".
2. Communication mechanisms: naming, addressing and routing schemes.
3. Self-* mechanisms: principles, mechanisms and proof-of-concepts necessary for self-management and resilience.
4. Testbed: testbed development and deployment.

The ultimate goal is to develop a novel autonomic network architecture that enables flexible, dynamic, and fully autonomic formation of network nodes as well as whole networks. It should exhibit a maximum degree of flexibility and provide support for functional scaling. Functional scaling means that the network is able to completely integrate new functionality. This is accomplished by abandoning the *one-size-fits-all* network architecture of the Internet and by providing an architectural framework which enables the co-existence of different network architectures. As a result, a main abstraction of ANA is the *compartment*. Each compartment can be individually managed and may use a completely different set of communication protocols.

In ANA, the network stack is not fixed as in the Internet, but it is dynamically built depending on the networks needs. This flexibility is achieved by defining a *Minimal INfrastructure for Maximal EXtensibility (MINMEX)* which has to be provided by any ANA node. The MINMEX provides the functionality which is required to bootstrap an run ANA. The actual networking functionality is implemented in the *playground*. The playground is an accumulation of *functional*

[1] and also one university from Northern America

blocks, each of which provides a certain networking service (e.g. encryption, compression, reliable packet transport etc.). The MINMEX coordinates the packet flow from one functional block to another. For this reason there is no direct communication between different functional blocks, but all communication is routed over the MINMEX. Figure 1 shows an overview of an ANA node. For more information about the ANA architecture refer to section 3 or the ANA Blueprint [28].
In the ANA project there is a team responsible for providing other developers with the ANA core software. This software allows future developers to populate the ANA node with autonomic features.

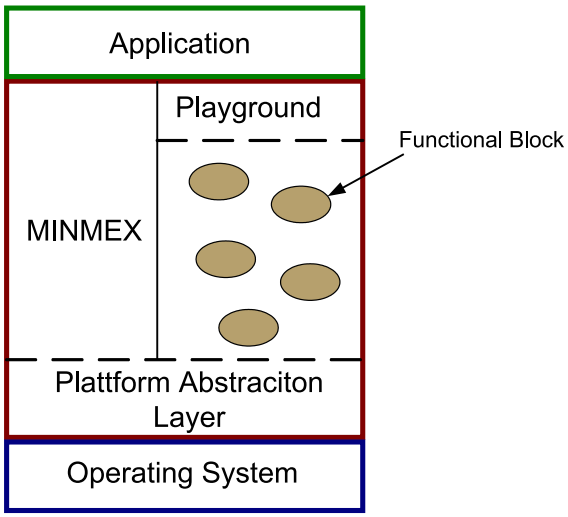


Figure 1: Overview over an ANA node.

1.3 Goals of this Thesis

This thesis is situated in the ANA project, more exactly in the development of the ANA core software. When this thesis started in March 2007 the development of the ANA core software had just started.

The ANA core should provide as much flexibility as possible, therefore we decided, that it should run in the Linux user as well as in the Linux kernel space.

This thesis covers mainly the following two areas:

1. Provide the underlying communication facilities between the MINMEX and the functional blocks in the Linux kernel space.
2. Design and implementation of a bootstrapping phase which allows different nodes to start to communicate together.

Besides these two main topics, a large variety of other tasks were performed, reaching from debugging of code in the different ANA core areas, to giving some presentations on an ANA coding workshop.

1.4 Outline

This report is structured as follows: Chapter 2 provides in its first part some information on other networking projects. In a second part it introduces some protocols which allow to discover autonomously some services provided on a network. The last part discusses some inter process communication mechanisms, since inter process communication is an important aspect of the ANA core implementation.
Chapter 3 summarizes the concepts underlying the ANA architecture and it explains how these concepts are mapped to the actual implementation.
Chapter 4 describes in detail the implementation of the ANA core in the Linux kernel space. It describes how the communication mechanisms between the MINMEX and the functional blocks are implemented and gives a short performance evaluation. Since the ANA core should run on kernel 2.4 and 2.6 as well as in user space it discusses some aspects of writing system agnostic code and it presents the main differences between the Linux kernel 2.4 and 2.6.
The second phase of this Masters thesis is covered in chapter 5. It describes the design of the bootstrapping phase in which different nodes learn about each other.
In chapter 6 we summarize the contributions of this Masters thesis and give an outlook over the next steps to be taken in the ANA development process.

2 Related Work and Background Information

This Masters thesis is accomplished in the ANA project, which builds a new, autonomic network architecture. Therefore we present in the first part of this section some projects related to either autonomicity or to new network architectures.

A focus of this Masters thesis is to provide a “bootstrapping process” during which different nodes get to know the features of other nodes in the same network. Some conventional protocols with this ability are described in the second part.

In the ANA implementation there are a lot of different processes which have to communicate together. Therefore we provide in the third section a summary of inter process communication (IPC) mechanisms. Since each process may reside in Linux user space or in Linux kernel space we do not only consider well know IPC mechanisms as UNIX sockets, but also IPC mechanisms between user space and kernel space applications.

2.1 Networking Projects

There are several projects related to developing new network architectures. The EU founded different projects in the area of autonomic networking, most of them are part of the sixth Framework Program, more precisely of the FET (Future and Emerging Technologies) [8]. The goal of FET is to identify possible areas for long-term, foundational, high risk and visionary research in communications. Among these projects are: “BIONETS” [32], “Haggle” [33], “CASCADAS” [34] and also “ANA” [35]. All of these projects cover a slightly different area of autonomic networking.

An important player in the field of autonomic computing is IBM [45]. IBM has its own research group in autonomic computing and provides different software products in the area of self managing autonomic computing.

In the US there is a big effort on building a network infrastructure called GENI [10]. The aim of GENI is to allow researchers to experiment with completely new network architectures under real networking conditions.

A slightly less ambitious goal is pursued by Click [39]. Click does not want to reinvent the Internet, but it provides a software architecture in which different elements of a router can be easily exchanged and therefore different elements and various router configurations can be tested.

IBM

IBM focuses its research on autonomic computing. It has defined the self * properties needed for a selfmanaging system (self-configuring, self-optimizing, self-healing and self-protection). In order to enforce the research activity in autonomic computing IBM plans to host and sponsor conferences, develop an Advisory Board and begin funding research on autonomic computing through awards and fellowships [45].

Ambient Networks Project

"The Ambient Networks project is (...) developing innovative mobile network solutions for increased competition and cooperation in an environment with a multitude of access technologies, network operators and business actors. It offers a complete, coherent wireless network solution based on dynamic composition of networks that provide access to any network through the instant establishment of inter-network agreements. The concept offers common control functions to a wide range of different applications and access technologies, enabling the integrated, scalable and transparent control of network capabilities." [30]

Bison

"BISON draws inspiration from biological processes and mechanisms to develop techniques and tools for building robust, self-organizing and adaptive NIS (Network Information Systems) as ensembles of autonomous agents. What renders this approach particularly attractive from a dynamic network perspective is that global properties like adaptation, self-organization and robustness are achieved without explicitly programming them into the individual artificial agents. Yet, given large ensembles of agents, the global behavior is surprisingly adaptive and can cope with arbitrary initial conditions, unforeseen scenarios, variations in the environment or presence of deviant agents. This represents a radical shift from traditional algorithmic techniques to that of obtaining the desired system properties as a result of emergent behavior that often involves evolution, adaptation, or learning." [31]

BIONETS

"The motivation for BIONETS comes from emerging trends towards pervasive computing and communication environments, where myriads of networked devices with very different features will enhance our five senses, our communication and tool manipulation capabilities." "BIONETS overcomes device heterogeneity and achieves scalability via an autonomic and localized peer-to-peer communication paradigm. Services in BIONETS are also autonomic, and evolve to adapt to the surrounding environment, like living organisms evolve by natural selection." [32]

Haggle

"Haggle is a new autonomic networking architecture designed to enable communication in the presence of intermittent network connectivity, which exploits autonomic opportunistic communications (i.e., in the absence of end-to-end communication infrastructures). We propose a radical departure from the existing TCP/IP protocol suite, completely eliminating layering above the data-link, and exploiting and application-driven message forwarding, instead of delegating this responsibility to the network layer. We use only functions that are absolutely necessary and common to all services, but that are sufficient to support a large range of current and future application." [33]

CASCADAS

"The overall goal of CASCADAS is identifying, developing, and evaluating architectures and solutions based on a general-purpose component model for autonomic communication services; specifically in such context autonomic service components autonomously achieve self-organization and self-adaptation towards the provision of adaptive and situated communication-intensive services. In other words, the project is driven by the ambition of identifying a fundamental, uniform abstraction for situated and autonomic communication entities, at all levels of granularity. This abstraction is called an ACE (Autonomic Communication Element), and it represents the cornerstone of the component model, in which the four driving scientific project principles (situation awareness, semantic self-organization, self-similarity, autonomic component-ware) will properly converge." [34]

GENI

GENI (Global Environment for Networking Innovations) [10] will give scientists a clean slate on which to imagine a completely new Internet that will likely be completely different from the one of today. GENI will consist of a collection of physical networking components, including a dynamic optical plane, forwarders, storage, processor clusters, and wireless regions. These resources are collectively called the GENI substrate. On top of the substrate, a software management system for GENI will layer experiments on the substrate.

Before GENI can be built it will take a few years of development and the NSF (National Science Foundation (of the US)) has to approve the founding.

Click

Click [39] is a software architecture for building configurable routers. A click router is assembled from many small elements which process the packets. With the help of a configuration file the desired elements can be put together in the order desired. This enables facile testing of different elements and different configurations.

A click router can be built in user space, in the Linux kernel or in the network simulator ns2. In order to build click in the kernel, the Linux kernel source code needs to be patched and recompiled. In kernel space there may exist only one click router which overwrites the standard Linux kernel packet processing. In userspace however it is possible to have the click router as well as the standard Linux routing in parallel.

2.2 Bootstrapping

A basic problem in networking is to initiate communication between different nodes. Before communication can start, nodes have to know the existence of each other and how they can reach other nodes.

The bootstrapping phase consists usually of three different steps:

1. A user wants to connect to a certain service. He either knows the address of it (e.g. the URL of a web server) or he chooses from a list the most

appropriate entry. This list can be for example the result of a search query in a web search engine or a list provided by the operating system with all available printers. This list is either written in a file or it is dynamically created with the help of a negotiation protocol.

2. The node has to learn the network address of the target node. This information can either be written in a file or it can be discovered with the help of some protocols, as for example DNS for the translation of domain names into IP addresses.

3. The application has to know how it can send a packet to the wire and a device driver has to know how to handle an incoming packet. In today's networks this is accomplished with the use of the TCP/IP protocol stack. In this protocol stack every protocol header specifies a next header field (or a port number) which indicates the next higher protocol. The values of these fields are hard coded, therefore it is immediately clear to which application a specific packet belongs.

This section discusses some of the approaches to discover other nodes or services autonomically.

AppleTalk

AppleTalk [11] is a protocol suite conforming to the OSI layer model. It was developed in the early 1980s. The main goal was to provide a facility to easily share resources, such as printers or files in a local network.

One protocol of the AppleTalk protocol suite is the Name-Binding Protocol (NBP). It is used to make applications available to other applications across the network.

Applications are identified by names, but addressed with numeric identifiers. The Name-Binding protocol provides a way of translating the names of applications into their addresses.

Each application (entity) can assign itself a name. In addition to a name, an entity can also have certain attributes. Each node maintains a table containing name-to-address mappings of all entities of that node. Any entity can enter its name and socket number into the name-to-address table to make itself visible by name. The union of the name-to-address tables of the network is called the name directory. It is a distributed database of name-to-address mappings.

Before a named entity can be accessed over an AppleTalk network or internet, the address of that entity must be obtained, by sending a query for the corresponding name.

NetBIOS, SMB, Browsing for Servers in Windows

NetBIOS [12] is used by Microsoft Windows to allow applications on separate computers to communicate over a local area network. NetBIOS may run over Ethernet, TCP/IP or IPX.

NetBIOS names represent a flat name space. Therefore NetBIOS packets cannot be routed, and hence TCP/IP is often used to send NetBIOS messages. In order to obtain a unique name, the Name Management Protocol broadcasts a system's intention for a new name to the network. If no other system objects,

the name is registered.
Names may be resolved into IP addresses either with the help of WINS (Windows Internetworking Name Server), with a statically configured file or the query is broadcasted. The latter cannot be used if client and server are on different network segments, since broadcast messages are stopped from routers.

The browsing functionality from Windows systems allows users to browse for other servers (e.g. in the "Network Neighborhood" environment). This browsing functionality is implemented with the Server Message Block protocol (SMB) which in turn uses NetBIOS.

On each NetBIOS network one machine will be elected to function as a "domain master browser" (DMB). The DMB contacts each Windows machine and exchanges the "browse list contents" with it. This way, all Windows machines will eventually obtain a complete list of all machines that are on the network [13].

Zeroconf

Zeroconf [15] is an IETF working group chartered in September 1999. The goal of zeroconf is to develop a protocol suite which is guided by the following principles:

1. Allocation of IP addresses without a DHCP server.
2. Resolution of hostnames to IP addresses without a dns server.
3. Autonomous service discovery in a network without a centralized directory server.

It is targeted to the scenario where a few nodes are connected and form a local network.

These goals are already solved by NetBIOS for Windows and AppleTalk for Mac. However the interconnection from Windows and Mac PCs in the same LAN still needs human intervention. Therefore zeroconf wants to develop an open and easy to implement protocol.

The working group has developed the protocol IPv4LL, which chooses a random IP Address from the 169.254.255.255 address range [2]. With the help of the ARP protocol (ARP probe and ARP announcement packets) it is verified that the address is not used by another host in the network. The resolution of hostnames is done with the help of multicast DNS [24]. There is a dedicated toplevel domain .local which is used for link local discovery only. Service discovery is also solved by using the DNS system. DNS Service Discovery (DNS-sd) [16] is a way of using standard DNS programming interfaces, servers, and packet formats to browse the network for services. There is a special DNS record in which services can be described. It may run with the "normal" DNS system or the multicast DNS system.

There are several implementations of the zeroconf protocols: Rendezvous as well as its successor Bonjour [17] have been developed by Apple and are also available for Linux, Solaris and Windows. Most printers also support the Bonjour protocol and can therefore be found by a client without interaction of

[2] This address range is dedicated for local networks.

the user. Avahi [18] is an implementation of the DNS service discovery and multicast DNS specifications for Zeroconf networking for Linux.

UPnP

The goal of UPnP [25] (Universal Plug and Play) is to enable devices to be automatically configured when they are connected to a network. The devices advertise their capabilities and they are controllable from other devices.

UPnP networks consist of two types of nodes: Controlled devices and control points. Controlled devices take the role of servers, answering the requests of control points. UPnP has the following steps:

1. The first step in a UPnP network is the discovery. A controlled device announces its service to the control points and control points look for devices of interest.
2. The second step is the description. The control point retrieves a detailed description of the discovered device. This description is expressed in XML.
3. In a third step the control point can send control messages to the device.
4. The forth step is eventing: A device may provide updates on some variables, which are received from the control point.
5. The fifth step is presentation: The device may provide a presentation page, which can be shown in a browser and over which a user can interact with the device.

BOOTP, DHCP

The Bootstrap Protocol (BOOTP) [22] as well as the Dynamic Host Configuration Protocol (DHCP) [23] serve the purpose of integrating a newly booted computer in a network. The newly booted computer has to learn its IP address and probably some other parameters like the network mask, the address of the dns server etc. In both protocols a centralized server is used for this task. However, before a client can send a query to this server, the servers address has to be found. This is done by broadcasting a DHCPDISCOVER query to the network. This query is answered by the BOOTP or DHCP server respectively and contains an IP address. In order that not each subnet has to offer its own DHCP server there is one DHCP relay agent in each subnet. Its task is to forward DHCPDISCOVER queries to the DHCP server.

DNS, Multicast DNS

The Domain Name System is used to resolve domain names into IP addresses. There exists a large server infrastructure in which each server is responsible for a certain subdomain. The entries in this servers have to be managed manually. In order to avoid this management overhead, multicast DNS [24] was designed. It describes how DNS queries have to be handled in order that they

can be resolved without the need of a DNS server. This is especially useful in spontaneously built networks, in which no DNS server is available and the participants are not interested in resolving global domain names, but only the once in the current network. Multicast DNS defines a top level domain .local which is used for the local naming of services.

Jini

The purpose of the Jini [20] architecture is to federate groups of devices and software components into a single, dynamic, distributed system. Jini is an extension to the Java technology.

In Jini, a service is an entity that can be used by a person, a program, or another service. A service may be a computation, storage, a communication channel, a software filter, a hardware device, or another user.

The heart of the Jini system is a trio of protocols called discovery, join, and lookup, which provide the facility to find other services.

- Discovery occurs when a service wants to make itself available on the network. The service provider locates a lookup service by multicasting a request on the local network for any lookup service.
- Join occurs when a lookup service has been found. A “service object” for the service is loaded into the lookup service. This service object contains the Java programming language interface for the service along with other descriptive attributes.
- Lookup occurs when a client or user needs to locate and invoke a service. The desired service is described by its interface type (written in the Java programming language) and possibly other attributes. Having sent the query to a lookup service, the corresponding service object is loaded into the client.
- The final stage is to setup the communication between the client and the service using Java Remote Method Invocation.

Jini was developed by Sun Microsystems but now runs under the Apache 2.0 license. It is also know as Apache River.

Webservices

Webservices provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. Webservices are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions thanks to the use of XML.

In order to make a webservice available there are three approaches: [14]

- The Registry Approach: Each service has to register itself explicitly in a directory.
- The Index Approach: Each service provides its information to the network. The index owner collects the description of the webservices and makes them available for lookup queries. This approach is similar to web search engines.

- Peer-to-Peer Discovery: At discovery time, a requester agent queries its neighbors for a suitable webservice. If any one of them matches the request, it replies. Otherwise each queries its own neighboring peers and the query propagates through the network until a particular hop count or other termination criterion is reached.

Summary

There are many protocols to automatically detect other nodes or other services in a network. In order to detect other services, most of them assume the existence of an IP network. There are two conceptually different methods to learn about other services:

1. Usage of a centraliced server. This server may actively look for content to save, or devices have to register explicitly with the server. Before the actual service discovery can start, the centraliced server has to be found, which is done by either statically predefining the address of the server, or by broadcasting a discovery message. Examples in this category are DHCP, jini, the windows browsing facility with WINS etc.

2. Distributed data base. There is no server, instead queries are broadcasted or multicasted over the network. These queries are answered by all the participating nodes together. Examples are the protocols from zeroconf, mDNS and DNS-sd, appletalk, NetBIOS over Ethernet etc.

Some of the protocols discussed provide both approaches. Depending on the current situation the most suitable method can be chosen. To this category belong e.g. webservices or the windows browsing facility, if it is configured correspondingly.

All presented protocols are dedicated for a certain application or protocol. This is in contrast to the ANA bootstrapping mechanism presented in section 5. This bootstrapping protocol can be used by every ANA networking element and by every ANA application.

The bootstrapping process we will present consists of two phases:

1. Unlike legacy applications which make use of the TCP/IP protocol stack, an ANA node cannot relay on a predefined protocol stack. Therefore the functional blocks have in a first step to build a dynamic protocol stack. This is accomplished with the help of a node local repository in which each functional block can publish and lookup some entries.

2. A functional block wants to learn about services on an other ANA node. Thereto “discovery queries” are broadcasted over the Ethernet segment.

2.3 Inter Process Communication Mechanisms

In ANA, the MINMEX as well as each functional block are implemented as individual processes. No two functional blocks communicate directly together, but all the communication goes over the MINMEX. Therefore there is the need of an inter process communication (IPC) facility between the functional blocks and the MINMEX. This communication should be as flexible as possible. The functional blocks as well as the MINMEX may reside in Linux kernel space or in user space or even on different physical devices. To allow virtualization it should also be possible that multiple MINMEX reside on one physical device.
This section gives a short summary over the different IPC mechanisms evaluated for the communication between the MINMEX and a functional block.
The following list describes the inter process communication concepts under consideration along with their advantages and drawbacks. The list is divided into four blocks, depending on where the two processes may reside.

USER-USER, KERNEL-USER, KERNEL-KERNEL

UDP Sockets	+	Allows communication from one PC to another PC
	+	Same communication for PC intern communication as for PC to PC communication
	-	IP/UDP has to be implemented which contradicts the assumption of a clean slate approach
	-	Quite a big control overhead (header, checksum etc.)

User-User

Shared Memory	+	Possible on every system, even the smallest
	-	Inconvenient to program
Unix Sockets	+	Similar programming for PC intern communication as for PC to PC communication (with UDP sockets)
	+	Less overhead than UDP sockets
	+	No assumption on existence of network protocols
Message Passing	+	Receiver can influence receiving order, search for a specific message type etc.
	+	Structured data
Pipes	+	Less overhead than UDP sockets
	-	No multicast possible

Kernel-User

Netlink Sockets	+	Use of well known socket interface in user space
	+	Communication can be started from the kernel
	+	Default kernel/userspace interface for networking
	+	Can be used for kernel/kernel communication as well
	-	Modification of the kernel source code needed (registration of a protocol type)
Generic Netlink	+	Makes use of netlink sockets
	+	Protocol registration is done at runtime (kernel source code needs no modification)

	+	Provides some degree of security, as the data type of the argument can be specified (integer, string etc., but also nested types)
sysfs, procfs	+	Allow to transmit a lot of data
	-	No immediate feedback provided
	-	Kernel cannot initiate a communication
syscalls	-	Modification of Linux source code needed
	-	Each system call has to have its own name and its own number registered in the Linux kernel source code
	-	Kernel cannot initiate a communication

Kernel-Kernel

Shared Memory	+	Possible on every system, even the smallest
	-	Inconvenient to program
Export functions	+	Normal way modules are taking with each other in the Linux kernel
	+	No IPC mechanism is needed
	-	Each function exported has to have a unique name

Summary

For the implementation of the ANA core communication, shared memory was dismissed from the beginning. The ANA prototype will run on Linux and therefore there exist more elaborate communication possibilities.
The user space application provides named pipes and UNIX sockets.
For the user space - kernel space communication, generic netlink is preferred over the standard netlink sockets, since they do not require the modification of the Linux kernel source code. Generic netlink sockets are discussed in section 4.1.2.
The kernel implementation provides a communication mechanism where functions are exported. Instead of exporting all functions a more elaborated design was developed, with which the number of exported functions is constant for an arbitrary number of functional blocks and MINMEXs. Section 4.1.3 discusses this design.
Since UDP sockets are the most flexible communication mechanism they are supported in user space as well as in kernel space. The implementation of the UDP communication mode in the Linux kernel space is discussed in section 4.1.1.
The other possibilities were dismissed either because they require the modification of the Linux kernel source code or because they have a big overhead.

3 The Architecture of the ANA Project

In this chapter we introduce the network architecture proposed by the ANA project. We start with a summary of the ANA Blueprint [28] in which the high level design principles are defined. In the second part we discuss how these design principles are implemented in the ANA core software.

3.1 Blueprint

The Blueprint focuses on the overall architectural aspects of ANA. It defines the basic abstractions and building blocks of ANA and it presents their basic operation and interactions. The Blueprint does not deal with autonomicity, but with the architecture which allows to implement autonomicity in a second step. In this section we describe the basic building blocks and the communication concepts of ANA. Please refer to the ANA Blueprint for a more detailed discussion.

3.1.1 Terminology

This section summarizes the keywords necessary to be able to follow the descriptions in the ANA Blueprint.
Figure 2 shows the structure of an ANA node with its main components:

- **MINMEX**: (Minimal INfrastructure for Maximal EXtensibility): The MINMEX defines the common denominator among ANA nodes, and must be present in all implementations. It provides the basic low level functionality which is required to bootstrap and run ANA.
- **Playground**: The playground is the execution environment where the more elaborated and complex networking functionality of ANA is placed. The playground hosts the optional protocols that one is free to develop with the help of the functionality provided by the MINMEX elements.
- **Hardware Abstraction Layer**: The hardware abstraction layer provides generic access to the hardware upon which ANA is executed.

The following list summarizes the most important elements of the ANA architecture. The relation of *Functional Blocks*, *Information Dispatch Points* and *Information Channels* is depicted in Fig. 3.

- **Functional Block (FB)**: Functional blocks are the information processing units of ANA. They generate, consume, process or forward information. A functional block runs on exactly one node. A functional block might consist of other functional blocks and resides in the playground of an ANA node. (Note, in the ANA Blueprint functional blocks are sometimes called clients).
- **Information Channel (IC)**: Information channels are an abstraction for communication channels. FBs communicate over ICs.
- **Information Dispatch Point (IDP)**: Information dispatch points are access points to FBs or ICs. The MINMEX manages the association of a

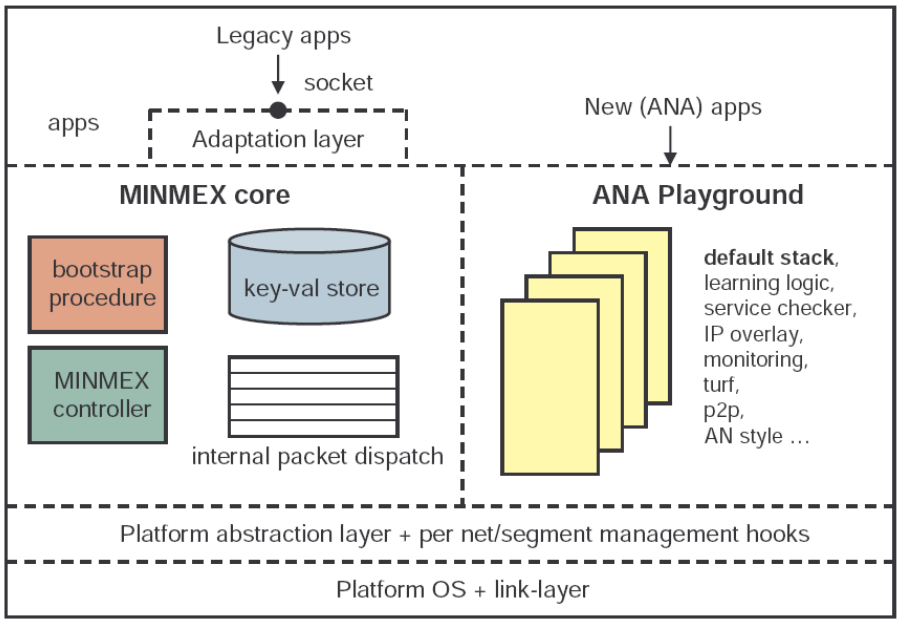


Figure 2: Main components of the ANA-node (src: ANA Blueprint [28]).

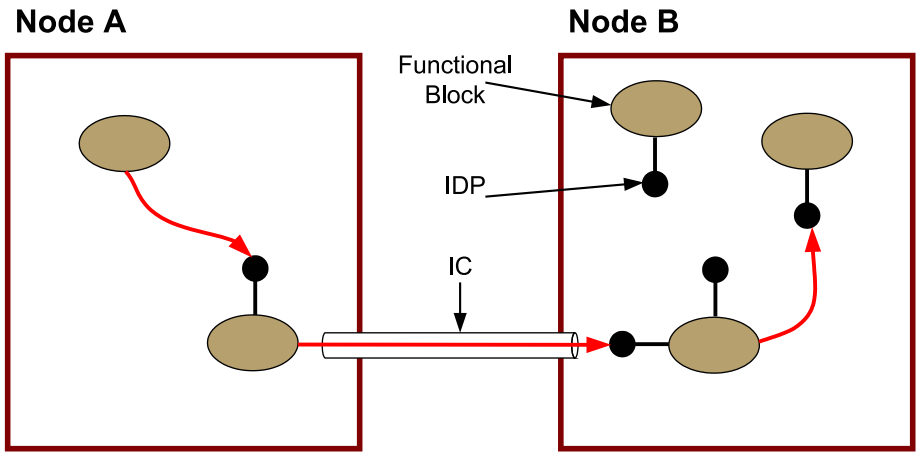


Figure 3: Data flow through Functional Blocks (FB), Information Dispatch Points (IDP) and an Information Channel (IC).

given IDP to a FB or IC. For example, the MINMEX is able to exchange a functional block behind a given IDP. This leads to the possibility to exchange functionality, whereas leaving the address constant. For example an encryption routine could be changed, without that a functional block using the encryption routine would have to change anything.

- **Label**: A label is a node local identifier for an IDP.
- **Compartment**: A compartment is a policed set of FBs, IDPs and ICs with some commonly agreed set of communication principles, protocols and polices.
- **Membership database**: The membership database stores information about available network compartments, functionality provided by other clients and functionality provided by the ANA software.

3.1.2 Compartments

The concept of compartments allows the division of communication networks into smaller units. The boundary of a compartment can be based on technological or administrative boundaries. Compartments using different technologies can communicate together through a common overlay compartment. They provide both: hiding of compartment internals form the outside world and hiding of communication complexity from its members.

Each compartment has to provide the following key functions:

- Registration and De-registration: The registration function assures that only admissible communication elements become a member of the compartment. It can be either some kind of authentication function or it can be completely open.
- Identifier management: Typically some kind of identifier is assigned to a member during registration phase. A resolution function has to be provided to look up an identifier and to obtain the information necessary to communicate with the requested member.
- Routing: The routing function is responsible to select a communication path inside the compartment.

Node Compartment

The node compartment is a special compartment which is present in each ANA node. The MINMEX as well as each functional block residing on one node belong to the node compartment of this node. The node compartment provides mechanisms for functional blocks to make themselves visible for other functional blocks on the same node. It provides also the possibility to search for a functional block which provides a certain functionality. Any FB or IDP belongs to exactly one node compartment. Therefore the node compartment has full control over the FBs and IDPs. The node compartment may provide different views of the actual ANA node to different functional blocks. Thereby it can hide some functionality from some functional blocks, or define some obligations for certain functional blocks.

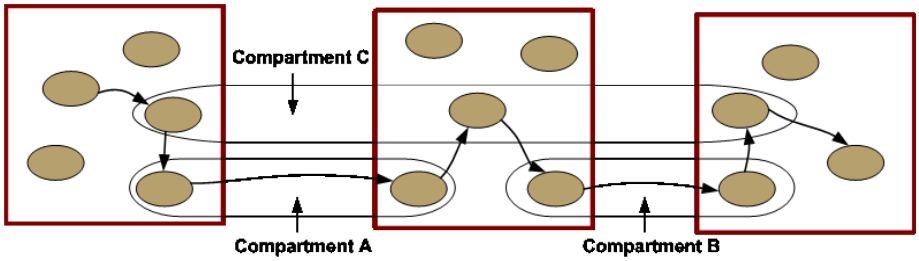


Figure 4: Layering of compartments in ANA.

Network Compartment

A network compartment encompasses several nodes and involves communication across an underlying network infrastructure. Each node which wants to belong to a given network compartment has to implement the required network stack. By providing the network functionality of different compartments an ANA node may connect to different network compartments. The communication between different nodes is provided by ICs and is typically accomplished over a physical link (but it may also be a virtual one). Network compartments can be combined in a flexible way. A network compartment may combine the functionality of several OSI layers, or it may provide only a part of an OSI layer. To allow communication between different network compartments they may be ”layered”. Therefore a communication between two nodes may traverse several network compartments. Figure 4 shows the layering of different compartments.

3.1.3 MINMEX

The MINMEX is a minimal component required for all nodes running ANA. The MINMEX offers the following services:

- Information Dispatch Framework (IDF):
  Data packets are always sent to an IDP which is bound to a functional block or an information channel. This allows to redirect packet flows without the sender being aware of this. Data is forwarded from one IDP to the next on a hop by hop basis. The IDF has to dispatch data belonging to different compartments with different sorts of addressing and naming schemes. Therefore IDPs are identified by local labels which have no networking meaning. IDPs are stored in Information Dispatch Tables (IDTs). An IDT typically contains all the IDPs that belong to some communication context. Therefore it is possible to check which member has the right to change which IDP or to block members consuming excessive resources.

- Key-Val Repository (KVR):
  The key-val repository is a directory service to access communication mechanisms, protocols and compartments. The key-val repository is part of the membership database of the node. Clients may add entries

with self-chosen values which can later be retrieved via the standard resolve functionality. An ANA node could announce available services to its neighbors by putting entries in the neighbors key-val repository.

- Bootstrap Procedure (BP):
  The bootstrap procedure is used to detect other nodes and to activate compartments. In addition it loads the static configuration of the ANA node, which is typically maintained by a human user. It can also start more sophisticated service discovery programs, instantiate FBs and information dispatch points.

- MINMEX Controller (MC):
  The MINMEX controller monitors the basic operation of an ANA node. It is truly autonomic and basically it performs sanity and health checks of the components running inside the node. Its main goal is to protect the MINMEX elements from faulty components in order to guarantee the performance of the ANA node. The MC controls the forwarding paths and the state of the IDTs. It performs a garbage collection of unused IDPs and entries in the key-val repository. It controls the state of all functional blocks and it provides a low level of access control for the IDTs.

3.1.4 Summary

In this section we have presented the architecture of the ANA Project. In each ANA node there are two main components: On the one hand there is the MINMEX which is the minimal component to be provided from each ANA node and on the other hand there is the playground in which the real functionality (provided by the functional blocks) is hosted. We have introduced the concept of compartments which allows different networks to coexist.
For this thesis the most important terms are: MINMEX, functional block, and IDP.

3.2 Prototyping Phase: From Abstraction to Implementation

The ANA Blueprint is the basis for the development of the ANA prototype. This section describes the implementation decisions as discussed on the ANA core developer meeting in Basel on 14. March 07.
The ANA prototype will run on Linux. As a basic concept the implementation should not need any other software. This leads to a simple installation of ANA, since only the ANA software has to be installed. However it dismisses any architecture which would use some ”helper” software as e.g. click [39] or any kind of middle ware. The ANA prototype should work on three different ”platforms”:

- Linux user space
- Linux kernel space
- Network Simulator ns2

The goal of the implementation is to share as much code as possible for all platforms. An additional requirement for the kernel space implementation is that it should not touch the Linux kernel source code. This would impose the burden of patching the Linux kernel source code and recompile it to future ANA developers. The userspace part as well as the framework is implemented at the Computer Science Department at the University of Basel [27] whereas the kernel space part and the ns2 part are developed at the Computer Engineering and Networks Laboratory at ETH Zurich [26].
During this Masters thesis the kernel part was developed.

This section describes the design of the core functionality to be provided from each node which wants to participate in the ANA world.

3.2.1 Building Blocks

There are two main components in the ANA core:

- Bricks: A Brick is the most atomic element in ANA. It corresponds to a functional block as described in the ANA Blueprint. It can be part of a bigger functional block or of a compartment. Together all Bricks form the playground of an ANA node. Before a Brick can participate in ANA it has to attach itself to the MINMEX.
- MINMEX: In an ANA node Bricks interact through the MINMEX. The MINMEX allows to discover other existing Bricks in the ANA node. It dispatches messages between IDPs owned by different Bricks.

In user space the MINMEX as well as the Bricks are individual processes. In kernel space the MINMEX as well as each Brick run as kernel modules.

3.2.2 Communication Principles

The communication between the Bricks and the MINMEX is realized with so called communication gates. A communication gate is the abstraction of an

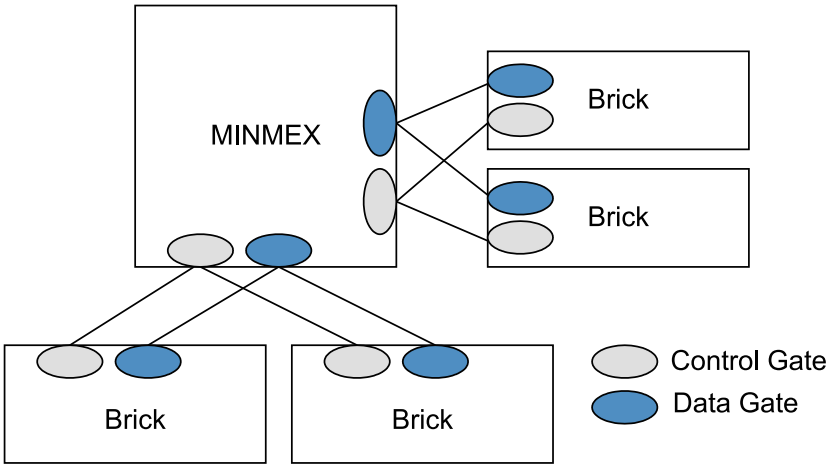


Figure 5: MINMEX - Brick interconnection with the control and the data gates.

inter process communication channel. Each MINMEX and each Brick has two different kinds of communication gates: one to receive data and one to receive control information. The MINMEX may provide several communication gates for data and for control information. This permits different types of Bricks to connect to the MINMEX (e.g. one could use UNIX sockets and another could use PIPES as a communication mechanism). Multiple Bricks are allowed to send messages to the same communication gate of the MINMEX. Figure 5 shows a possible setup between a MINMEX and multiple Bricks. Note that there is no direct communication channel between two Bricks. All communication is routed through the MINMEX.
The following paragraphs describe the three different kinds of messages to be exchanged between the MINMEX and a Brick:

Control
Control messages are used between a Brick and the MINMEX to attach a Brick to the MINMEX and to manage the handling of the IDPs. They are always initiated from the Brick to the MINMEXs control gate. The execution of the Brick is blocked until it has received the corresponding reply. The MINMEX processes the request and sends a reply message to the Bricks control gate. There are no unsolicited control messages for the Brick.

Data
Data messages are sent to the data gate of the destination. For both, the MINMEX and the Bricks, data messages can arrive asynchronously to the control flow. Data messages have the format $[destinationIPD][data]$. There is no reply for a data message.

Notification
The MINMEX can send unsolicited notification messages to a Brick (e.g. to inform the Bricks about a MINMEX which is about to go down etc.). These

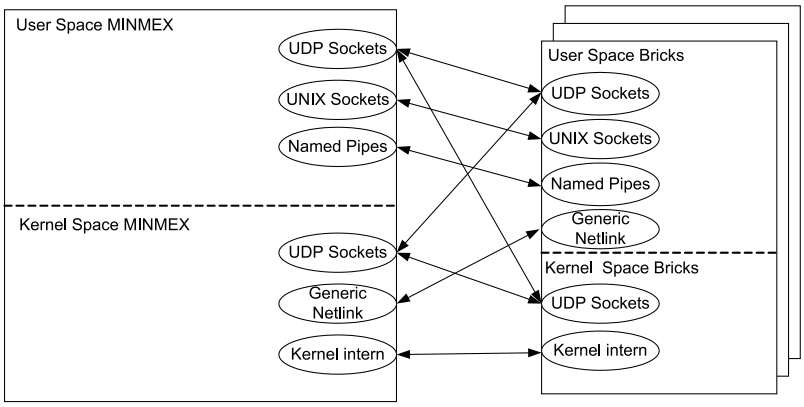


Figure 6: Overview over communication possibilities between the MINMEX and Bricks.

messages are sent asynchronous to the program flow of the Brick. The only possibility for a Brick to receive asynchronous messages is via its data gate. Therefore a Brick registers an IDP on the MINMEX to which the MINMEX can send notification messages. A received notification message is forwarded to the registered notification parsing function. There is no reply to notification messages.

Figure 6 shows an overview over the communication possibilities between the MINMEX and the Bricks. The current implementation provides the following communication mechanisms between MINMEX and Brick:

- UDP: Communication over UDP sockets
  can be used in all user space $\leftrightarrow$ kernel space configurations.
- UNIX sockets: Used for user space $\leftrightarrow$ user space communication.
- PIPES: Communication over named pipes, only used for user space communication.
- GENETLINK: Communication over generic netlink sockets, used for a MINMEX in kernel space and a Brick in user space.
- KERN: Kernel intern communication by direct function call, only used for kernel space communication.

The Linux kernel specific implementation of these communication mechanisms is discussed in section 4.1.

3.2.3 Important Data Structures

This section summarizes the tables and lists used to save the information necessary to operate an ANA node. All of them are depicted in Fig. 7. A more thorough discussion can be found in Appendix C where all the individual structures and lists are described in detail and in the ANA core documentation [48]

where those management units are explained for future ANA developers.

Brick Table
The Brick table belongs to the MINMEX. It holds the information about each Brick attached to the MINMEX. It is a hash table with a size defineable upon program start up. For each Brick it holds the addresses of its data and control gates and the notification label. This way, the MINMEX knows how to send data, control and notification messages to its attached Bricks.

Information Dispatch Table (IDT)
The IDT holds the information on the registered information dispatch points (IDPs) in the MINMEX. The IDT is used to map a label to a Brick. Upon the arrival of a labeled message, the IDT is consulted to know to which Brick this IDP belongs. With the help of the Brick table, the MINMEX knows how to forward the message to the correct Brick.

Shadow Dispatch Table (SDT)
Each ANA Brick has a hidden management unit called the Shadow Dispatch Table. The SDTs role is to map IDP labels to functions. When the Brick receives a data message, it checks its SDT for the given IDP label and it executes the function corresponding to the given IDP.

Figure 7 shows how a data message is routed from one Brick to another. The sending function (Brick C, Fct 1 on Fig. 7) has to specify the destination IDP. The MINMEX checks to which Brick this IDP belongs and how this Brick can be reached. Afterwards he forwards the message to the data gate of the Brick owning this IDP. The Brick checks in its shadow dispatch table which function corresponds to this IDP and finally it invokes the corresponding function with the given argument.

3.2.4 ANA API

This section describes the basic ANA API which is used from Bricks to communicate with the MINMEX. The ANA API is divided into different layers, each of which provides a different abstraction level.

- API Level -1: It defines the platform dependant communication mechanism between the Bricks and the MINMEX.
- API Level 0 (AL0): AL0 groups all functions that permit to access the MINMEX in a platform unspecific way. It consists of attachment and detachment functions, as well as a registration function for callback functions.
- API Level 1 (AL1): AL1 is a library of procedures for interacting with ANA compartments, ICs and FBs. It does not remove the need to manage communication with the MINMEX at AL0. But once attached there are procedures for creating requests and parsing replays.
- API Level 2 (AL2): At AL2 functionality can be accessed through ordinary method calls, object destruction runs automatically the necessary low level cleanup actions.

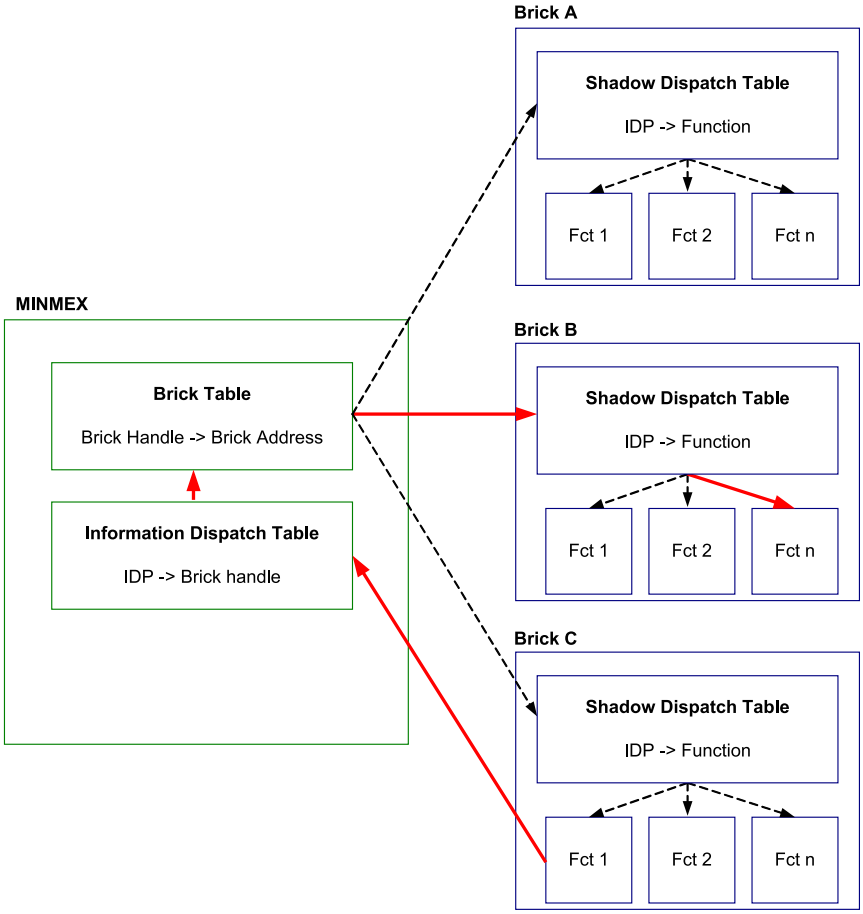


Figure 7: Path of a data message: A Brick sends the message to the MINMEX, the MINMEX looks for the correct destination Brick, the destination Brick invokes the function corresponding to the IDP.

At the point of writing, the API Level -1, AL0 and AL1 are implemented. The API Level -1 cannot be considered as an API, but rather it is the system specific implementation of the functions to be provided from API Level 0. The focus of the first part of this Masters thesis lies on the implementation of the API AL0 in the Linux kernel space. Therefore we shorty introduce the functions to be provided from AL0.

Functions to be provided from API Level 0
API Level 0 provides the basic functions which are needed for a Brick to interact with the MINMEX. All functions required are summarized in the following list:

- attach: establishes the link between a functional block an the MINMEX.
- detach: terminates the attachment of the Brick, all resources are freed.
- send: send data packets to a dispatch point.
- registerCallback: requests the ANA node compartment to create an IDP for the callback function. Any message sent to this IDP will result in a call of the registered function. Transmission is unreliable → packets may never arrive at the callback function.
- unregisterCallback: unbinds the callback bound to the given IDP, all resources get freed.
- redirect: registers a new callbackfunction for a given IDP.

3.2.5 Example Communication

Each Brick and each MINMEX has a dedicated channel for control and for data messages respectively. The MINMEX acts as a server which waits for requests and executes the corresponding commands. Once connected, the Brick as well as the MINMEX perform regular heart beat checks, to determine whether the other is still alive.

Attachment
A Brick which wants to become a member of an ANA node has to know the MINMEXs control interface (e.g. socket address, pipe etc.). It sends an attach message to that interface with the following format:
$[a][controlMode][controlAuxLength][controlAux]$
$[dataMode][dataAuxLength][dataAux][desiredMode]$
where the modes describe which kind of link the Brick wants to communicate over and the *Aux fields contain the corresponding addresses. Upon receiving an attach message the MINMEX performs the following actions:

1. Check whether it has the requested modes.
2. Generate a handle for that Brick.
3. Add the Brick to the Brick table.
4. Generate a label for the node compartment and put it in the IDT.

5. Compose the reply message and sends it back to the Brick over the control link specified.

The reply message has the following format:
$[r][handleSize][labelSize][maxMsgSize][handle]$
$[controlLabel][nodeDataMode][dataAuxLength][dataAux]$
Where the handle corresponds to the identifier of the Brick chosen by the MINMEX. The controlLabel refers to the node compartment label, labelSize indicates the size of all IDP labels (e.g. identifiers for the callback functions) and handleSize indicates the size of the handles (e.g. identifiers for the Bricks). maxMsgSize corresponds to the maximum message size allowed to send and dataAux specifies the address where the MINMEX is listening for data.

Registration
A Brick can send a register request for one of its callback functions to the MINMEX. The Brick sends a request message to the MINMEX which tells the MINMEX to

1. Generate a new label.
2. Add it in its IDT.
3. Return it to the Brick.

Thereafter the Brick generates a new entry in its SDT where it maps the label returned by the MINMEX to the actual callback function. This step complets the communication setup. The registered callback functions are now accessible by other Bricks, by sending a message to the IDP belonging to this callback function.

Data exchange
Data messages have the following format:
$[TargetIDP][Data]$
Each time the MINMEX receives a data message, it checks whether the indicated function (IDP) exists and whether the Brick is allowed to access it. Afterwards it forwards the data message to the Brick which owns the IDP. The Brick looks up the IDP in the shadow dispatch table and invokes the corresponding callback function. The data received is given as an argument to the callback function.

Detachment
The detachment of a Brick from the MINMEX either occurs explicitly with the exchange of a detach message, or implicitly with the help of a heard beat mechanism. Bricks and the MINMEX exchange in regular intervals heart beats. If no heart beat is received for a configurable amount of time the counterpart is assumed to be dead and all resources are freed.
In order to explicitly detach, either a Brick or the MINMEX may send a detach message to its counterpart. Upon receiving a detach message all resources belonging to that connection are freed.

3.2.6 Summary

In this section we have linked the ANA Blueprint with the actual implementation. We have introduced the term Brick which corresponds to a functional block in the Blueprint. We have described the communication between the MINMEX and the Brick, which makes use of the control gates and of the data gates. The lifecycle of a Brick consists of the following four steps: attachment, registration, data exchange and detachment. These functions have to be provided by the ANA API Level 0.

4 Implementation of the ANA Core in the Linux Kernel

This chapter describes the implementation of the ANA core in the Linux kernel. It starts with a description of the communication mechanisms between the Bricks and the MINMEX. This gives an insight in some fundamental concepts of the Linux kernel, such as interrupt handling or the delayed execution of tasks. We present a short performance evaluation of the implemented communication modes and conclude that the ANA node implemented in the kernel is by far the fastest. In the second section we present some helper functions to write system agnostic code. They allow the developer to write code which runs in user or in kernel space without having to care about the system specific implementation. In section 4.1 and 4.2 the description is based on a kernel 2.6. But the ANA core runs also on a kernel 2.4. Therefore we have summarized the most important changes from kernel 2.4 to kernel 2.6 in section 4.3.

4.1 Communication Between the MINMEX and the Bricks

This section describes in detail the three communication mechanisms “UDP sockets”, “generic netlink sockets” and the “export of functions” in the Linux kernel in general, and how they are used in ANA. The communication mechanisms described, are not visible to an ANA developer, but they are used in the implementation of the AL0 functions to pass the commands between the Bricks and the MINMEX.

The explanations are based on the kernel version 2.6.20. However, we have ported the code to run on every 2.6 kernel as well as late 2.4 kernels.

The MINMEX as well as each Brick are implemented as Linux kernel modules. The MINMEX in kernel space offers 3 different communication modes:

- UDP: Communication over *UDP sockets*. This communication mode can be used from any Brick.
- GENETLINK: Communication over *generic netlink sockets*. This will be used from Bricks living in user space.
- KERN: This communication mode enables the Brick to call the functions provided by the MINMEX directly. It can only be used from Bricks which are in kernel space.

4.1.1 UDP Socket Communication

This section discusses the Linux kernel socket API along with all its implications like callback functions, interrupt context and work queues. The second part gives an overview about how UDP sockets are used in the ANA core.

Linux Kernel Socket Interface

The Linux kernel provides a similar API for socket programming as is known from user space. The major difference lies in the identification of the sockets: in user space each socket is referenced with the help of a file descriptor,

whereas in the kernel each socket is referenced with a `struct socket`. Table 1 lists the most common kernel socket functions. A complete list can be found in the Linux kernel header file `include/linux/net.h`.

creation	**sock_create** (int family, int type, int proto, struct socket **res)
bind	**kernel_bind** (struct socket *sock, struct sockaddr *addr, int addrlen)
send msg	**sock_sendmsg** (struct socket *sock, struct msghdr *msg, size_t len)
recv msg	**sock_recvmsg** (struct socket *sock, struct msghdr *msg, size_t size, int flags)
setsockopt	**kernel_setsockopt** (struct socket *sock, int level, int optname, char *optval, int optlen)
close	**sock_release** (struct socket *sock)

Table 1: Linux kernel socket interface.

The system calls invoked from user space and the kernel socket API map to the same functions. In the Linux kernel it is assumed, that these functions are invoked from user space. Each function which has a pointer as an argument (e.g. `sock_sendmsg`) checks whether this pointer really belongs to a userspace address segment and if not it returns -EFAULT, which stands for “Bad Address”. To overcome this problem the memory area has to be adjusted with the function `set_fs()`. The argument is either `KERNEL_DS` or `USER_DS`, standing for kernel data segment or user data segment, respectively. After having executed the function, the old state should be restored. The adjustment of the memory area can be accomplished with the following code snippet.

```
m_segment_t old_fs = get_fs();
set_fs(KERNEL_DS);
/*call function here*/
set_fs( oldfs );
```

Receiving Data in Kernel Space

There are two different mechanisms to receive data from a socket in kernel space. One that is similar to user space and one that is kernel space specific. In the first mechanism the function `sock_recvmsg` is called, and the program execution gets blocked until the socket receives some data. As in user space a dedicated thread has to be started to be able to receive messages asynchronously to the program execution. However, unlike in user space, there is no `select()` function which would allow to wait on multiple sockets. Therefore this approach should only be used if the module wants to receive a message synchronously to the program flow and has to wait until the socket has received the message.
To receive messages asynchronously to the program flow, the second, kernel specific mechanism should be chosen: A function which should be executed

upon data reception is assigned to the socket. The Linux kernel invokes this "callback function" when data is received on the specified socket. The callback function has as an argument the `struct sock` upon which the data was received. The callback function is responsible to dequeue the message from the sockets receive message queue, to process the data and finally to free the memory occupied by the message. For TCP, the dequeued message holds the data without the header, whereas for UDP the data returned has the header still attached. This means, that the actual message starts 8 bytes later (2 bytes source port, 2 bytes destination port, 2 bytes length and 2 bytes checksum). The following code snippet shows a minimal callback function for a UDP socket:

```
void callback(struct sock *sk, int bytes){
        struct sk_buff * skb;
        skb = skb_dequeue(&sk->sk_receive_queue);
        printk("received data: %s with len %u \n",
               skb->data+8, skb->len-8);
        kfree_skb(skb);
}
```

A callback function can be assigned to a socket with the following code snippet:

```
struct socket *mysocket;
/*creation and binding similar to user space*/
/*register callback*/
mysocket->sk->sk_data_ready = callback;
```

The callback function is executed in interrupt context. Meaning that (1) all interrupts are masked and that (2) there is no process associated with the current execution. (1) imposes the need of a fast execution of the callback function, since otherwise no further interrupts may be handled. (2) imposes that the function is not allowed to sleep, since a process context is needed to reschedule the sleeping function.

Therefore the callback function has to be executed as fast as possible and especially it is not allowed to sleep in the callback function. A function may sleep under the following conditions:

- Execution of a sleep function (e.g. `msleep()`).
- Call of a function which may sleep itself (e.g. `sock_sendmsg()`).
- Waiting for the hardware to be ready.

If a process is in interrupt context the task of processing some data is usually split in two parts:

1. A first part which prepares the data (top half).
2. A second part which processes the data (bottom half).

The top half is executed in interrupt context where all interrupts are disabled. Therefore it only does some basic processing and passes the received data to

the bottom half which does the real processing of the data. The Linux kernel offers different mechanisms to implement bottom halves:

- Kernel timers
- Tasklets
- Work queues

Kernel timers as well as tasklets are executed in “soft interrupt context”, meaning that all hardware interrupts are enabled, but that there is no backing process assigned. This implies that kernel timers as well as tasklets are not allowed to sleep, in contrast to work queues, which are executed in process context. Each work queue has one or more dedicated processes (e.g. kernel threads) which run the functions submitted to the queue. This allows that the function executed may sleep, since there is a process context assigned to the function which is needed to reschedule the function after a sleep.
Work queues allow kernel code to request that a function may be invoked at a later time. It can be requested that the function is executed as soon as possible or after a predefined amount of time. There exists a default work queue which can be used from anywhere in the Linux kernel. Apart from the default work queue each kernel module can create its own work queues. Since a lot of different functions can be registered to the default work queue, it should only be used for tasks running a short period of time, and when the tasks are not time critical.
Functions to be executed from the work queue are called work queue handler functions. Only one instance of a work queue handler function can be inserted in the work queue at any given time. This has an impact when the work queue is used in combination with the socket API. Since it is unknown how many packets arrive on a socket before the work queue gets executed, the socket callback function should not dequeue the packet from the sockets receive message queue. Instead it triggers only a work queue handler function. This function is responsible to dequeue all messages in the sockets receive message queue.

A work queue handler function has to have a predefined interface:

```
void workQueueFunction(struct work_struct *myWorkStruct);
```

A drawback of this interface is the lack of a parameter to supply data to the work queue handler function. But this can be solved with the following trick: A wrapper struct has to be created, which has as one of its entries the `struct work_struct`. The other entries can be customized according to the functions needs. Such a wrapper function may look as the following:

```
struct wq_wrapper{
        struct work_struct worker;
        struct sock * sk;
};
```

Upon receiving a message the top half fills in the customized fields and inserts

the work queue handler function in the work queue. When the work queue handler function gets executed it determines the base address of the `wq_wrapper` struct. The Linux kernel provides therefore the following macro:

```
container_of(ptr, type, member);
```

where `ptr` points to the known element (e.g. `myWorkStruct`) , `type` indicates the struct to which `ptr` belongs (e.g. `wq_wrapper`), and `member` is the name of `ptr` in the struct (e.g. `worker`). The `container_of` macro returns the address of the whole struct.

The following example shows the whole process from receiving a packet until freeing the corresponding socket buffer.

```
struct wq_wrapper wq_data;

/* the sockets callback function
 * triggered each time a message arrives on the socket
 * referenced by sk
 * queue_work fails if the work queue handler function
 * is already in the work queue.
 */
static void callback(struct sock *sk, int bytes){
        wq_data.sk = sk;
        queue_work(wq, &wq_data.worker);
}

/* the work queue handler function
 * first it determines the base address of the wq_wrapper struct
 * then it dequeues packets as long as there are any in the queue
 * having processed a packet it frees the socket buffer
 * skb_queue_len returns the number of packets in the recv queue
 */
void workQueueHandler(struct work_struct *data){
        struct  wq_wrapper * foo = container_of(data,
                                    struct wq_wrapper, worker);
        int len = 0;
        while((len = skb_queue_len(&foo->sk->sk_receive_queue))>0)
        {
                struct sk_buff *skb = NULL;
                skb = skb_dequeue(&foo->sk->sk_receive_queue);
                printk("data %s,len %i", skb->data+8, skb->len-8);
                kfree_skb(skb);
        }
}
```

Use of UDP Sockets in the ANA Core

The **ANA MINMEX** provides two distinct interfaces: one for control and one for data messages. There may be multiple instances for each of these interfaces for different IP address/port pairs. Since in the MINMEX control as well as data messages may arrive at any given time, the execution model with the callback function was chosen.
In the next few lines we introduce the function prototypes used for handling UDP sockets in th MINMEX. The numbers in brackets refer to Fig. 8.
The MINMEX provides two socket callback functions: one for data and one for control reception respectively (independently of the number of UDP data and control gates specified). For each data gate specified upon module insertion function (1) is registered as a socket callback function, and for each control gate function (2).

Upon receiving a control message the MINMEX does some processing and sends an answer back to the originator. Data messages may be forwarded to another Brick. Both of these actions lead to a call to `sock_send`. This function needs to access the network card and it may therefore sleep. Since the callback function is executed in interrupt context as discussed in the last section, the data processing has to be split in a top and in a bottom half. For each MINMEX there exists a dedicated work queue which is responsible for the execution of all work queue handler functions of this MINMEX. This solution is preferred over the solution with the kernel wide work queue, since there is less interference with the world outside of ANA.
The only task of the callback functions (1) and (2) is to insert the work queue handler function (3) and (4) in the work queue. The work queue handler functions dequeue the messages from the given socket as discussed in the last section. They determine the actual data and the length of the data and invoke the data or control parsing functions (6) and (7) respectively. These functions do the actual parsing of the message. They check for the type of the message (e.g. attach, register a callback function, send data to an IDP etc.) and forward the message to the corresponding function. This function performs the operation requested and sends finally a confirmation back to the Brick or forwards the data message to the correct Brick. Since these functions are common to all communication mechanism for kernel and user space they are not further discussed here. After the `parseData` or `parseControl` function returns, the socket buffer gets freed from the work queue handler function.

The **ANA Brick** has one control interface and one data interface for each MINMEX it wants to be connected to. The handling of the control communication is simple, since the Brick always initiates the communication and waits until it receives an answer. Therefore there is no need of a callback function, but the answer can be received with a call to `sock_recvmsg()`. This call may block, but this is allowed since we are not in interrupt context and since the execution of the program has to wait until the response is received.
Data messages can be received asynchronously to the control flow of a Brick. Therefore the Brick registers a callback function for the socket on which it listens for data messages. The steps required are analogously to the one in the MINMEX for receiving data and control messages and they are therefore not explained here.

```
static void cb_data(struct sock *sk, int bytes);            (1)
static void cb_control(struct sock *sk, int bytes);         (2)
void parseDataTemp(struct work_struct *data);               (3)
void parseControlTemp(struct work_struct *control);         (4)
struct wq_wrapper{                                          (5)
        struct work_struct worker;
        struct sock * sk;
};
void parseData(char * buff, int len);                       (6)
void parseControl(char * buff, int len);                    (7)
```

Figure 8: Function prototypes for the handling of UDP sockets in the MINMEX.

An interested reader is referred to the source code available at [47]. The following files are of special interest:

- `C/bricks/AL0/kernel/KanaLib0.c`
- `C/bricks/AL0/common/anaLib0.c`
- `C/minmex/kernel/kernel_minmex.c`

4.1.2 Generic Netlink Communication

This section gives an introduction to netlink sockets and explains the steps necessary to work with generic netlink sockets. The last paragraph gives an overview over the generic netlink sockets as they are used in the ANA core software and it introduces a small user space library to simplify the handling of generic netlink sockets.

Introduction to Netlink Sockets

Netlink is a special IPC used for transferring information between kernel and user space processes. Netlink provides a full-duplex communication link between the Linux kernel and user space. It makes use of the standard socket APIs for user-space processes and a special kernel API for kernel modules. Netlink sockets use the address family `AF_NETLINK`, as compared to `AF_INET` used by a TCP/IP socket. Both `SOCK_RAW` and `SOCK_DGRAM` are valid values for `socket_type`. However, the netlink protocol does not distinguish between datagram and raw sockets. Each entity using netlink sockets has to define its own protocol type (family) in the kernel header file `include/linux/netlink.h`. Currently (for kernel 2.6.20) there are 17 different netlink families registered with the Linux kernel. Among them are:

- `NETLINK_ROUTE`, used for the communication between user space routing daemons and the kernels packet forwarding module.

- NETLINK_NFLOG, used as a communication channel between the user space iptable management tool and kernel space Netfilter module.
- NETLINK_IP6_FW, used to transport IPv6 packets from Netfilter in the Linux kernel to the user space.
- NETLINK_GENERIC, generic netlink interface, to be used from different applications.

Netlink sockets have the following advantages against other communication mechanisms for user space / kernel space communication as summarized in [36]:

- It is simple to interact with the standard Linux kernel as only a constant has to be added. There is no risk to pollute the kernel or to drive it in instability, since the socket can immediately be used.
- Netlink sockets are asynchronous as they provide queues, this means that they do not disturb the scheduling of the Linux kernel. This is in contrast to system calls which have to be executed immediately.
- Netlink sockets provide the possibility of mullticast.
- Unlike systemcalls netlink sockets allow to initiate a communication form kernel space.
- They have less overhead (header and processing) compared to standard UDP sockets.

Beside these advantages netlink sockets have also some drawbacks:

- The Linux kernel has to be modified and therefore to be recompiled. A task that many users are afraid of.
- The maximum number of netlink families is 32. If everyone registers its own protocol this number may exhaust.

Generic Netlink

To eliminate these drawbacks the ”generic netlink family” was introduced in Linux kernel 2.6.15. It acts as a netlink multiplexer, in a sense that different applications may use the generic netlink address family. The architecture of the generic netlink family is described in [37] and [38].

The generic netlink architecture consists of five components:

1. The **netlink subsystem** which serves as the underlying transport layer for all of the generic netlink communications.
2. The **generic netlink bus** which is implemented inside the kernel, but which is available to user space through the socket API and inside the kernel via the netlink and generic netlink APIs.
3. The **generic netlink users** who communicate with each other over the generic netlink bus; users can exist both in kernel and user space.

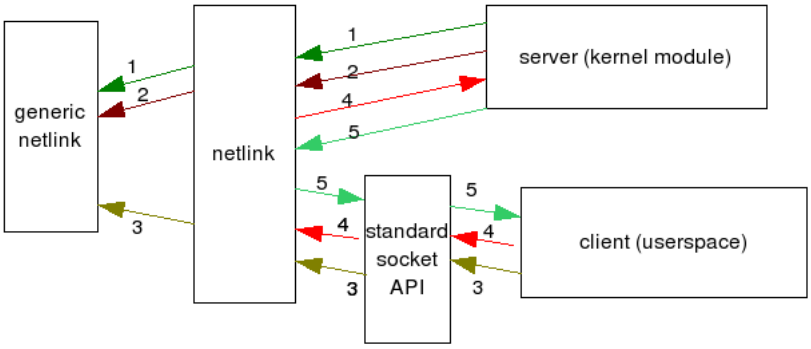


Figure 9: Schema of generic netlink registration.

4. The **generic netlink controller** which is part of the kernel and is responsible for dynamically allocating generic netlink communication channels and other management tasks. The generic netlink controller is implemented as a standard generic netlink user, however, it listens on a special, pre-allocated generic netlink channel.

5. The kernel **socket API**. Generic netlink sockets are created with the `PF_NETLINK` domain and the `NETLINK_GENERIC` protocol value.

Users which provide services over the generic netlink bus establish new communication channels by registering a generic netlink family with the generic netlink controller. Users who want to use a service query the controller with the name of the corresponding generic netlink family to see whether the service exists and to determine the correct channel number. Typically users which provide a service reside in kernel space, whereas users which want to use a service reside in user space.

A generic netlink family consists of a name and a set of functions, which process the data received from the client. Functions belonging to a family are identified with the help of indices. In order to send data to a specific function the client has to know the channel number as well as the index of this function. For security reasons there has to be a policy for each argument which a function expects. This policy defines the type of the argument (e.g. integer, string etc.). If a message is received the Linux kernel verifies the conformance of the argument with the policy for that function. If the argument type is correct, the function is invoked, otherwise the message is simply discarded.

Figure 9 illustrates the registration of a new generic netlink participant.

1. The server has to register the protocol family with the generic netlink controller. The server provides a name for his family and the generic netlink controller assigns a free netlink protocol number to this name. This id is required for any message sent from a client to a server. Figure 10 shows the elements involved to register a generic netlink family.

```
int genl_register_family(struct genl_family *family);
/**
 * struct genl_family - generic netlink family
 * @id: protocol family identifier
 * @hdrsize: length of user specific header in bytes
 * @name: name of family
 * @version: protocol version
 * @maxattr: maximum number of attributes supported
 * @attrbuf: buffer to store parsed attributes
 * @ops_list: list of all assigned operations
 * @family_list: family list
 */
struct genl_family
{
   unsigned int       id;
   unsigned int       hdrsize;
   char               name[GENL_NAMSIZ];
   unsigned int       version;
   unsigned int       maxattr;
   struct nlattr **   attrbuf;        /* private */
   struct list_head   ops_list;       /* private */
   struct list_head   family_list;    /* private */
};
```

Figure 10: Registration of a generic netlink family.

2. The server registers his functions with the generic netlink controller. Figure 11 explains the elements involved in registering a callback function for a given generic netlink family. The doit function should point to the function which will process the data. cmd is the identifier for that function to be used from a client.

3. The client queries the netlink controller for the id of the family. The client has to provide the name of the registered family. If the family is available the id is returned, otherwise a NACK is returned.

4. The client sends a message to the server. The id of the function to be executed, the id of the message type (int, string etc.), the message itself and a unique identifier of the sending process along with some flags have to be provided. Upon receiving this message the netlink interface checks whether the supplied argument, the id of the argument type and the expected type of the argument match and calls the function corresponding to the id given.

5. The server may respond to the client.

For more details and a ”hello world” example refer to [44]. And as the generic netlink HOWTO suggests: ”... As usual, the kernel source code is your best friend...” [38].

Userspace API
All the netlink operations originating from user space can be accomplished with the standard socket interface. However the usage is quite complex, as different headers and different ids and flags have to be set correctly. Therefore there exists the library libnl [41] which takes care of the low level interactions with the netlink interface. Unfortunately this library supports the generic netlink family not yet in its official release. But there is generic netlink support in the current svn version. As the integration of the whole library in the ANA core would be too much overhead, only some tricks are taken from that library, and an ANA specific library was developed.

Binding a Netlink Socket
As for a TCP/IP socket, the netlink bind() API associates a local (source) socket address with the opened socket. The netlink address structure is as follows:

```
struct sockaddr_nl
{
  sa_family_t    nl_family;  /* AF_NETLINK   */
  unsigned short nl_pad;     /* zero         */
  __u32          nl_pid;     /* process pid */
  __u32          nl_groups;  /* mcast groups mask */
} nladdr;
```

The `nl_pid` serves here as the local address of this netlink socket. The client has to choose a unique 32-bit value to fill in `nl_pid`, or set it to zero. If it is set to 0, the Linux kernel associates a pid for that socket, but there is no

```
int genl_register_ops(struct genl_family *, struct genl_ops *ops);

/**
 * struct genl_ops - generic netlink operations
 * @cmd: command identifier
 * @flags: flags
 * @policy: attribute validation policy
 * @doit: standard command callback
 * @dumpit: callback for dumpers
 * @done: completion callback for dumps
 * @ops_list: operations list
 */
struct genl_ops
{
   u8                   cmd;
   unsigned int         flags;
   struct nla_policy    *policy;
   int                  (*doit)(struct sk_buff *skb,
                                struct genl_info *info);
   int                  (*dumpit)(struct sk_buff *skb,
                                  struct netlink_callback *cb);
   int                  (*done)(struct netlink_callback *cb);
   struct list_head     ops_list;
};

/**
 * struct nla_policy - attribute validation policy
 * @type: Type of attribute or NLA_UNSPEC
 *        available types are listed in include/net/netlink.h
 * @len: Type specific length of payload
 */
struct nla_policy {
   u16   type;
   u16   len;
};
```

Figure 11: Registration of a callback function for a generic netlink family.

common way to get to know this pid in the client. If a userspace process has only one open netlink socket, the `nl_pid` can simply be set to the pid of this process. However if a client has multiple open netlink sockets, things become more complex, since the pid cannot be used as a unique identifier. The solution is based on the fact, that a Linux system will never allow 2^{32} different processes running simultaneously, and hence not all bits of the int returned by `getpid()` are used. Therefore the `nl_pid` can be combined from the pid of the process and a process local unique identifier. A function providing a unique `nl_pid` is provided in the libnl library. A server in the Linux kernel which wants to send a message to the client uses this pid as a destination address.

Use of Generic Netlink Sockets in the ANA Core

Kernel Space MINMEX
The MINMEX registers a netlink family with the generic netlink controller. A Brick can identify this family by its name ”ANA”. Thereafter the MINMEX registers a callback function for control messages and one for data messages as well as the policies for the argument of the callback function. In the current implementation the policy is defined to be a string of maximum length 1024. The callback functions call the data respective control parsing functions common to all communication modes. Figure 12 shows the registration of the generic netlink environment to receive AL0 control messages in the MINMEX.

This design provides a lot of flexibility, since new callback functions as well as new policies can be added easily. Also, when in a future step security becomes more important the policy can be easily extended to constrict the arguments more precisely. It would be even possible to change the message format from the current character array, to a struct which holds the different elements.

User Space Brick
An ANA Brick may use any combination of UDP and generic netlink sockets to send and receive data and control messages respectively. There exists at most two generic netlink sockets, one for control communication, and one for data communication, since they can be easily used in duplex mode. If either sending or receiving of data goes with UDP sockets, there exists a “one way” UDP socket and a “one way” generic netlink socket.

The assignment of a unique address is done upon the initialization of the Brick, with the `generate_local_port` function provided by the ANA core library. In order to send data from kernel space to user space, the kernel space has to know about this ID. Therefore the ID of the control as well as of the data socket is part of the attach messages sent from the Brick to the MINMEX.

To facilitate the usage of generic netlink sockets from within a Brick, a small library has been developed. This library is not limited to the ANA world, but it could be used from any user space process interested in the generic netlink sockets. The library consists of the following functions:

- `uint32_t generate_local_port(void)`
  This function generates a unique id which may be used to bind the socket.
- `int create_nl_socket(int protocol, int groups, int pid)`
  This function generates a generic netlink socket and binds it to the pid

```
/*ANA family definition*/
struct genl_family ANAFamGN = {
   .id = GENL_ID_GENERATE,  //genetlink should generate an id
   .hdrsize = 0,
   .name = "ANA",           //the name of this family,
   .version = 1,
   .maxattr = ATTR_MAX,     //defined in anaCommon.h
};
/*control structure definition*/
struct genl_ops controlOpsGN = {
   .cmd = COMMAND_CONTROL,  //defined in anaCommon.h
   .flags = 0,
   .policy = controlPolicyGN,
   .doit = recvControlGN,
   .dumpit = NULL,
};
static struct nla_policy controlPolicyGN[ATTR_MAX + 1] = {
   [ATTR_ID] = { .type = NLA_STRING, .len = 1024 },
                            //ATTR_ID defined in anaCommon.h
};

/*registered callback function for processing control messages */
int recvControlGN(struct sk_buff *skb_2,
                  struct genl_info *info)
{
   /* for each attribute there is an index in
    * info->attrs which points to a nlattr structure
    * in this structure the data is given
    */
   struct nlattr * na = info->attrs[ATTR_ID];
   char * mydata = (char *)nla_data(na);
   int length = nla_len(na);
   /*evoke the ANA parse control function */
   parseControl(mydata, length);
   return 0;
}

/*registration of the generic netlink family*/
genl_register_family(&ANAFamGN);
genl_register_ops(&ANAFamGN, &controlOpsGN);
```

Figure 12: The generic netlink family for receiving control messages in the MINMEX.

specified in the argument. `protocol` has to be set to `NETLINK_GENERIC` and `groups` to a multicast group mask (or 0 if no mulitcast messages should be received).

- `int sendto_fd(int s, const char *buf, int bufLen)`
  This function sends `buf` to the socket descriptor `s`.
- `int get_family_id(int sd, char * family)`
  This function returns the id of the family identified by `family`. `sd` has to be a valid generic netlink socket descriptor.

There exist also functions to send and receive data and control messages. However these functions are integrated in the “standard” functions to send and receive messages provided by the ANA core library.

More information about the generic netlink implementation in the ANA core can be found in the source code [47]. The following files are of special interest:

- `C/minmex/kernel/kernel_minmex.c`
- `C/bricks/AL0/userspace/UanaLib0.c`
- `C/bricks/AL0/common/anaLib0.c`

4.1.3 Kernel Intern Communication

This section describes the scenario where both MINMEX and Brick are in the Linux kernel space. We describe how modules can access functions from other modules, without the need to export all these functions in general. In the second part we describe the ANA core specific implementation.

Minimize the Number of Exported Functions

Since in the Linux kernel all modules run as a single process there is no need for an inter process communication mechanism. Instead, a module can call the functions from another module directly. However, functions are in general private to the module owning the function. In order to make a function visible to other modules the owning module has to export the function explicitly. Since every exported function has to have a unique name the number of exported functions should be kept small. If module A needs nevertheless to make many functions accessible for other modules it can do the following: In a first step it saves all function pointers in a struct. In a second step it exports a function which returns a pointer to this struct. If a module B wants to call a function from module A, it first has to call the exported function. From the struct received, module B can gather the addresses of all the functions registered from module A.

Due to module dependencies this mechanism has the drawback that module A has always to be loaded when module B is loaded, even when module B does no longer need the functions of A.

Another drawback arises when multiple similar modules exists, since then every module has to export a function with a different name. This situation could arise in the ANA world in a test setting, where multiple ANA MINMEXs

run on the same machine. To avoid this drawback a third “management module” is inserted. It provides global functions to register and unregister other modules. Upon registering, a module provides a struct with its function pointers along with a name. Another module interested in these functions can query the control module for the requested name and it gets the corresponding pointer to the struct. After this step communication between the two modules does no longer need the control module.
This architecture provides the following benefits:

- There is no direct dependency between the modules A and B.
- The amount of exported functions does not depend on the number of inserted modules.
- The management module can be used for monitoring and other control purposes too.

There is one drawback with this approach: Module A can be removed from the kernel regardless the needs of module B. Therefore module B could use a function which belongs to module A even when module A is already removed, which would lead to a segmentation fault. This implies the need of a mechanism, that module A can contact module B and inform it about its upcoming removal.

Implementation of the KERN Mode in the ANA Core

In addition to the ANA MINMEX and the Brick modules there exists an anaControl module.

Figure 13 shows the control module along with two registered MINMEX. Two Bricks have queried the control module for the MINMEX and they have obtained the information necessary to communicate with the MINMEX. This procedure is explained in more detail in the next few paragraphs.

The control module provides the following three functions:

- `registerAnaMinmex(struct anaMinmexFunctions * anaMinmexFct, int mode, char * name)`
  registers the struct `anaMinmexFct` with given name and communication mode.
- `unregisterAnaMinmex(char * name)`
  unregisters the node functions belonging to the given name.
- `getMinmexFunctions(char * name)`
  returns a pointer to the node functions identified by the given name.

This simple interface is not yet safe, since the unregister function can be called from any module. However it would be not difficult to add a third private value to the `registerAnaMinmex` function, which has to be provided for unregistering the MINMEX. The control module manages its entries with a linked list as provided by the Linux kernel (`linux/list.h`).
Upon initialization, the ANA node registers a name, its mode and the following struct with the control module:

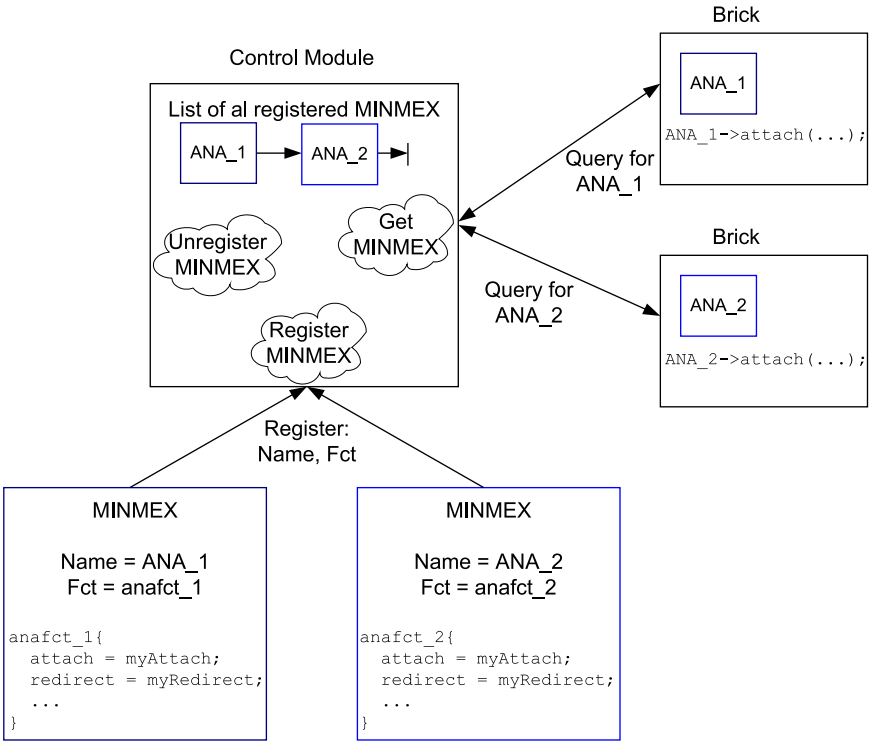


Figure 13: A PC hosting two different MINMEX. The Bricks query the control module for the functions provided by the MINMEX.

```
struct anaMinmexFunctions{
    int  (*attach)(char *msg, int len);                    (1)
    void (*detach)(char *msg, int len);                    (2)
    int  (*regist)(void *msg, int len);                    (3)
    int  (*redirect)(char *msg, int len);                  (4)
    void (*unregist)(char *msg, int len);                  (5)
    void (*heartBeat)(char *msg, int len);                 (6)
    void (*notification)(char *msg, int len);              (7)
    int  (*changeView)(char *msg, int len);                (8)
    void (*callback)(char *msg, int len);                  (9)
    void (*brickDataChannel)(struct anaBrickFunctions     (10)
                             *brickDataFunctions,
                             anaHandle_t handle)
};
```

Functions (1) to (9) are the normal functions executed from the MINMEX as described in section 3.2.4. Function (10) is a helper function. It is called by the Brick to register the functions it provides for the MINMEX. In the current implementation this is the function which parses data messages received from the MINMEX (11) along with a urn (12). Theoretically a Brick may attach to more than one MINMEX. Therefore the Brick has to know from which MINMEX it has received the data. The urn is exactly for this purpose. It corresponds to the data gate urn specified upon insertion of the Brick module. Upon sending a data message to the Brick, the MINMEX invokes the callback function. The argument `name` is set to the urn which identifies itself.

```
struct anaBrickFunctions
{
   /*data*/
   void (*callback)(char * msg, int len, char * name);       (11)
   /*urn over which this Brick is reachable for this node*/
   char * urn;                                               (12)
};
```

A Brick which wants to be attached to a given MINMEX queries the control module with the name of the MINMEX. If a MINMEX with this name is registered the corresponding `anaMinmexFunctions` pointer is returned, and the Brick can access the MINMEXs functions directly. During the attachment process it calls the MINMEXs `attach()` function as well as the `brickDataChannel()` function to register its data processing function. The attachment procedure between a Brick and a MINMEX is shown in Fig. 14. Figure 15 shows the basic code used to send a data message from the MINMEX to a Brick.

```
/* Brick */
/* urn over which this Brick is reachable for the MINMEX
 * in the "real" implementation this is given as cmd line argument
 */
char * urn = "NodeANA"
/* initialize Bricks callback function */
struct anaBrickFunctions *myFunctions
              = mallocWrapper(sizeof(struct anaBrickFunctions));
/*the parseCallback function parses the received data messages */
myFunctions->callback = parseCallback;
myFunctions->urn = mallocWrapper(strlen(urn)+1);
memcpy(myFunctions->urn, nodeName, strlen(urn)+1);

/* get the functions from the MINMEX with the name "ANA". */
struct anaMinmexFunctions * MINMEX1 = getNodeFunctions("ANA");

/* attach Brick to MINMEX with the prepared attach message */
MINMEX1->attach(attachmessage, len);


/* MINMEX */
int handleAttach(char *msg, int length) {
        /* parse the attach message and create a new handle
         * for this Brick, insert Brick in BrickTable,
         * copy reply message to msg
         * /
         memcpy(msg, replyMsg, replyLen)
 return replyLen;
}


/* Brick */
/* parse the reply
 * get the handle over which this MINMEX identifies this Brick
 */
/* register the Bricks data parsing functions with the MINMEX */
MINMEX1->brickDataChannel(myFunctions, handle);


/* MINMEX */
/* update the entry in the client table for this Brick */
void handleClientDataChannel(struct anaBrickFunctions
                             * brickDataFunctions,
                             anaHandle_t handle){
        struct CTentry_s* entry = getCTEntry(&CT, handle);
        entry->dataAux = brickDataFunctions;
}
```

Figure 14: Attachment from a Brick to the MINMEX.

```
/* MINMEX */
/* Data is sent in the following way: */
struct CTentry_s* entry = getCTEntry(&CT, handle);
struct anaBrickFunctions * clifunc  = NULL;
clifunc = (struct anaBrickFunctions *)entry->dataAux;
clifunc->callback(msg, len, clifunc->urn);


/* Brick */
void parseCallback(char * msg, int len, char * urn){
        /*search for the correct MINMEX */
        struct anaNodeSpecs_s * tmp;
        tmp = minmexList;
        while (tmp != NULL){
                struct anaBrickFunctions * myfnc;
                myfnc = tmp->myDataFunctions;
                if (!memcmp(myfunc->urn, urn, strlen(myfnc->urn)))
                        break;  /* correct entry found */
                }
                tmp=tmp->next;
        }

        /* check whether tmp == NULL -> no MINMEX found
         * otherwise correct MINMEX found ->
         * lookup the IDP in the shadow dispatch table,
         * the first element in msg points to the idp
         * /
         anaLabel_t idp = msg;
         struct SDTentry_s *entry = NULL;
         entry = getSDTEntry(tmp->shadowT , idp);

         /* send the data to the function corresponding to the IDP
         data = msg + tmp->labelSize;
         entry->callBackFct(data, len - tmp->labelSize, idp,
                            tmp, entry->aux);
}
```

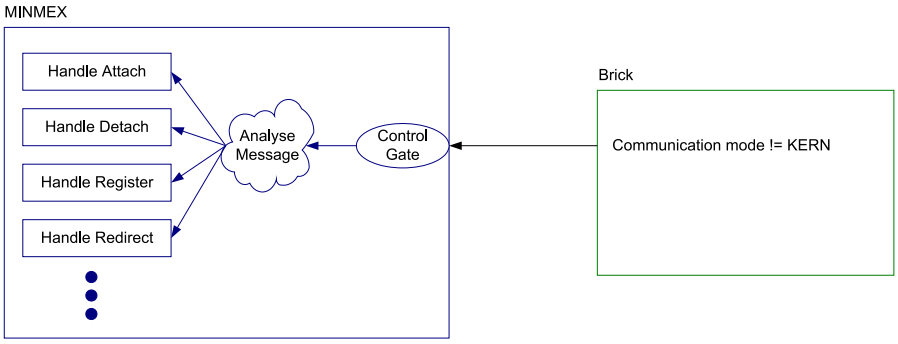
Figure 15: Sending of a data message from the MINMEX to a Brick identified by “handle“.

If a MINMEX is removed from the kernel (e.g. rmmod anaMinmex) it notifies the Bricks about this event by calling the `kickAllBricks()` function. Thereafter all Bricks free the resources belonging to this MINMEX. Finally the MINMEX unregisters itself from the anaControl module.

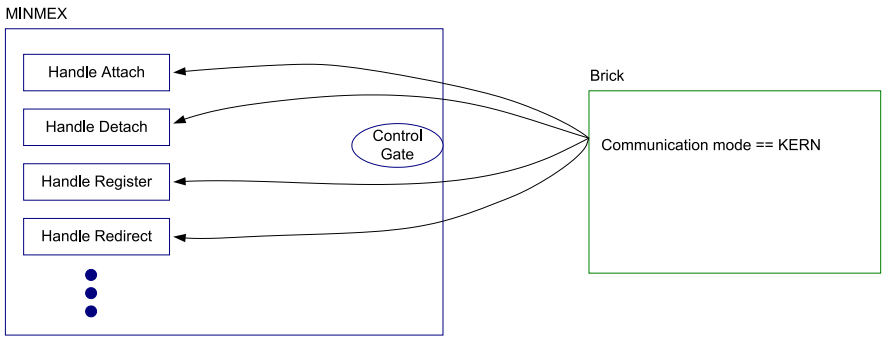
There is a conceptual difference between the communication modes UDP, UNIX, PIPES and GENETLINK on the one side and KERN on the other side. The first imply that the Brick has to send a control message to the control gate of the MINMEX. The MINMEX then analyzes the request message and invokes the corresponding function. When the function is executed the MINMEX sends a response message back to the control gate of the Brick. If both MINMEX and Brick are in kernel space, the Brick invokes the control function of the MINMEX directly. In order to achieve compatibility with the message passing system of the first group, the exchanged arguments are the same. This means that the kernel space version encodes its request in a message and invokes the control function with this message as an argument. The control function parses the message and executes the function. When the function is executed it writes its reply in the buffer which was before used for the command argument. Figure 16(a) shows the message flow for the UNIX, PIPES, UDP and GENETLINK mode, the kernel intern mode is depicted in Fig. 16(b).

More Information about the KERN communication mode can be found in the following code files:

- `C/bricks/AL0/kernel/KanaLib0.c`
- `C/bricks/AL0/common/anaLib0.c`
- `C/minmex/kernel/kernel_minmex.c`
- `C/minmex/common/minmexFunctions.c`



(a) Control Path for UNIX, PIPES, UDP and GENETLINK.



(b) Control Path when the MINMEX and the Brick are in kernel space.

Figure 16: Brick starting a control request. If the MINMEX and the Brick are both in kernel space, the Brick calls the control function directly. Otherwise the Brick sends a control request message to the MINMEX, whereupon the MINMEX invokes the correct control function.

4.1.4 Evaluation: Throughput of the ANA Core

The aim of this section is to compare the performance of the different communication mechanisms implemented. We are not interested in exact numbers or in long term studies since the goal of the ANA prototype is to provide a network architecture that has different properties than the Internet and not one that is optimized for speed.

Of special interest is the maximum possible packet rate a Brick can send to another Brick on the same node without loosing packets. The tests show that the kernel intern communication mode is by far the fastest and the only one which never looses a packet.

The test scenario is as follows: There is a packet sending and a packet receiving Brick. The sender Brick sends 12 Million packets continuously to the MINMEX which forwards them to the receiver Brick. The receiver Brick counts each packet received, but does no further packet processing. Figure 17 shows the test setup.

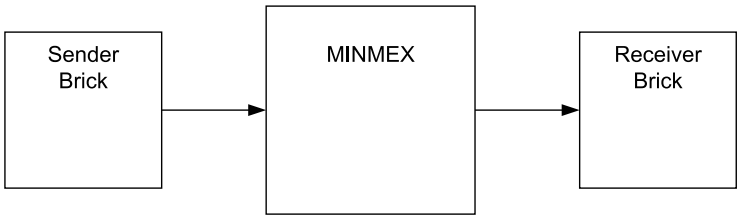


Figure 17: Test setup: The “sender Brick” sends continuously messages to the “receiver Brick”.

The maximum possible packet rate (without packet loss) is depending on the payload of each packet transmitted. Therefore we did the tests with the following two payload sizes:

1. Payload of 6 Bytes.
2. Payload of 1 kByte.

Throttling of sender packet rate

In order to regulate the packet rate, the sending Brick has to wait in regular intervals for a short period of time before sending more packets. The waiting is accomplished by putting the sending process to sleep. The minimum possible sleep time corresponds to the Linux kernel timer interrupt resolution, which is at most $1ms$. This is not optimal, since packets are always sent in bursts during which the packet rate is high followed by a period in which no packets are sent. Presumably the maximum possible packet rate would be higher if the packet rate would be constant. Nevertheless the reported packet rates give a clear indication of which communication modes are fast and which are slow.

Results

Table 2 summarizes the packet throughput for a payload of 6 Bytes and Tab. 3 summarizes the results for a payload of 1 kByte. Three main points can be observed:

1. IPC mechanisms specialized for communication within a computer are more efficient than UDP which is designed as a communication protocol between distant computers.
2. The ANA core can process more packets the more parts of it run in the Linux kernel space. By far the fastest communication mechanism is “KERN” in which the Bricks call the functions provided by the MINMEX directly.
3. The packet rate decreases by approximately 30% if the payload is changed from 6 to 1024 Bytes. This increases the bandwith used by a factor of approximately 100.

MINMEX	Brick	Packet Rate	Bandwith
u-space: UDP	u-space: UDP	29’000	1.4
u-space: UNIX	u-space: UNIX	41’000	2
k-space: UDP	u-space: UDP	41’000	1.9
k-space: GENETLINK	u-space: GENETLINK	98’000	4.7
k-space: UDP	k-space: UDP	65’000	3.1
k-space: KERN	k-space: KERN	366’000	17.7

Table 2: Maximum packet rate (packets/second) and Bandwith (Mbit/s) for packets with a payload of 6 Bytes without packet loss.

MINMEX	Brick	Packet Rate	Bandwith
u-space: UDP	u-space: UDP	21’000	171
u-space: UNIX	u-space: UNIX	37’000	299
k-space: UDP	u-space: UDP	20’000	163
k-space: GENETLINK	u-space: GENETLINK	69’000	565
k-space: UDP	k-space: UDP	47’000	385
k-space: KERN	k-space: KERN	266’000	2180

Table 3: Maximum packet rate (packets/second) and Bandwith (Mbit/s) for packets with a payload of 1 kByte without packet loss.

Using Fig. 18 and Fig. 19 we can compare the packet loss and the CPU load for different packet rates. These values are obtained from a test with minimum sized packets. As clearly can be seen the KERN communication mode poses by far the least computational burden to the CPU. Whereas the GENETLINK mode is computational expensive, but does not loose any packets until the packet rate is approximately 100’000 Packets per second. GENETLINK as well as UNIX are not able to send packets faster than a threshold, but up to this value the packet loss rate is approximately 0. By contrast the UDP mode. UDP mode is able to send packets very fast, but the faster they are sent the more packets get lost.

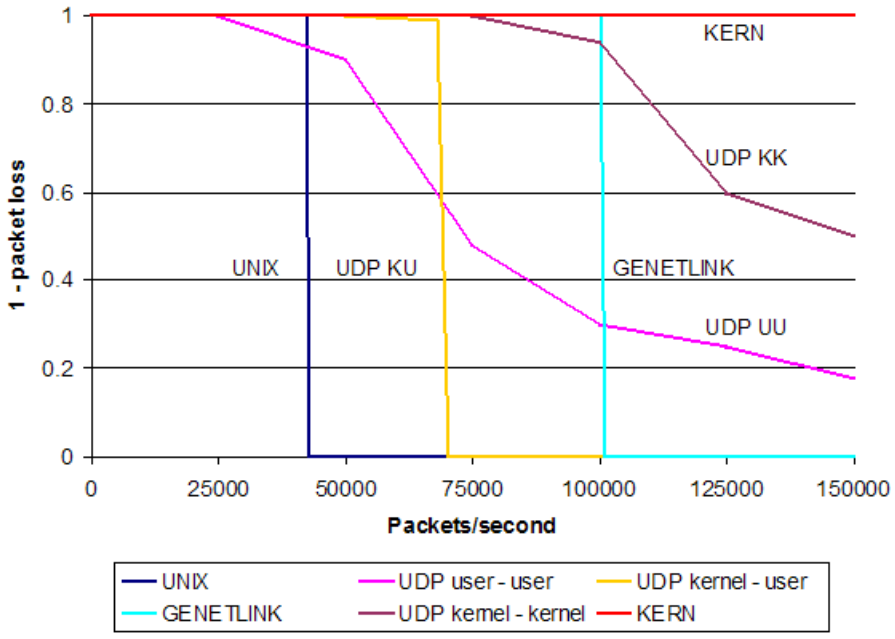


Figure 18: Percentage of delivered packets depending on the packet rate.

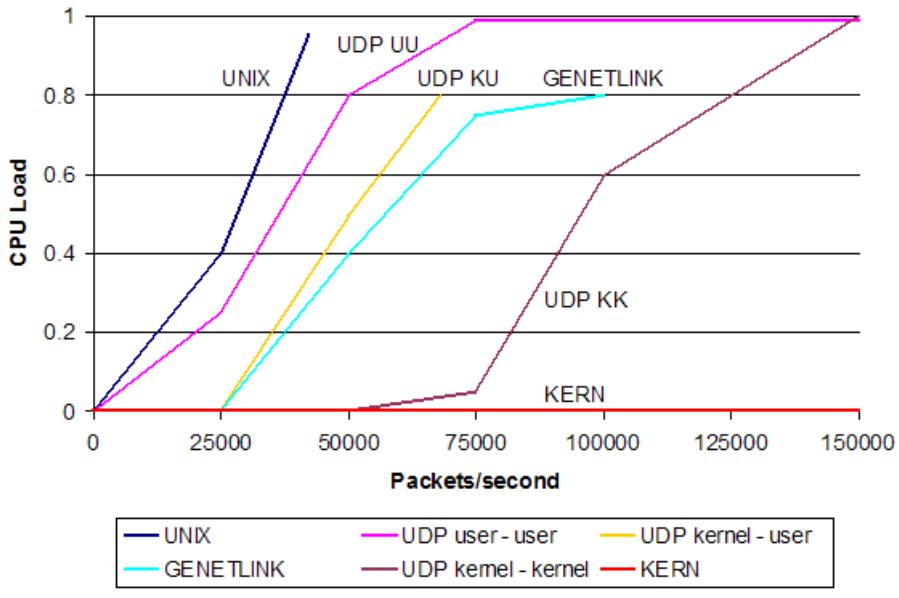


Figure 19: CPU load for different packet rates.

4.1.5 Summary

The described communication mechanism UDP, GENETLINK and KERN along with the userspace communication modes UNIX and PIPES provide an adequate communication channel between the MINMEX and the Bricks for any given situation. However all these communication mechanisms are well hidden in the ANA core software and the different properties have been merged in order to fit in the given AL0 API.

The evaluation has shown, that the communication modes have different characteristics as far as maximum possible packet rate and packet loss is concerned. The KERN communication mode is by far the fastest and the only one which provides a reliable connection between the MINMEX and the Bricks. Packet loss in an ANA node could lead to severe problems in the future. To see this we examine the following scenario: In the traditional network stack, if http sends a packet it can assume that the packet is delivered since it uses TCP as a transport protocol. In the ANA world, both the http and the TCP protocol would be implemented as dedicated Bricks. Since UNIX, GENETLINK and especially UDP may loose some packets in one single node, the http connection would still be unreliable.

Therefore either all Bricks which need a reliable connection have to run in the kernel space, or we have to provide another communication mode which offers a reliable connection between the MINMEX and the Bricks.

However for a network node which is loaded only moderately no packet loss should occur for any of the implemented communication modes.

4.2 Writing System Agnostic Code

A goal of the ANA core software is to provide an environment which runs in Linux user space, kernel space and on the network simulator ns2. Upon writing a Brick a developer should not have to care about differences in the system specific functions. Therefore we have implemented a set of wrapper functions, which can be used by the Brick developer and which hide all the details. This sections describes the concepts behind these wrapper functions. Besides the need of wrapper functions some other things have to be considered when writing system agnostic code. All of them are listed in Appendix B.1.

4.2.1 Standard Wrapper Functions

We have implemented a library of our own functions for the basic interaction with the system. The functions provided from this library are for example malloc and free. An up to date list can be found in the source code in the file `C/shared/anaCommon.c`.

The names of these wrapper functions correspond to their userspace names. It is important to note that the gnu functions and the ANA functions do not correspond in a 1 to 1 relationship. For example the ANA malloc() function initializes explicitly the allocated memory area to 0, and the ANA free() function sets the freed pointer to NULL.

4.2.2 Mutual Exclusion - ANA Locks

The Linux kernel 2.6 is preemptive. This means that the scheduler can revoke the CPU from a thread and pass it to another thread at any time. Since some data structures are used by different threads they need to be locked properly.

To avoid race conditions we have developed ANA locks, which are used to protect all critical data structures of the ANA core software. The Linux kernel provides two different kinds of locking mechanisms:

1. Spinlock: Spinlocks may be used when the process is in “atomic context”, e.g. upon processing an interrupt. When holding a spinlock it is forbidden to sleep, which limits the set of possible functions to be executed while holding a spinlock.

2. Semaphore: When holding a semaphore a process is allowed to sleep. However it is illegal to acquire a semaphore when the process is in interrupt context.

Both spinlocks as well as semaphores are available either as exclusive locks, or as reader/writer locks. When a process holds an exclusive lock, it is the only one who can enter the critical section. For reader/writer semaphores the programmer may distinguish whether he wants to read the data only, or whether he wants to change something. Multiple reads are allowed simultaneously, but writes are guaranteed to have exclusive access to the critical section.

Since we want to provide the ANA locks for all future ANA developers, they have to be easy to use and they should not put any constraints on the functions to be executed while holding the lock. Therefore in the Linux kernel space

the ANA locks are mapped to semaphores with exclusive access to the critical area. In user space they map to a pthread_mutex.

The use of ANA locks is explained in the ANA Core documentation [48].

4.2.3 Parallel Execution - ANA Threads

A Brick receives all data messages on one communication gate. This gate invokes the callback function corresponding to the IDP in the data message. While the callback function does not return, the gate is not able to process any further messages. This could lead to a situation where a callback function of a Brick executes an infinite loop and therefore the whole Brick would be blocked. To avoid this situation, we have implemented an ANA thread library. This library can be used in Linux user space as well as in the kernel space. Thread handling in the kernel space differs from the one in user space. The most important difference between userspace threads and kernel space threads appears upon termination of a program. In user space, upon pressing CTRL-C, all threads get killed as well. Whereas in kernel space upon removing a module the threads continue to live. This situation should be avoided, since it will most likely end in a kernel oops, when the thread tries to refer some memory from the removed module.
In the Linux kernel threads cannot be stopped, but they need to stop themselves. The Linux kernel provides some functions for handling thread termination. A termination signal can be sent to the thread to be killed. The execution of the process sending this signal gets blocked, until the thread to be terminated has really stopped. A kernel thread has to poll for the termination signal and upon receiving it, the thread has to terminate itself.

The main goal of the ANA thread library is to facilitate the termination of threads in the kernel space and to avoid kernel oops caused by orphaned threads. The ANA thread library is designed as follows: There exists one ANA thread environment for each Brick. This environment holds a list of all currently active threads. All threads created with the ANA thread library start with the same thread function. This function has two tasks:

1. Call the thread function specified by the programmer.
2. Unregister itself with the thread environment when the thread function has returned.

Upon starting a thread, the thread identifier (e.g. the pid in user space and the pid as well as the `struct task_struct` in kernel space) is inserted in the threads list.

When the thread environment is quitted in user space, it sends to each thread in the thread list a `SIGTERM` signal.
In kernel space it calls the `kthread_stop` function, which informs the thread that it should stop, and waits until the thread has really stopped. This implies that the thread has to check in regular intervals whether it should stop. For this purpose there exists the `anaThreadShouldStop` function. If a thread does not check for the termination signal, the module which started the thread cannot be removed.

The use of ANA threads is explained in the ANA Core documentation [48].

4.2.4 Summary

Thanks to the ANA wrapper functions, a Brick may run in user space or in kernel space, without the need of modifying their source code.
With the help of the basic wrapper functions it is easy for a developer who is comfortable with user space programming to write a Brick which runs in kernel space as well. The more elaborated functions like ANA locks and ANA threads require some effort of the developer in order to get used to the APIs, but they are much simpler than the standard Linux kernel APIs.

4.3 Backporting to Linux Kernel 2.4

A lot of embedded devices today still run on kernel 2.4[3]. To allow ANA to run on such embedded systems, the ANA core software was backported to late kernels 2.4. The ANA framework is written in a manner that it automatically detects whether it runs on a kernel 2.4 or 2.6 and it chooses the appropriate code to be compiled and executed.

The overall Linux kernel architecture changed from kernel 2.4 to kernel 2.6 significantly. The following list summarizes the main changes as they are relevant for the ANA core software.

- The scheduling mechanism has changed from a non-preemptive kernel 2.4 to a preemptive kernel 2.6. This means that with kernel 2.4 the programmer is responsible to release the CPU to give other tasks the opportunity to be scheduled, whereas in kernel 2.6 the scheduler can revoke the CPU at any time. Both behaviors have pros and cons: in kernel 2.4 one module can consume a lot of CPU time, whereas in kernel 2.6 an unfair module will be suspended. In kernel 2.6 it is important that all data structures which are accessed from multiple functions are appropriately locked, since it is never known when an other function is invoked. This function could manipulate the data structure, which could lead to race conditions.

- In kernel 2.4 all global variables are visible in the whole kernel by default. In the kernel 2.6 series global variables are only visible in one module. In order to make a variable or function accessible by an other module they need to be exported explicitly.

- In kernel 2.4 the programmer is responsible to take care that a module which is used by other modules cannot be removed. This task is performed automatically in kernel 2.6.

- The interface for deferring work in an interrupt handler has changed completely. In kernel 2.4 an interrupt handler uses the task queue when the processing of some data may sleep. Kernel 2.6 interrupt handlers use a work queue. Work queues allow much more flexibility than the task queue: The programmer can easily create multiple work queues and schedule tasks to one of them, whereas in kernel 2.4 one usually uses the default task queue. This implies that tasks in the task queue may block each other, whereas in 2.6 only the work from one and the same work queue may block each other. Furthermore work queues allow the execution of a task to a predetermined time in the future, whereas the task queue is always executed “as soon as possible”. In order to defer work until a predetermined time in the future, in kernel 2.4 a combination of kernel timers and the task queue is needed.

- The functions for handling kernel threads have changed completely. In kernel 2.6 there are dedicated functions to be used in order to stop a kernel thread. In kernel 2.4 this task is accomplished with the help of signals and the use of a so called completion object.

[3] E.g. the released version of the OpenWRT [46] Linux distribution for embedded systems runs on Linux kernel 2.4.

- The interface on how commandline parameters for a kernel module are specified has changed.
- The whole Makefile structure has changed completely. Kernel 2.4 is more oriented on a ”normal user space Makefile” whereas kernel 2.6 has a dedicated Makefile format which is more comfortable to use.

5 The Bootstrapping Phase How Nodes Get to Know Each Other

Whereas the first part of this Masters thesis is concerned about the communication between Bricks and the MINMEX in the same node, the second part deals with the communication setup between different nodes. The focus lies on the bootstrapping phase, in which nodes discover each other and exchange information in order to learn each others capabilities.

Upon starting an ANA node, the Bricks have to organise themselves in order to be able to process packets from other nodes. The building of this dynamic protocol stack is described in the first section. In a second phase the Bricks have to learn about similar Bricks residing on other nodes. Thereto exist some Bricks which provide a bootstrapping service. This bootstrapping mechanism is specified in the second section.

5.1 Building a Dynamic Protocol Stack

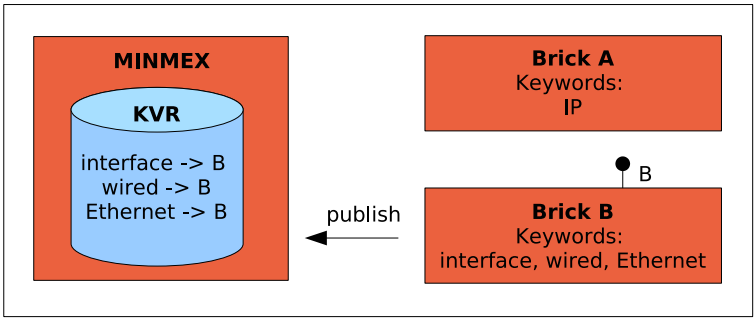
During the start up of an ANA node, each Brick registers itself with the MINMEX. But the individual Bricks do not know anything about other Bricks residing on the same node. Since the ANA project wants to achieve autonomicity, the Bricks cannot relay on a fixed protocol stack, but they have to build a protocol stack dynamically. This allows the exchange of some Bricks during operation, depending on the networks current needs.

Figure 20 shows how a dynamic protocol stack is set up. This dynamic protocol stack is built with the help of the Key-Val Repository (KVR) introduced in section 3.1.3. Each Brick interested in participating in the protocol stack has to accomplish the following steps:

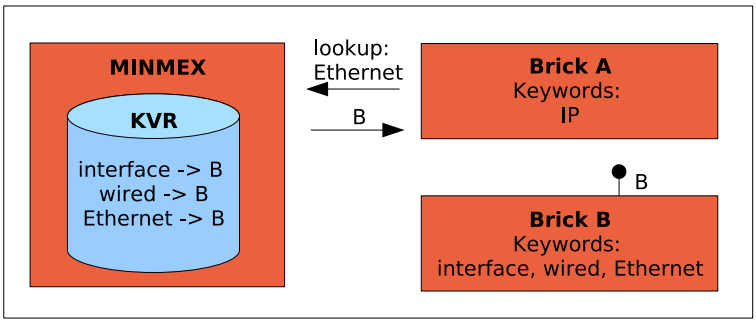
1. Publish some keywords which describe its capabilities in the KVR.
   Fig. 20(a): Brick B publishes its keywords "interface, wired, Ethernet" in the KVR.

2. A Brick interested in a certain service sends a lookup query to the MINMEX. The MINMEX queries the KVR and returns the IDPs matching the query.
   Fig. 20(b): Brick A queries the KVR for the keyword "Ethernet" and obtains the IDP B.
   Upon knowing this IDP the Bricks can start to communicate. They can negotiate some protocol information (e.g. MTU etc.), and decide how data messages are exchanged.

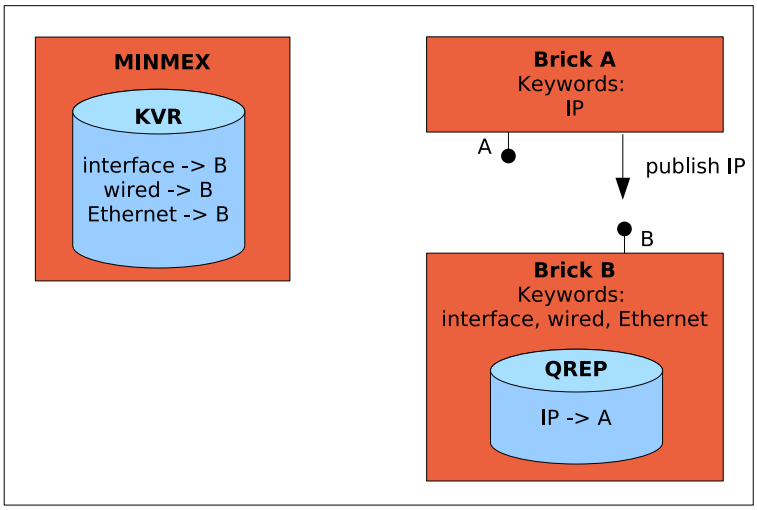
3. Bricks may offer bootstrapping services, which can be used from "common" Bricks to get to know Bricks residing on other nodes. A common Brick can choose whether it wants to publish some keywords to a Brick with boostrapping functionality. This keywords are stored in a repository of the Brick with bootstrapping functionality. The entries of this repository are publicly available and are used during the bootstrapping phase.
   Fig. 20(c): Brick A publishes the keyword "IP" in the bootstrapping repository of Brick B.



(a) Publish keywords in KVR.



(b) Query KVR for a keyword.



(c) Publish keywords in the bootstrapping Brick.

Figure 20: Three steps for building a dynamic protocol stack.

4. Since communication is in most cases bidirectional it is useful to provide the address of the originator of a bootstrapping request to the Bricks targeted by the query. For this purpose a Brick with bootstrapping functionality may provide a second repository in which other Bricks can publish their description. When a bootstrapping query matches an entry in this repository, the corresponding Brick obtains the information necessary to contact the querying Brick. We refer to this process as subscription.

5.2 Get to Know Other Nodes

This section describes the bootstrapping phase where Bricks get to know Bricks residing on other nodes. Throughout this section it is assumed, that the protocol stack is already built, and that Bricks interested in the bootstrapping phase have already published their description in the corresponding repositories.

In order to be able to communicate with other Bricks, the IDPs of the peer Bricks have to be learned. For this study it is assumed that all nodes participating are attached to the same Ethernet segment. Access to the Ethernet is open to each node, and therefore all Ethernet Bricks can directly communicate with each other. However, a Brick which has no direct media access, e.g. which is subscribed to the Ethernet Brick, has first to discover other Bricks of the same type.

Apart from the standard message exchange, each Brick can choose between three different bootstrapping options:

1. **Aggregation:** All bootstrapping replies are delivered in one package.
2. **Cache:** The bootstrapping Brick manages a cache of recently learned addresses. Instead of broadcasting the bootstrapping request it queries its cache repository.
3. **Subscription:** A Brick can subscribe to bootstrapping events. These Bricks get the address of the “querying Brick” whose boostrapping request matched the Bricks keywords.

The most comfortable way to describe this bootstrapping procedure is to look at an example. Figure 21 shows the basic example setup, consisting of two nodes connected by an Ethernet segment. Each node has 4 functional blocks. One of them is an Ethernet functional block. It consists of two Bricks, one responsible for the framing of the Ethernet packets and sending them to the wire and one which offers bootstrapping functionality to the remaining three functional blocks.

In a first step we are looking at node A which initiates the bootstrapping request and in a second phase we cover node B which answers the bootstrapping request.

Node A

Brick A describes itself with the keywords “A” and “O”.[4] It is interested in other Bricks which describe themselves either with “A” or with “O”. During the bootstrapping procedure the following steps are executed in node A (q.v. Fig. 22):

[4] This keywords can be replaced by anything according to an ontology which is still to be defined.

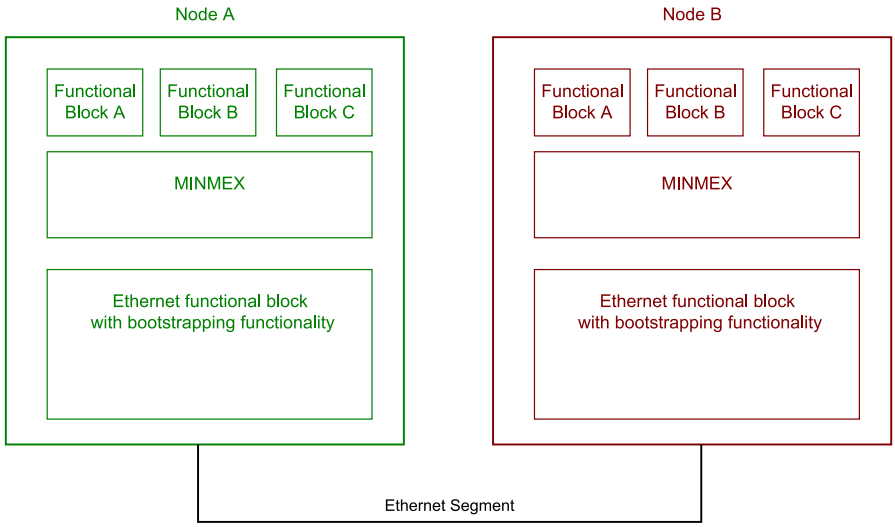
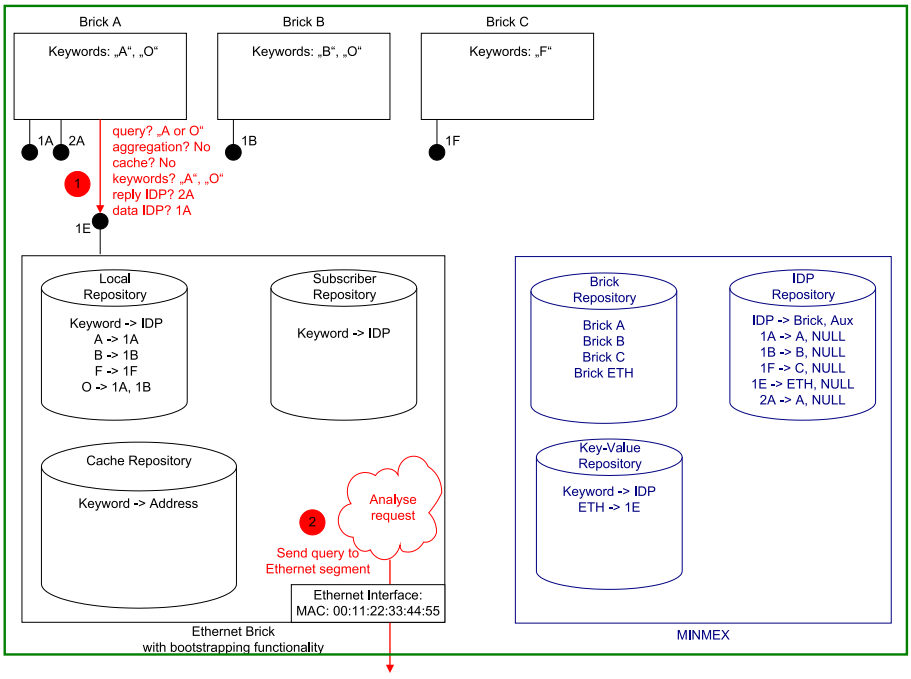
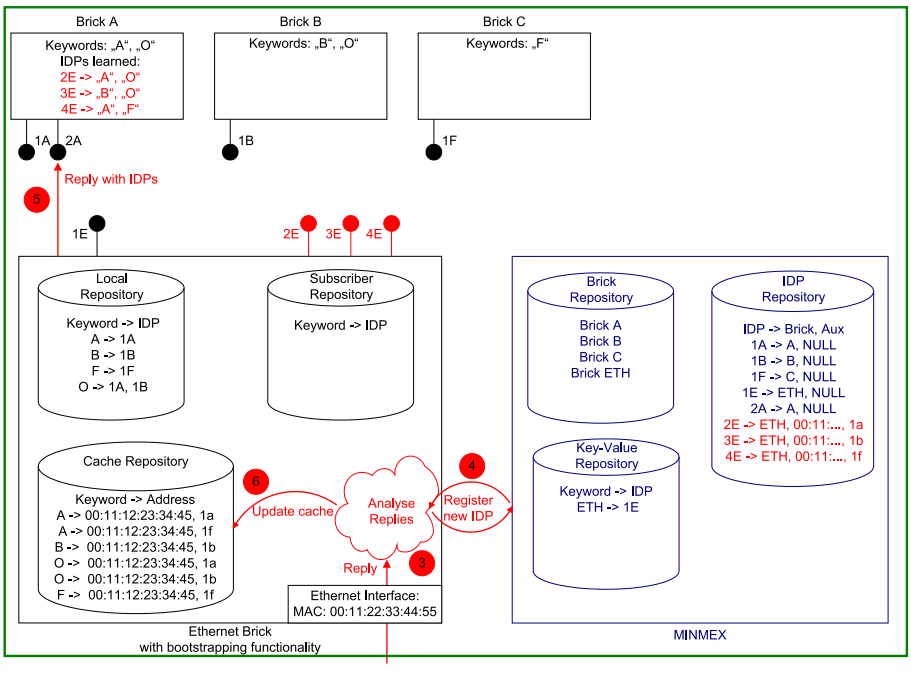


Figure 21: Example setup to explain the bootstrapping process.

1. Brick A sends a query describing the request to the Brick with the bootstrapping functionality. This request consists of the following 6 arguments:
   (a) Query: The query consists of keywords concatenated by AND, OR or NOT. It describes the Bricks of interest.
   (b) Aggregation: Describes whether the answers should be given one by one or aggregated in a package.
   (c) Cache: The cache parameter specifies whether the query should be looked up in the local cache or whether the query should be broadcasted.
   (d) Keyworks: Brick A describes itself with this keywords, they are used to update the cache of remote Bricks.
   (e) Data IDP: Brick A waits on this IDP for data messages, used to update the cache of remote Bricks.
   (f) Reply IDP: The boostrapping answers have to be sent to this IDP.
2. The Brick with the bootstrapping functionality analyzes the request. Depending on the cache parameter, the query is looked up in the local cache or the request gets broadcasted.
3. Upon receiving an answer the following information is obtained:
   (a) The MAC address of the remote network interface.
   (b) The IDP where the peer Brick listens for data messages.
   (c) A set of keywords describing the peer Brick.



(a) Brick A initiates a bootstrapping request.



(b) The Ethernet Brick handles a bootstrapping reply.

Figure 22: Bootstrapping process in node A.

4. The bootstrapping Brick registers a new IDP with the MINMEX for each discovered Brick. This IDP is associated with the function the Brick provides for sending data messages along with the MAC address and the IDP of the peer Brick.

5. Depending on the aggregation parameter of the original request, the responses are sent back immediately, or after a network specific delay. A response consists of the newly generated IDP and the keywords identifying the peer Bricks.

6. The bootstrapping Brick updates the local cache with the keywords, MAC address and IDP learned from the boostrapping reply.

Node B

In our example node B receives the bootstrapping request. In order to minimize the amount of messages to be exchanged, Brick A and Brick B wants to learn about Bricks which have sent a query matching their keywords. Therefore they have published their keywords in the subscriber repository of the Brick offering bootstrapping functionality. The bootstrapping procedure in node B is depicted in Fig. 23. During the bootstrapping phase the following steps are executed:

1. Upon receiving a bootstrapping request, the Brick with the bootstrapping facility checks the local repository for entries corresponding to the query. All IDPs along with their keywords are sent back to the originator of the query.

2. The cache repository gets updated with the information learned from the bootstrapping request packet.

3. The Bricks subscribing to the bootstrapping events get notified.

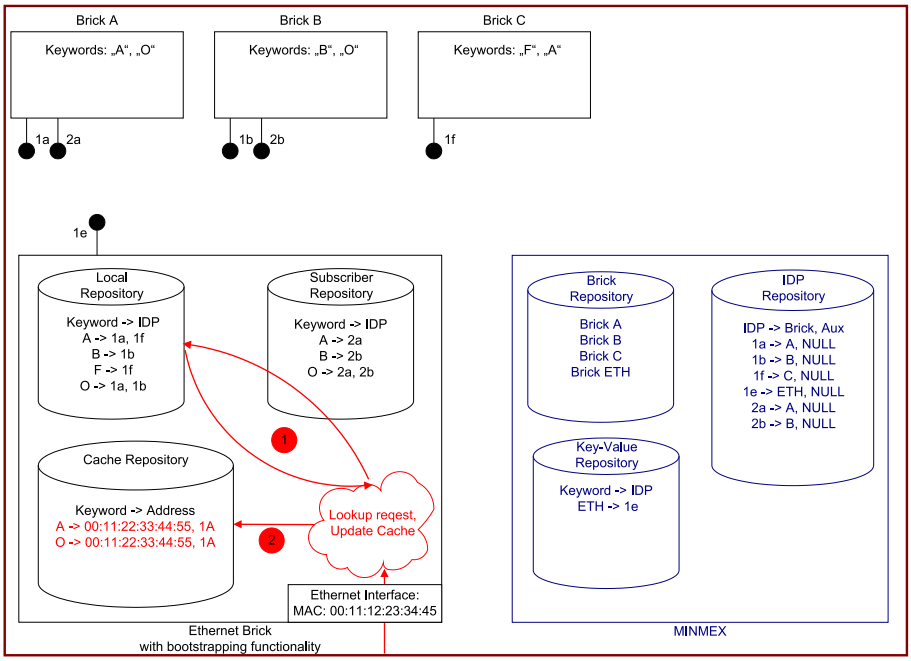
   (a) The subscriber repository is queried.

   (b) For each Brick found a new IDP is registered.

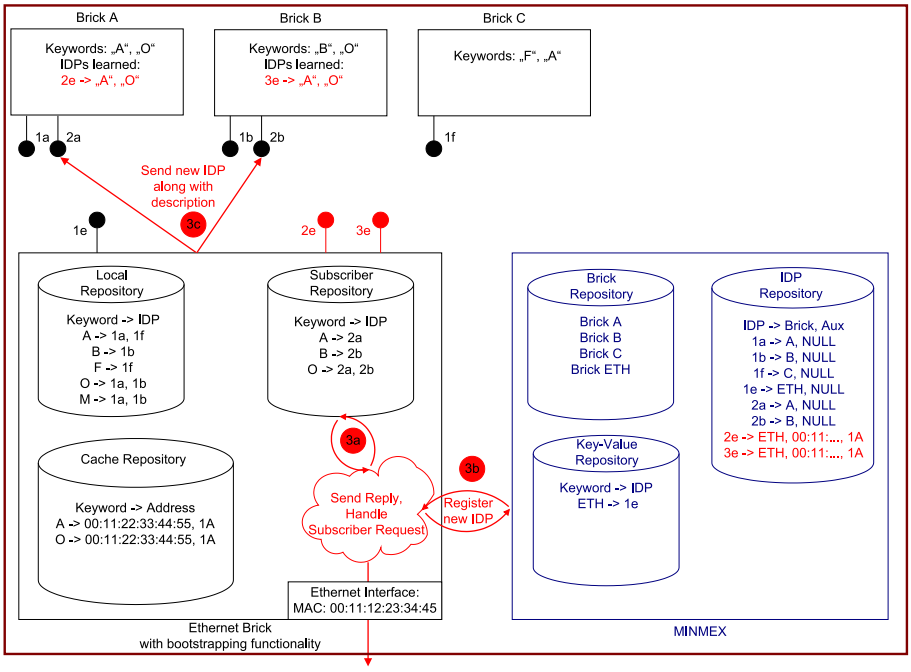
   (c) The subscriber Bricks are notified of the newly generated IDP along with the description of the source Brick. The subscriber Brick can use this IDP to send a message to Brick A on node A.

5.3 Chat Application

We have written an Ethernet compartment which provides the described bootstrapping functionality. Instead of accessing the hardware directly it uses another Brick (called vlink Brick) which provides virtual access to the hardware. This allows the creation of virtual Ethernet links between distant networks. On top of this “vlink-Ethernet” compartment we have implemented a simple chat application. All participants on the same Ethernet segment communicate together, there is no private chat up to now.
If a chat Brick starts, it publishes its description “chat” in the Ethernet Brick and it subscribes itself to bootstrapping events matching the keyword “chat”. Thereafter it sends a bootstrapping request to the Ethernet Brick, which forms



(a) Answering the bootstrapping request.



(b) Notify subscriber of bootstrapping request.

Figure 23: Bootstrapping process in node B.

an Ethernet frame for the request and forwards this to the vlink Brick. The vlink Brick processes this frame according to its configuration (e.g. it could send a broadcast message through eth0). Since all participants have subscribed to bootstrapping events, they immediately know the address of the newly started Brick and can start to send chat messages to this Brick.

5.4 Summary

The described bootstrapping procedure implements a “default off” policy. This means that each Brick has to explicitly choose over which other Bricks it wants to be reachable. Default off polices have the drawback, that each communication has to be allowed explicitly, whereas in a default on policy all possible communication channels are allowed. However, default off policies are preferred over default on policies, since they offer much less flexibility to attackers.

Bricks participating in the bootstrapping phase can choose which information is publicly available, by adding a different set of keywords to the local repository, to the subscriber repository and to the bootstrapping request message.

The bootstrapping procedure offers great flexibility. Each of the three different parts: Aggregation, Cache and Subscription may be implemented independently and for a lightwight bootstrapping process they can be omitted completely.

An implementation of this bootstrapping scenario can be found in the vlink-Ethernet compartment. It is the first compartment written for ANA. It offers Ethernet connectivity running over virtual links, thus enabling to connect ANA nodes on different physical networks. We have written a simple chat application which uses the vlink-Ethernet compartment. All the participants are able to communicate together without the need of configuring anything and therewith we have proven that the ANA core software and the described bootstrapping procedure work as expected.

Although we usually speak in terms of “Brick A sends a message to Brick B” it is important to remember, that actually Brick A sends a message to the MINMEX, the MINMEX checks the given IDP and sends the message to the Brick owning this IDP. This allows that Bricks may be exchanged easily, and that IDPs may be redirected to point to other functions.

6 Summary and Further Work

6.1 Summary

This thesis is realized within the ANA project which builds an autonomic network based on a clean slate approach. During this thesis we have developed the underlying communication facilities between the two main objects in ANA: the MINMEX and the Bricks. Although no future developer will have to deal with the code written during this thesis, it will be used upon sending or receiving every packet.

In a second phase we have developed a bootstrapping process which allows different Bricks to learn about each other. We have implemented the first ANA compartment which shows that the ANA core software and the designed bootstrapping process operate seamlessly.

The following list summarizes the major contributions of this thesis to the ANA core development process.

- Implementation of the MINMEX as well as the framework for the Bricks in the Linux kernel space. They are stable and designed in a way, that future programmers do not have to write a single kernel specific line of code. The most interesting part is the communication interface between the MINMEX and the Bricks as well as the ANA thread library which allows a programmer with no kernel knowledge to use kernel threads.

- A short performance evaluation which compares the different communication modes implemented has shown that the KERN communication mode is by far the fastest. This clearly indicates the benefit when the ANA software runs in the kernel space.

- The provided code runs on any kernel 2.6 and on recent kernel 2.4. This allows an easy distribution of the ANA code, since the developers can stay with their actual kernel version and they do not have to compile a new kernel. The ANA code can even run on embedded devices, most of which run a kernel 2.4.

- Design of a flexible bootstrapping mechanism which allows Bricks on different nodes to learn about each other and to setup communication between them. The bootsrapping mechanism is designed in a way that it can be used from any Brick in ANA regardless whether it is an application or any other Brick in the networking stack. The bootstrapping process has the three options aggregation, cache and subscription. This allows to choose the most appropriate procedure for each situation.

- The chat application in combination with the Ethernet compartment work as a proof of concept for the ANA framework as well as for the bootstrapping process. Such demo applications are also useful for future programmers to see how programming in the ANA world works.

- I have learned a lot about the Linux kernel internals. Now I have a thorough understanding of the handling of interrupts, the deferring of work and the implications of putting a process to sleep as well as of generic

netlink, a mechanism to exchange data between the kernel and the user space. This knowledge I could pass to other ANA developers in a workshop, many discussions and even more emails.

Beside these main contributions, there are many minor contributions of this thesis in the context of the ANA project, ranging from debugging of code written by other developers until restructuring the ANA code repository or providing some guidelines for writing code in ANA.

6.2 Further Work

The ANA core is far from being finished. The most desirable enhancements are listed below:

- In order to make the life of future ANA Brick developers more enjoyable we have to provide a simpler API. Most of the developers will have their origin in user space application programming, and will therefore be used to the standard socket API. However the actual ANA API is quite different. The next API should therefore provide a functionality similar to today's socket API. The ANA coding workshop in Basel has shown that there is the need of some example applications running on ANA. They should clearly show how the API is to be used, what is similar to ordinary socket programming and what is special to ANA.
- At the moment the ANA playground is nearly empty. There are only some demo Bricks. Now we can start to populate the playground. We need some packet processing elements like routing Bricks etc. In order to reach the goal of autonomicity we need some monitoring Bricks.
- Up to now, the ANA core has not much of autonomicity in it. There is still the lack of a packet called "Information Flow". This packet will be responsible for setting up an optimal protocol stack in the ANA node, and to modify it depending on the current needs of the network.
- We have to review the bootstrapping process in order to be sure that it is the appropriate way to go. We have to think about whether the chosen bootstrapping process is generic enough and whether there is an easier way to go.
- In addition to the hardware abstraction layer, an ANA abstraction layer would be useful. It should allow legacy applications to run over ANA. Therewith we could evaluate the benefits of the ANA network architecture in comparison with the current Internet.

We hope to cover all these steps until the project ends in December 2009.

A Howto: Start using ANA

This appendix describes all steps necessary to start the chat application.

A.1 Compilation

1. Get the source code:
   `'svn checkout https://subversion.cs.unibas.ch/repos/ana/'`
   You will need a username and a password. These can be obtained from Christophe Jelger from the University of Basel.

2. Switch to the `trunk` directory and compile the program.
   `'cd ana-core/trunk/'`
   `'make user'` for a user space compilation
   `'make kernel'` for a kernel space compilation
   All the user space binaries will be copied in the `bin` directory and all the kernel modules in the `modules` directory.

A.2 Bricks Required for the Chat Application

1. vlink: Virtual Link support, up to now the only way to access the hardware.

2. eth-vl: Ethernet over vlink: Provides the bootstrapping functionality to find other chat partners.

3. chat: The chat application.

For the MINMEX, the vlink and the eth-vl Brick you can choose whether you want to run them in the Linux kernel or the Linux user space. The chat Brick has to run in user space, since it provides interaction with the user. In this example we will insert the MINMEX, the vlink and the eth-vl Brick as modules in the kernel and we will define KERN as communication mode. We will start the chat Brick in user space and we chose the communication mode GENETLINK.

A.3 Loading the MINMEX and the Bricks

For the modules we do not have to provide a special commandline argument, since the default values are already OK. Refer to section A.5 to see how to change the commandline arguments.
The vlink Brick has to be configured to send all packets to an Ethernet interface. A more detailed description on how to configure the vlink can be found in [48].
For the chat Brick we have to specify that we want to use GENETLINK as communication mode and we have to provide the “address” of the MINMEX.
Note that you need root rights in order to insert the modules.

```
'insmod modules/anaControl.ko'
'insmod modules/anaMinmex.ko'
'insmod modules/anaVlink.ko'
'./bin/vlconfig create 1'
'./bin/vlconfig add_if vlink1 eth0'
```

```
'./bin/vlconfig up vlink1'
'insmod modules/anaVEthBrick.ko'
'./bin/chat -c genetlink://123 -d genetlink://123 -n genetlink://ANA -D ANA_ERR'
```

This should start your ANA chat client and you can start sending messages to other people running the chat client on the same Ethernet segment.

A.4 Troubleshooting

- `'error while loading shared libraries:...'`
  You need to specify the path to the ANA libraries:
  Setting it for the current shell:
  `'export LD_LIBRARY_PATH=/path/to/ana/ana-core/trunk/lib:${LD_LIBRARY_PATH}'`
  To set it system wide, add the ANA library path to the file `/etc/ld.so.conf` and type `'ldconfig'` in the shell. (You need to be root).

- Brick starts, but claims something like
  `'Unable to attach to ANA minmex'.`
  - Some permissions may be violated. If the MINMEX runs as root, the Brick has to run as root too (only if UNIX is chosen as a communication mode).
  - The chosen communication modes may be incompatible. E.g. if the MINMEX runs in kernel space, a Brick cannot choose UNIX as communication mode.

- The vlink Brick does not work.
  You need to run it with root privileges and you need to configure it with the vlconfig tool.

A.5 Configuring the MINMEX and the Bricks

The MINMEX supports the following commandline arguments:
-c: control gate
-d: data gate
-h: handle size in bytes (default 2)
-l: label size in bytes (default 2)
-b: mtu for messages between client and node (default 200)
-C: hash Mask size and size of client table (default 2^6)
-T: hash Mask size and size of the IDT (default 2^{10})
-D in user space and debug in kernel space: DebugLevel, with one of the following values:
ANA_NONE: no messages
ANA_EMERG emergency, something really bad happened, brick/MINMEX may crash, be unusable
ANA_ERR error condition, system drops a packet, a lookup fails
ANA_NOTICE normal situation, worthy to note
ANA_DEBUG debugging information

Each Bricks supports the following commandline arguments:
-d: data gate

-c: control gate
-n: control gate of the MINMEX to connect to
-D in user space and debug in kernel space: DebugLevel

In user space the commandline arguments have to be specified in the form `[hyphen][optionname][space][optionvalue]`:
`'./bin/minmex -D ANA_ERR'.`
For a module this is different: `[optionname][equal sign][optionvalue].`
`'insmod modules/anaMinmex debug=ANA_ERR'`

Generally each kernel module provides a list with all supported arguments along with a description of these arguments. This list can be accessed with the following command:
`modinfo path/to/module.ko`
Commandline arguments are generally specified with the following schema:
`insmod path/to/module.ko varName='Value' arrayName='Value1','Value2'`

A.6 Addressing Schemes

Each communication mode has a different scheme to identify the "address" on which it listens for data or control messages respectively:

KERN	"kern://name"
GENETLINK	"genetlink://name"
UDP	"udp://ip:port"
UNIX	"unix://path/to/file"
PIPES	"pipe://path/to/pipe"

If no gate is specified on the command line a default gate is chosen. In userspace it is a UNIX socket and in kernel space it the KERN mode.
Upon running the MINMEX two different communication gates need to be specified: One to receive data and one to receive control messages.
For **KERN** and **GENETLINK** mode the "name" is not important. For KERN it corresponds to the module name which is obtained automatically and for GENETLINK it is hard coded in the source code. For **UDP** mode the urn specifies on which IP address/port pair the MINMEX is waiting for data or control information.

Bricks have to specify three gates: one for receiving control information, one for receiving data messages and the control gate of the MINMEX they want to send messages to.
For **GENETLINK** the "name" does not matter for the Bricks control and data gate. The MINMEX control gate has to be specified to be "ANA".
In **KERN** mode the names of the data and control gate are used as an identifier to distinguish from which MINMEX a message was arrived. But generally a Brick is only attached to one MINMEX and therefore this value can be omitted. The name of the MINMEXs control gate needs to be the module name of the MINMEX, e.g. "anaMinmex".

B First Steps in Writing Code for ANA

Everything presented in this chapter is based on svn revision 230. It may be possible (likely) that there will be several changes, for example to the wrapper functions or to the template for the Bricks.

B.1 Guidelines for Writing System Agnostic Code

This appendix lists some guidelines for writing code that should run in the Linux user and in the kernel space.

1. Programming language: C.
2. Variables have to be declared at the beginning of a block.
3. `void` has to be explicitly mentioned in the parameter list of a function declaration.
4. Global variables cannot be declared in a header file. They need to be defined in one file and the other files need to use the keyword `extern`.
5. Avoid user space specific functions such as bzero() or strtok(). Use memset() or strsep() instead which are available in both environments.
6. The ANA core provides a library of wrapper functions, which makes system specific functions (e.g. memory allocation) accessible for kernel and user space:
   - int atoiWrapper(char *mychar);
   - int inet_addrWrapper(const char *cp);
   - void freeWrapper(void *pointer);
   - void *mallocWrapper(int size);
   - int ANAsleep(int seconds);
   - int randomWrapper(void);
   - int gettimeofdayWrapper(struct timeval *tv, struct timezone *tz);
   - socket manipulation wrapper functions. These are specific to the ANA core:
     - int setSockOptWrapper(void *sock, int level,int op, char * optval, int optlen);
     - int readWrapper(void *sock, char * data, int len);
     - int sendSocketWrapper(void *sock, int domain, void *aux, void *msg, int len);
     - void closeWrapper(void * sock);
     - int sockCreateWrapper(void *toReturnSock, int domain, int type, int protocol);
     - int bindWrapper (void * sock, struct sockaddr * address, int size);
7. anaPrint(debugLevel, “format string”): A macro to be used for printing messages.

8. The ANA core provides a “triple Makefile structure”: There exist dedicated Makefiles for userspace, kernel 2.6 and kernel 2.4. In order that a Brick gets compiled in each of these three environments all of the Makefiles have to be adapted.

B.2 Writing Your First Brick

In this section we give some hints on how to start writing your own Bricks.

- If you want to develop a Brick which is intended to run mainly in kernel space you should obtain an actual Linux kernel source code (e.g. 2.6.20) and compile your kernel with as many debugging options as possible.
- Obtain the ANA source code:
  `‘svn checkout https://subversion.cs.unibas.ch/repos/ana/’`
  You will need a username and a password. These can be obtained from Christophe Jelger from the University of Basel.
- Read the ANA core documentation in the `ana-core/trunk/doc` directory. This is important! It provides you with all the necessary information on the concepts of ANA and it introduces the API as well as the wrapper functions provided.
- Switch to the Bricks directory and create your own directory.
  `‘cd ana-core/trunk/C/bricks’`
  `‘mkdir myFirstBrick’`
- Copy the file `examples/brickAL0_1.c` as well as the three Makefiles in your directory.
  `Makefile` is the kernel 2.6 Makefile, `Makefile-24` the kernel 2.4 Makefile and the `Makefile-user` the one for the user space.
  The `brickAL0_1.c` file contains a minimal Brick which sends a message via the MINMEX to itself.
- Adapt the Makefiles in your directory, and the ones in the `C/bricks/` directory.
- Open your copy of the file `brickAL0_1.c`. Be not scared because of the code you see. Most of it you will not have to understand.
- Change `mymodename` into the name of your Brick (e.g. myBrick). This string will be printed in front of each `anaPrint()` in user space. In kernel space, the name of the Brick as obtained by `THIS_MODULE->name` is printed in front of each message.
- The execution of your Bricks code will start in the `brick_start()` function. In this function you can start to register callback functions, query the MINMEX for some other Bricks etc.
- Upon removing the Brick (by typing Ctrl-C or rmmod moduleName) the `brick_exit()` function gets executed. In this function you should unregister all your callback functions and free all memory etc.

- You will rarely have to change anything below the `/* Do not change anything below here, unless you know exactly what you do! */` mark. This area covers the user space and kernel space specific initialization of the Brick. It parses the commandline arguments and attaches the Brick to the MINMEX.
- The necessary steps to compile and run your Brick are described in Appendix A. The compilation has to be started in the `'ana-core/trunk'` directory.
- New wrapper functions should be declared in the file `trunk/C/include/anaCommon.h` and implemented in `trunk/C/shared/anaCommon.c`.
- Some coding guidelines:
  - A callback function should return as fast as possible. If you need to perform a lengthy action start a dedicated thread. As long as a callback function gets executed no further messages can be processed.
  - Callback functions may get invoked at "random times". Therefore you will have to carefully lock all data structures used by callback functions and threads.
  - You should not stay in `brick_start()` for "a long time". This will block the shell upon inserting the kernel module.
  - Carefully code the removement of your Brick. Errors upon removing a Brick may crash your kernel...
- Write and test your code carefully, since the kernel space is much more sensitive to programming mistakes than the user space.

C Important Data Structures

This Appendix describes the management units used by the MINMEX as well as by the Brick in detail. It lists all structs used and explains the elements of these structs. Note that the described structs correspond to svn revision 100. The elements may have changed in the meantime. Especially the names will be different. All elements called “client” have changed to “brick” and most elements called “node” have changed to “minmex” (except for the elements related to the node compartment).

C.1 MINMEX

The MINMEX has two management tables, one to save all information belonging to Bricks and one for all IDPs, as well as two linked lists to save all different communication capabilities. The first list holds all control communication gates whereas the second list holds all data communication gates. Each element of these lists is of type **commGate_s**.

```
struct commGate_s {
        int mode;                                       (1)
        int descriptor;                                 (2)
        struct socket * commGateSocket;                 (3)
        struct anaNodeFunctions * commGateFunctions;    (4)
        char *urn;                                      (5)
        struct commGate_s *next;                        (6)
};
```

1. Mode for this gate: UDP, UNIX, PIPES, GENETLINK, KERN.
2. File descriptor to identify this gate. Used in userspace for UDP and PIPES, used in kernel space for GENETLINK to hold the family ID.
3. Used in kernel space to identify the socket, used for UDP and GENETLINK, unused for KERN.
4. Pointer to a struct anaNodeFunctions which holds pointers for each ANA MINMEX function that can be called from a Brick. Only for the KERN mode.
5. URN of this gate, e.g. for UDP: udp://127.0.0.1:6666, for KERN it is the name of this module.
6. pointer to the next gate, used when more than 1 gate is registered

The **client table** holds the information of each Brick attached. It is a hash table with a size defineable upon program start up. There is a global variable CT of type `ClientTable_t` which holds some management information for the client table along with a pointer to an array holding the actual Bricks. Each entry of the client table is of type `CTentry_s`.

```
typedef struct {
        unsigned int size;       (1)
        int hashMaskSize;        (2)
        int handleSize;          (3)
        int nbClients;           (4)
        struct CTentry_s **tab;  (5)
} ClientTable_t;
```

1. maximum numbers of Bricks pow2 of `hashMaskSize`
2. size of the mask used in hash function, can be specified with the -C parameter upon program start
3. defines the size of the keys (e.g. the size of anaHandle_t), can be specified with -h parameter
4. number of active entries in the table
5. pointer to the actual table, member ”i” of this table can be accessed in the following way:

   ```
   ClientTable_t CT;
   struct CTentry_s *tmp = CT.tab[i]
   ```

```
struct CTentry_s {
        anaHandle_t handle;            (1)
        int dataMode;                  (2)
        int dataDescriptor;            (3)
        struct socket * dataSocket;    (4)
        void *dataAux;                 (5)
        int controlMode;               (6)
        int controlDescriptor;         (7)
        struct socket * controlSocket; (8)
        void *controlAux;              (9)
        int age;                       (10)
        anaLabel_t notificationLabel;  (11)
        struct CTentry_s *next;        (12)
};
```

1. key of the hash table, identifier of the Brick
2. mode over which data has to be sent to this Brick (UDP, UNIX, PIPES, GENETLINK, KERN) defined in `include/anaCommon.h`
3. UDP and PIPES in user space: file descriptor used to send data messages to this Brick, GENETLINK: uid of this Bricks data gate
4. socket for sending data messages to this Brick, used for UDP in kernel space
5. complementary argument depending on the mode. For UDP this is a sockaddr_in, for KERN this is the function pointer on which the Brick waits for data, GENETLINK and PIPE do not use it
6. mode over which control information has to be sent to this Brick (UDP, UNIX, PIPES, GENETLINK, KERN) defined in `include/anaCommon.h`
7. UDP and PIPES: file descriptor used to send control information to this Brick, GENETLINK: uid of this Bricks control gate
8. socket over which control messages are sent used for UDP in kernel space,
9. Complementary argument depending on the mode. For UDP this is a sockaddr_in, not used for KERN, GENETLINK and PIPE
10. counts the number of unacknowledged heart beat requests in a series, set back to zero on reception of a HB request from the Brick
11. IDP used by the MINMEX to send unsolicited notifications to the Brick (e.g. tell the Brick that the MINMEX will go down)
12. in case of a hash conflict all entries are put in a linked list

The **information dispatch table** holds the information on the registered information dispatch points (IDPs). Since in user space the MINMEX runs in a different process than the Brick it is not possible to let the MINMEX map the IDP directly to a function. Instead the MINMEX redirects data received for a given IDP to the corresponding Brick. There is a global variable IDT of type `IDT_t` which holds some management information for the information dispatch table along with a pointer to the information dispatch table. Each entry in this array is of type `IDTentry_s`.

```
typedef struct {
        unsigned int size;              (1)
        int hashMaskSize;               (2)
        int labelSize;                  (3)
        int handleSize;                 (4)
        int deadEntryLimit;             (5)
        struct IDTentry_s **tab;        (6)
} IDT_t;
```

1. number of entries in the information dispatch table, pow2 of `hashMaskSize`
2. size of the hash mask, definable upon program start with the -T parameter
3. size in Bytes of the labels stored in this table (-l parameter)
4. size in Bytes of the handles stored in this table (-h parameter)
5. an entry not used for more than deadEntryLimit heart beat checks time intervals will be deleted
6. a pointer to the entries of the IDT

```
struct IDTentry_s {
        anaLabel_t label;              (1)
        int private;                   (2)
        anaHandle_t handle;            (3)
        nodeCallback_t callBackFct;    (4)
        void *aux;                     (5)
        int age;                       (6)
        struct IDTentry_s *next;       (7)
};
```

1. Information Dispatch Point (IDP): key of the hash table, used to identify a function registered by a Brick
2. flag indicating if the IDP is private or shared with other Bricks, 0 = shared, 1 = private. Private entries are only shown to the Brick which has registered this IDP
3. indicates the owner of the IDP (corresponds to entry (1) of `CTentry_s`)
4. function handling the IDP (e.g. the function which sends the data to the corresponding handle)
5. auxiliary argument, in the current implementation it equals the handle of the owner of this IDP
6. incremented upon each heartbeat check, reset upon using this IDP. A value of -1 means that this entry is permanent and will not be removed.
7. in case there is a hash conflict each entry is put in a list

C.2 Brick

The Brick manages in a linked list all the MINMEXs it is connected to. For each MINMEX there is an element of type struct `anaNodeSpecs_s`, which saves all the information necessary for the communication with this MINMEX.

```
struct anaNodeSpecs_s {
        anaHandle_t handle;                                (1)
        anaLabel_t nodeCompLabel;                          (2)
        anaLabel_t notificationLabel;                      (3)
        int labelSize;                                     (4)
        int handleSize;                                    (5)
        int mtu;                                           (6)
        int myDataMode;                                    (7)
        int myDataDescriptor;                              (8)
        struct socket * myDataSocket;                      (9)
        struct anaClientFunctions * myDataFunctions;       (10)
        void *myDataAux;                                   (11)
        int myControlMode;                                 (12)
        int myControlDescriptor;                           (13)
        struct socket * myControlSocket;                   (14)
        void *myControlAux;                                (15)
        int nodeDataMode;                                  (16)
        int nodeDataDescriptor;                            (17)
        struct socket * nodeDataSocket;                    (18)
        void *nodeDataAux;                                 (19)
        int nodeControlMode;                               (20)
        int nodeControlDescriptor;                         (21)
        struct socket * nodeControlSocket;                 (22)
        void *nodeControlAux;                              (23)
        int age;                                           (24)
        void *buffer;                                      (25)
        struct anaNodeSpecs_s *next;                       (26)
        struct shadowIDT_s *shadowT;                       (27)
        unsigned char *attachMessage;                      (28)
};
```

1. The handle chosen by the MINMEX to identify the Brick
2. label identifying the IDP to reach the MINMEX compartment via the data link
3. This label is used by the MINMEX to notify the Brick of an unsolicited information. E.g. used in the heartBeat process to receive the heartBeat reply from a MINMEX. This label is generated by the API0 at the attach process
4. size of labels for this MINMEX
5. size of handles for this MINMEX
6. maximum size of messages
7. mode over which this Brick receives data (must be value specified `in anaCommon.h`)
8. file descriptor over which Brick receive data messages (used in user space)

10. socket over which Brick receive data messages (used in kernel space)
11. function which receives data message (for KERN mode)
12. complementary argument depending on the mode: GENETLINK: uid, PIPES: dummy descriptor and timeout value
13. mode over which this Brick receives control information
14. file descriptor over which this Brick receives control messages (used in user space)
15. socket over which this Brick receives control messages (used in kernel space)
16. complementary argument depending on the mode: GENETLINK: uid, PIPES: dummy descriptor and timeout value
17. mode over which the MINMEX receives data messages
18. file descriptor over which data messages has to be sent (used in user space)
19. socket over which data data messages have to be sent. (used in kernel space)
20. complementary argument depending on the mode: UDP: struct sockaddr_in, GENETLINK: family id
21. mode over which the MINMEX receives control messages
22. file descriptor over which control information is sent (use in user space)
23. socket over which control information is sent to the MINMEX (used in kernel space for UDP)
24. complementary argument depending on the mode: UDP: struct sockaddr_in, GENETLINK: family id, KERN: pointer to the MINMEX functions,
25. age of this MINMEX (incremented from the heartbeat check), reinitialized to zero on MINMEX notification reception. Once the age is above a threshold, the entry is deleted
26. buffer for data reception, will be allocated according to the MTU
27. pointer to the next MINMEX
28. shadow dispatch table associated with MINMEX
29. This String is built according to the information above by the `anaL0_initNodeSpecs` function, it is sent to the MINMEX by the `attach()` function

The **shadow dispatch table** maps the callback functions registered with a MINMEX to the IDP (label) chosen by that MINMEX. This implies that the Brick has to know from which MINMEX a message was received, in order to look up the IDP in the corresponding shadow dispatch table.
The shadow dispatch table is of type `shadowIDT_s` and its entries are of type `SDTentry_s`.

```
struct shadowIDT_s{
        unsigned int size;          (1)
        int hashMaskSize;           (2)
        int labelSize;              (3)
        struct SDTentry_s **tab;    (4)
} ;
```

1. size of the shadow information dispatch table
2. size of the hash mask, is log2 of table size
3. size of ANA labels, keys of the table
4. table holding the entries

```
struct SDTentry_s {
        anaLabel_t label;           (1)
        anaCallback_t callBackFct;  (2)
        void *aux;                  (3)
        struct SDTentry_s *next;    (4)
};
```

1. the IDP chosen by the MINMEX
2. the function corresponding to the IDP
3. auxiliary argument to callBack function
4. in case there is a hash conflict the entries are put in a linked list

References

[1] R.D. Pethia, Computer Security, Testimony Before the Committee on Government Reform Subcommittee on Government Management, Information, and Technology on March 9, 2000

[2] M. Handley, and A. Greenhalgh, Steps Towards a DoS-Resistant Internet Architecture, Proceedings of ACM SIGCOMM Workshop on Future Directions in Network Architecture, 2004

[3] Requirements Analysis, Sixth Framework Programm, Project Number: FP6-IST-27489, Deliverable 1.2, Thomas Plagemann et al, 8.11.07

[4] A. Nakao, L. Peterson, and A. Bavier, A Routing Underlay for Overlay Networks, Proceedings of 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2003

[5] L. Peterson, S. Shenker, and J. Turner, Overcoming the Internet Impasse through Virtualization, Proceedings of 3rd Workshop on Hot Topics in Networks (HotNets-III), 2004

[6] B. Jacob, R. Lanyon-Hogg, D.K. Nadgir, and A.F. Yassin, A Practical Guide to the IBM, Autonomic Computing Toolkit (Book Title), IBM Corporation, April 2004

[7] State of the Art, Sixth Framework Programm, Project Number: FP6-IST-27489, Deliverable 1.1 Christophe Jelger et al, 30.6.2006

[8] Future and Emerging Technologies, http://cordis.europa.eu/ist/fet/ (21.06.07)

[9] Situated and Autonomic Communications (SAC), http://cordis.europa.eu/ist/fet/comms.htm (13.09.07)

[10] Global Environment for Networking Innovations (GENI), http://www.geni.net/ (27.08.07)

[11] Inside AppleTalk, Gursharan S. Sidhu, Richard F. Andrews, Alan B. Oppenheimer developer.apple.com/MacOs/opentransport/docs/dev/Inside_AppleTalk.pdf (20.08.07)

[12] NetBios, NetBEUI, NBF, SMB, CIFS Networking, http://timothydevans.me.uk/nbf2cifs/book1.html (26.08.07)

[13] The Official Samba 3.2.x HOWTO and Reference Guide, 10. NetworkBrowsing http://www.samba.org/samba/docs/man/Samba-HOWTO-Collection/NetworkBrowsing.html (26.08.07)

[14] Web Services Architecture, W3C Working Group Note 11 February 2004, http://www.w3.org/TR/ws-arch/ (21. 08.07)

[15] Zero Configuration Networking (Zeroconf), http://www.zeroconf.org/ (21.08.07)

[16] DNS Service Discovery (DNS-SD), http://www.dns-sd.org/ (27.08.07)

[17] http://developer.apple.com/opensource/internet/bonjour.html (21.08.07)

[18] http://avahi.org/ (21.08.07)

[19] DNS-Based Service Discovery, Stuart Cheshire, Marc Krochmal, Apple Computer, Inc, 10th August 2006 http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt (21.08.07)

[20] http://www.jini.org

[21] http://www.artima.com/jini/jiniology/lookup.html

[22] BOOTSTRAP PROTOCOL (BOOTP), rfc951

[23] Dynamic Host Configuration Protocol, rfc 2131

[24] Multicast dns http://www.multicastdns.org/ (26.08.07)

[25] UPnP Device Architecture, July 20, 2006, http://www.upnp.org/specs/arch/UPnP-DeviceArchitecture-v1.0-20060720.pdf (26.08.07)

[26] ETH Zurich, Computer Engineering and Networks Laboratory, 8092 Zurich, www.tik.ee.ethz.ch

[27] Universität Basel, Departement Informatik, 4056 Basel, http://informatik.unibas.ch

[28] ANA Blueprint, Sixth Framework Programm, Project Number: FP6-IST-27489, Deliverable D1.4/5/6v1, Christophe Jelger et al, 15.02.07

[29] Active Technologies, Vorlesungsslides von B. Plattner, SS 2007

[30] Ambient Networks Project, http://www.ambient-networks.org/ (21.06.07)

[31] Bison: Biology-Inspired techniques for Self-Organization in dynamic Networks, http://www.cs.unibo.it/bison/ (21.06.07)

[32] BIONETS: bio-inspired service evolution of the pervasive age: http://www.bionets.eu/ (21.06.07)

[33] Haggle: A European Union funded project in Situated and Autonomic Communications, http://www.haggleproject.org/ (21.06.07)

[34] CASCADAS: Bringing Autonomic Services to Life, http://www.cascadas-project.org/ (21.06.07)

[35] ANA: Autonomic Network Architecture, http://www.ana-project.org/(21.06.07)

[36] Why and How to Use Netlink Socket, Kevin He, http://www.linuxjournal.com/article/7356 (21.06.07)

[37] Patch: Generic Netlink HOW-TO based on Jamal's original doc, http://lwn.net/Articles/208755/ (10.11.2006)

[38] Generic Netlink HOWTO,
http://linux-net.osdl.org/index.php/Generic_Netlink_HOWTO (21.06.07)

[39] The Click Modular Router Project, http://read.cs.ucla.edu/click/ (21.06.07)

[40] Click concepts tutorial: a general overview of concepts in Click, http://www.pats.ua.ac.be/content/software/click-1.5/concepts.pdf (26.08.06)

[41] libnl - netlink library, http://people.suug.ch/ tgr/libnl/ (21.06.07)

[42] Wikipedia Autonomic Networking,
http://en.wikipedia.org/wiki/Autonomic_Networking (21.06.07)

[43] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Driver 3rd Edition, O'Reilly, 2005

[44] Kernel Module Programming: Examples,
http://people.ee.ethz.ch/ arkeller/ma (21.06.07)

[45] About IBM autonomic computing,
http://www-306.ibm.com/autonomic/about.shtml (21.06.07) The Vision of Autonomic Computing, The Vision of Autonomic Computing, 2003 IEEE, http://www.research.ibm.com/autonomic/manifesto/

[46] OpenWrt, http://www.openwrt.org (13.08.07)

[47] ANA code repository,
https://subversion.cs.unibas.ch/repos/ana/ (13.08.07)

[48] ANA core documentation, https://subversion.cs.unibas.ch/repos/ana/ana-core/trunk/doc/anacore-doc.pdf