Dynamic Protocol Stacks

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Science

presented by
ARIANE TRAMMELL-KELLER

Master of Science ETH in Electrical Engineering and
Information Technology
born April 10, 1982
citizen of Sarmenstorf (AG)

accepted on the recommendation of
Prof. Dr. Bernhard Plattner, examiner
Prof. Dr. Marco Platzner, co-examiner
Dr. Martin May, co-examiner

2014
Abstract

Many people around the globe enjoy the amenities of a ubiquitously available communication infrastructure. This infrastructure is used in a variety of scenarios: It is used for personal communication with friends or business partners, to obtain information or entertainment from a service provider, or for buying things without being physically present in a shop. Additionally, the same infrastructure is used by electronic devices to share information. For example, sensor nodes measure temperature, humidity, air pollution etc., and share this information with a central server which gathers aggregated knowledge from the individual measurement points. Or different local sites of a company might exchange a high data volume to synchronize and / or back up their data base.

Each of these applications impose different communication requirements on the infrastructure, ranging from different levels of security and privacy, to different allowable transmission delays, to different requirements on the transmission reliability. However, all these applications run over the same network architecture, the Internet architecture. Currently, many add-ons to the original architecture make this possible. However, it is questionable whether this add-on approach is scalable an can accommodate future applications. Therefore, many researchers develop new network architectures from scratch, so called clean slate architectures, that are designed to be more appropriate for future challenges.

In this dissertation we have developed such a clean slate network architecture called *Dynamic Protocol Stack (DPS) architecture*. The DPS architecture tackles the challenge of providing a future-oriented architecture by introducing flexibility at two levels. First, newly
developed communication protocols can be included seamlessly in the architecture and second, the protocol stack used for the communication can be adapted to the current environment and communication needs while transferring data.

However, such a flexible network architecture poses a new challenge: How can it be implemented without suffering from performance degradation introduced by the flexibility? We address this challenge by providing two different implementations of the DPS architecture. With the first implementation we show how the DPS architecture can be implemented on a general-purpose CPU, and we show that it has the same performance as an implementation of the Internet architecture on the same system. With the second implementation we show how the DPS architecture can make use of hardware acceleration for the individual network protocols. Therefore, we implemented the DPS architecture on an FPGA-based system-on-chip platform. This implementation, called EmbedNet, allows for the dynamic run-time mapping of network functionality to either software or hardware. Thus it offers hardware acceleration for arbitrary network functionality in the DPS architecture as dedicated hardware for a specific network protocol would in the Internet architecture.

Additionally, we show in two use cases the benefits of having a dynamic network architecture. First, we show how we can minimize protocol overhead by adapting the protocol stack to the current network conditions. Second, we show how we can improve the system performance when adapting the hardware / software mapping to the current network traffic. Finally, we discuss how more advanced optimization algorithms could be developed to optimize the provided network functionality.

To summarize, with this dissertation we have shown that it is feasible to provide dynamic network architectures and our results suggest that it is worthwhile to go further in that direction.
Kurzfassung


All diese Anwendungen stellen die verschiedensten Anforderungen an die Kommunikationsinfrastruktur, wie zum Beispiel unterschiedliche Bedürfnisse zum Sichern der übertragenen Informationen, verschiedene maximal zulässige Übertragungszeiten oder verschiedene Anforderungen an die Zuverlässigkeit der Datenübertragung.

Trotz der unterschiedlichen Anforderungen verwenden alle Anwendungen dieselbe Netzwerkarchitektur: die Internet-Architektur. Im Laufe der letzten Jahre wurde die Internet-Architektur an vielen Stellen erweitert, um neuartigen Anforderungen gerecht werden zu können. Es ist aber fraglich, ob dieser Ansatz auch in Zukunft den Anforderungen immer neuer Anwendungen genügen kann. Deshalb beschäftigen sich viele Forscher damit, radikal neue Kommunikationsarchitekturen zu entwerfen. Diese Architekturen sollen besser auf zukünftige Anforderungen reagieren können.


Zusammenfassend verdeutlichen wir mit dieser Doktorarbeit, dass es möglich ist, dynamische Netzwerkarchitekturen zu entwickeln und dass es vielversprechend ist, weiter in diese Richtung zu forschen.
## Contents

Abstract iii

Kurzfassung v

1 Introduction 1
   1.1 From Active Networking to Future Internet Research . . 2
   1.2 Flexible Network Architectures ....................... 4
   1.3 Contributions ........................................ 7
   1.4 Scope ................................................ 8
   1.5 Thesis Structure ....................................... 9

I Flexible Network Architectures 11

2 The Autonomic Network Architecture 12
   2.1 Node Architecture .................................... 12
   2.2 Protocol Stack Setup ................................ 14
   2.3 Node Local Protocol Stack Adaptation ................ 17
   2.4 Inter Node Protocol Stack Adaptation ................ 19

3 The Dynamic Protocol Stack Architecture 21
   3.1 Node Architecture .................................... 21
   3.2 Protocol Setup ....................................... 23
   3.3 Node Local Protocol Stack Adaptation ................ 31
   3.4 Inter Node Protocol Stack Adaptation ................ 32
4 Comparison of ANA and the DPS Architecture 34
5 Section Conclusion 37

II Execution Environments for Dynamic Protocol Stacks 38
6 Packet Processing Execution Environments 39
7 DPS on a General-Purpose Processor 42
  7.1 Architecture ........................................ 42
  7.2 Performance Evaluation .............................. 49
8 DPS with Hardware Accelerators 51
  8.1 EmbedNet Architecture .............................. 52
  8.2 EmbedNet Hardware Design ......................... 58
  8.3 Performance Evaluation .............................. 65
9 Section Conclusion 72

III Case Studies with Dynamic Adaptation 73
10 Autonomic Adaptation of the Protocol Stack Functionality 74
11 Autonomic HW/SW Mapping in EmbedNet 78
  11.1 Mapping Algorithms ................................. 80
  11.2 Network Traffic Mixes ............................... 81
  11.3 Experimental Results ............................... 81
12 Section Conclusion 87

IV Towards Self-Awareness in Networking 88
13 Adaptation Strategies 89
14 Self-Awareness in Networking  
14.1 Proprioceptive Framework .......................... 92  
14.2 Levels of Self-Awareness ......................... 94  

15 Section Conclusion  

V Conclusions  

16 Comparison with Related Work  
16.1 Future Internet Architectures ....................... 99  
16.2 Network Processing with FPGAs ................... 102  

17 Review of Contributions  

18 Critical Assessment  

19 Future Work and Outlook  

Abbreviations  

Acknowledgements
Regular computer users enjoy the amenities of today's computing devices. They stay connected with their friends over social media platforms, emails and voice over IP; they quickly access information about latest news, restaurants and movies and they entertain themselves by watching YouTube videos or reading Wikipedia. These services are available at home, in the office and even while traveling. The fewest users wonder how these services are actually provided - as long as everything works they are happy. However, behind the stage huge efforts are made to provide high quality services. This ranges from configuring applications and routing algorithms over supporting enough hardware for redundant operation to developing new protocols to address new communication requirements. Especially the newly developed protocols address many problems of the base Internet design: The lack of built in security, the exhaustion of the address space and the lack of support for mobile devices. Some patches to the Internet architecture such as e.g., CIDR [14] and NAT [15] were applied smoothly, but broke the underlying design principles of the Internet. Others, such as IP multicast [16], IP mobility [17] and IP version 6 [18] have a hard time to be deployed.

Therefore, a growing number of researchers is concerned whether the Internet can adapt itself to the challenges of future communication requirements [19,20] and is investigating into alternative, clean slate, approaches [21–25]. Already in the early ages of the Internet new, more flexible architectures were proposed.
1.1 From Active Networking to Future Internet Research

Already in the early 1990s, as the Internet rapidly grew and became more and more popular, researchers were unhappy with the static TCP/IP architecture [26,27]. Tennenhouse and Wetherall [28] proposed Active Networks in which users could inject custom code into the network. This code would be associated with a set of packets that traversed the network from the source over several routers to the destination. The code would be executed on intermediate nodes and would modify the packets on the fly as desired. Tennenhouse and Wetherall suggested four different possibilities for active packet processing:

- Network operators inject code on the intermediate nodes.
- Every packet contains the program code to be executed.
- Packets can put code into a node and other packets could use that code.
- Packets contain a reference to code on an external server and the routers download the code from that server and store it in a local cache.

Based on those four possibilities many researchers worked on specific active networking architectures in the following ten years. This work is summarized by several survey papers [29–32]. However, none of the active network architectures found its way into the commercial Internet, some because of performance issues, some because of security concerns and some because of the lack of supporting hardware.

Even though active networks were not integrated into the commercial Internet, research on a better network architecture continued. This research is driven by the limited extensibility, security and reliability of the Internet architecture coupled with a paradigm change from a static, provider-centric network to a mobile, user-centric network. Several efforts tackle these issues on a case by case basis, for example with IPv6 [18], Network Address Translation (NAT) [15], virtual private networks (VPNs) [33], and an extensive number of new routing or transmission protocols for mobile networks such as mobile IP [17] or hop by hop transmission [34].

Over the last decade, the new term Future Internet was introduced. Under this umbrella researchers not only investigate in additions and
patches to the Internet architectures, but they also investigate in *clean slate architectures* where they are not bound to the Internet architecture but where a completely new architecture can be developed. Within the European Union, future Internet research has a high priority. Alone in the European Union Seventh Framework Programme (EU-FP7) over 100 projects are funded in the area of future Internet [35] and the European Future Internet Assembly (FIA) brings together around 150 research projects [36]. These projects are diverse ranging from cloud computing over trustworthy information and communication technology (ICT) to socio-economic considerations and network architectures.

Within the Future Internet Research and Experimentation (FIRE) initiative [24], the EU has spent over 100 million Euro to support highly innovative and revolutionary experimental research on the future Internet, even though these concepts might only be applied in the long term. Similar initiatives are going on in the US [23,37] and Japan [38].

Those numbers clearly show that not only researchers, but also governments believe in the need to fundamentally re-engineer the Internet architecture in order to ensure appropriate connectivity in the future. However, the large number and variety of projects show that there is no consensus yet as of how this architecture should look like; rather all directions are researched with the hope of finding an optimal network architecture.

Many projects focus on the development of a network architecture that is optimized for a specific goal (such as maximise security or throughput, minimize power consumption, support mobility, or support communication over a network that lacks continues connectivity). Such optimization approaches usually lead to a static architecture which works very well for the chosen optimization goal or application scenario, but often they are inappropriate for the use in a wider range of scenarios, difficult to adapt to new requirements, and unable to cope with changing communication requirements.

In contrast to such static network architectures, we look at *flexible network architectures* in this dissertation. A flexible network architecture is a network architecture that can adapt the packet processing functionality according to user and application needs as well as packet transmission characteristics.
1.2 Flexible Network Architectures

In contrast to the Internet architecture that specifies exactly which protocols to use on which layer, a flexible network architecture is a meta architecture that describes how protocol stacks can be built according to application needs. Flexible network architectures provide the following benefits over the traditional Internet architecture:

- Allow for multiple network architectures to run in parallel.
- Allow for the optimization of the protocol stack.
- Allow for a dynamic reconfiguration of the protocol stacks.

1.2.1 Multiple Network Architectures

Since a flexible network architecture is only a meta architecture, it allows for having several network architectures to be present on a single network node simultaneously. If we look at the Internet architecture as it is deployed nowadays a different picture arises (see Figure 1.1(a)). Most applications make use of the protocols TCP [39] or UDP [40] and IP (either version 4 [39] or version 6 [18]). Innovation mostly happens below the Internet layer, e.g., to cope with different transmission media, or above the transport layer e.g. to provide additional services to applications such as security.

In a flexible network architecture (see Figure 1.1(b)), several, completely different network architectures coexist. Such network architectures might be tailored to sensor networks, offer new communication paradigms such as content centric routing, or even be the well known TCP/IP architecture.

1.2.2 Optimization of Protocol Stack

Instead of providing a single protocol stack that has to be used by all applications, in a flexible network architecture, the protocol stack can be built to provide exactly the required functionality. Therefore, flexible network architectures split the network functionality into individual building blocks that can be combined as required. There is no strict association of network functionality or even a protocol to
1.2 Flexible Network Architectures

Figure 1.1: Comparison of the Internet architecture (a) with a flexible network architecture (b).

(a) Internet Architecture    (b) Flexible Network Architecture

Figure 1.1: Comparison of the Internet architecture (a) with a flexible network architecture (b).

a specific layer as this is the case in the Internet architecture. This comparison is shown in Figure 1.2.

The flexible configuration of the protocol stack has the following three benefits:

- The user experience can be enhanced if the protocol stack offers exactly the requested quality of service as opposed to general-purpose functionality, since the requirements for real time applications such as video conferencing and bulk data transfer are rather different.

- Resources can be saved, by building a protocol stack that uses only the minimum set of required functionality, instead of a default protocol stack. This is of special interest in resource-constrained devices such as mobile phones or in high performance compute clusters, where a high data transmission rate should be achieved while minimizing the power consumption.

- Novel network protocols can be evaluated easier and successful innovations can be adapted easier on a bigger scale without the need of changing the operating system.
1.2.3 Dynamic Protocol Stack Reconfiguration

Flexible network architectures also offer the possibility to dynamically change a protocol stack that is already in use – without the involvement of the application. This is of interest if

- the network characteristic change,
- bug fixes (either related to implementation mistakes or even small protocol design mistakes) need to be deployed,
- or policies change (e.g., in the area of privacy).

The first case is illustrated with Figure 1.3 where an application sends data that needs to be transmitted reliably. The initial protocol stack does not offer a reliability protocol, since the network is already reliable by itself. This might be the case if two devices transmit data at a low rate over a short distance over a wireless medium. However, if the two devices have a further distance, the network gets unreliable and therefore, a reliability protocol is required.

The second case is illustrated with Figure 1.4 where a bug fix need to be deployed in a system that requires high availability. The buggy protocol can simply be replaced without the need to restart the application, or even the whole compute node.
1.3 Contributions

With this dissertation we provide a holistic view of flexible network architectures. We start with the evaluation of an existing architecture and show that it does not completely cover all requirements. Based on this insight we develop our own architecture and show how such an architecture can be implemented. We do not content ourselves with a typical software-only implementation, but we show how flexible network architectures can benefit from hardware acceleration.

Specifically, we make the following contributions:

- Our work within the Autonomic Network Architecture (ANA) project allowed us to critically assess the ANA architecture. This revealed that while the API was thoroughly thought out, and the adaptation of the protocol stack can easily be done on the local node, a mechanism to change the protocol stack between
several nodes was missing. Furthermore, it is not straightforward to include such a mechanism in the architecture.

- Based on those insights we have developed a truly dynamic network architecture called **Dynamic Protocol Stacks (DPS)**. This architecture allows the adaptation of the protocol stack during communication on the local host and between different communication partners.

- We have evaluated execution environments on which flexible network architectures could be implemented, and we have shown how flexible network architectures can benefit from **hardware acceleration**, even when the functionality to be executed is not known at the design time of the networking device. To this end we built EmbedNet, a system on chip execution environment for the DPS architecture.

- We have experimentally shown that **flexible network architectures might perform better than static architectures**. We conducted experiments in which the actual functionality of the protocol stack was continuously optimized to the network environment, as well as experiments in which the hardware/software mapping of network functionality was dynamically adapted to the current network traffic.

- Finally, we looked into the new research area of self-awareness and investigated how **flexible network architectures can benefit from self-awareness**. To this end we show that the DPS architecture fits well in the self-awareness framework and that network traffic patterns can be learned autonomously.

### 1.4 Scope

For this dissertation, we defined the scope as follows:

- The DPS architecture allows for the free adaptation of network functionality on top of the data link layer (layer two in the OSI reference model [41]). It assumes that the lower layers (physical layer and data link layer) are given. Additionally, it is assumed that those protocols are ether Ethernet or Wireless LAN, and
that they are implemented by the underlaying operating system. If desired, the DPS architecture could be extended to also allow for the adaptation of those lower layer protocols.

- The DPS architecture and execution environment provide a framework in which novel protocols as well as protocol stacks can be easily tested. It is out of scope of this dissertation, to develop novel protocols or protocol stacks. Rather, we made the implementation freely available and thus other researchers can use it to evaluate their ideas.

- The DPS architecture and execution environment enable the dynamic adaptation of network functionality and even the dynamic mapping of this functionality to either software or hardware. However, we do not provide any sophisticated adaptation algorithms that would use machine learning techniques to find the optimal protocol stack, or the optimal hardware / software mapping. In order to include such sophisticated algorithms that might be developed in the future, the DPS architecture and execution environment provide dedicated hooks.

1.5 Thesis Structure

The remainder of this dissertation is structured in five parts: architecture, execution environments, case studies, advanced optimization algorithms, and conclusions. More specifically, the work is structured as follows:

Part I: Architecture
In Chapter 2 we discuss the Autonomic Network Architecture whereas in Chapter 3 we present the Dynamic Protocol Stack architecture. In both chapters we first describe the basic concepts and the internal structure of a networking node implementing the given architecture. Then we discuss how protocol stacks are setup and how they can be adapted. In Chapter 4 we compare ANA with the DPS architecture and we show in which areas the DPS architecture goes further than ANA.

Part II: Execution Environment
We introduce Part II with an introduction on execution environments for packet processing in Chapter 5. In Chapter 6 and 7 we present the two execution environments for the DPS architecture we developed during this dissertation. In both cases we chose the Linux operating system as the underlying operating system, but we changed the actual packet processing hardware. In Chapter 6 we present how the DPS architecture can be implemented on a general-purpose CPU and in Chapter 7 we present our EmbedNet architecture, which is implemented as a system on chip solution on an FPGA. We conclude both chapters with a brief performance evaluation.

Part III: Case Studies
In this part we look at two case studies. First, we present how we can adapt the protocol stack functionality to minimize protocol overhead while keeping a given quality of service in Chapter 8. In Chapter 9 we show how we use dynamic hardware / software mapping of network functionality in order to improve the packet processing throughput on a network node.

Part IV: Advanced Optimization
In this part we discuss a few other aspects that should be considered, when advanced optimization algorithms are developed. Since it is unfeasible to perform larger scale case studies on the actual EmbedNet platform due to implementation overhead, we show how the EmbedNet platform can be abstracted and used in simulations in Chapter 11. In Chapter 12 we discuss adaptation strategies in general and we present one algorithm with which repeated traffic patterns can be learned. Finally, in Chapter 13 we look into the new research area of self-awareness and we show how the DPS architecture fits within this research topic and how our research could benefit from the insights that will be obtained from research in the area of self-awareness.

Part V: Conclusions
With this part we conclude this dissertation. We first set our work in context with related work in Chapter 13. In Chapter 14 we review our contribution and we list our papers that describe individual aspects of our work. In Chapter 15 we provide a critical assessment of the chosen research area and finally, in Chapter 16 we present the most interesting topics that could be investigated in the future.
Part I

Flexible Network Architectures

In this part we first present the Autonomic Network Architecture ANA, and then the Dynamic Protocol Stack (DPS) architecture which is derived from ANA. Finally, we compare the two architectures and show in which areas the DPS architecture goes further than ANA.
2 The Autonomic Network Architecture

The autonomic network architecture was developed within a European Union research project. The goal of the project, as stated on the web page [42], is as follows: *The ultimate goal is to design and develop a novel autonomic network architecture that enables flexible, dynamic, and fully autonomous formation of network nodes as well as whole networks.* To achieve this goal, a novel network node architecture as well as a set of communication primitives for setting up protocol stacks were developed.

2.1 Node Architecture

A single node within ANA consists of two parts, i) the MINMEX and ii) the Playground (see Figure 2.1). The actual network functionality is implemented as *functional blocks* (FB) within the playground, whereas the MINMEX (Minimal INfrastructure for Maximum EXtensibility) is responsible for forwarding the packets correctly between the functional blocks. This introduces a layer of indirection which is one of the key principles within ANA. It allows for building dynamic protocol stacks, in the sense that a node processing a packet can change the protocol stack without any input from the application. This is in contrast to the Internet architecture where the protocol stack is static. In the transmission direction, it is defined by the socket chosen by the application, and in the reception direction it is defined by the specified next header field in each packet.
In the following, we describe the four most important building blocks of ANA:

- **Functional Blocks (FB)**: While the Internet architecture divides network functionality into layers and protocols, ANA introduces the concept of *functional blocks (FB)*. In the simplest case, a functional block corresponds to a protocol, however, a functional block can also correspond to a monolithic network stack, or a small entity like the computation of a checksum.

- **Information Dispatch Points (IDPs)**: While in the Internet architecture the protocol stack as a whole can be identified by a socket descriptor, in ANA, each functional block can be identified by an *information dispatch point (IDP)*. Similar to socket descriptors, IDPs are used to send network packets between the functional blocks.

- **MINMEX**: The *MINMEX* is a controller that forwards the packets between the different functional blocks. It requires a dispatch table, that stores the relation from IDPs to functional blocks, and a key-val repository that is used for setting up protocol stacks (refer to Section 2.2).

- **Compartment**: A *compartment* is a set of FBs and IDPs with some commonly agreed set of communication principles, protocols
and policies. Typical network compartments are an Ethernet segment, the public IPv4 Internet, a private IPv4 subnet, the Domain Name System (DNS), or peer-to-peer systems like Skype.

ANA has a special compartment called the node compartment. The node compartment encompasses all FBs and IDPs within a node.

Based on those four components, a flexible network architecture can be built. Unlike traditional networking, where applications communicate with the network stack over BSD sockets, ANA provides its own API. This API is used for setting up the communication between applications and between different functional blocks and therewith for the construction of arbitrary network stacks. This API is described in the next section.

2.2 Protocol Stack Setup

In ANA, the process of setting up a protocol stack is started by an application that makes use of a set of communication primitives. The communication primitives are as follows:

- \texttt{IDP}_{s} \texttt{publish(IDP}_{c}, \texttt{CONTEXT, SERVICE)}
  A functional block uses the publish call to registers its functionality with other compartments (node or network compartments). The service argument specifies the functionality offered by the functional block. It is specified as a key word such as ”IPv4” or ”reliability”, or as an address such as ”10.0.0.1”.

- \texttt{int unpublish(IDP}_{c}, \texttt{IDP}_{s})
  Before a functional block is unloaded it has to unpublish itself, which leads to the removal of the associated service.

- \texttt{IDP}_{r} \texttt{resolve(IDP}_{c}, \texttt{CONTEXT, SERVICE)}
  A functional block that would like to make use of the service of another functional block, resolves the required functionality. Upon a successful resolve, it receives an IDP for the communication with the functional block providing the requested service.
• **int release**(IDP\(_c\), IDP\(_r\))
  
  If an IDP is not required anymore, it can be released. Upon releasing an IDP all associated resources are freed.

• **void* lookup**(IDP\(_c\), CONTEXT, SERVICE)
  
  The lookup primitive provides reachability information for a certain service. It is used for services such as DNS (Domain Name System), but not for a node internal protocol stack setup.

• **int send**(IDP\(_r\), DATA)
  
  A functional block can send packets to another functional block by using the send function.

The same primitives are used for setting up the protocol stack on the local node and for setting up the communication with the destination node. The primitives only have to be called with a different CONTEXT argument, either specifying the node compartment for the node local protocol stack setup, or a network compartment for setting up a communication between nodes.

Figure 2.2 shows the protocol stack setup corresponding to a traditional two-layer stack (FB1, Ethernet), where node A (on the left) wants to send data to node B (on the right). In this example, we assume that the FB1 block on node B is identified by Addr B, in a real scenario this could be an IP Address, port number, user name, etc. The setup of the protocol stack consists of the following steps:

1. The Ethernet functional blocks register themselves with the minmex.

2. The FB1 blocks on both nodes resolve the Ethernet block on their node via the minmex. As a result they receive the respective IDPs (e on node A and f on node B).

3. FB1 on node B publishes its address in the Ethernet block (using the IDP f). This generates the IDP i over which FB1 can receive data.

4. FB1 on node A performs a resolve request for the Addr B (using the IDP e).

5. The Ethernet block of node A sends a broadcast message over the network to resolve the requested address.
6. The Ethernet block of node B receives the message and verifies whether the requested address was published earlier. If so, it generates a new label that is mapped to IDP $i$. This label is sent back to the Ethernet block that initiated the request. This additional label is generated, since IDPs are considered to be private to the local node.

7. Upon receiving the reply, the Ethernet block of node A generates a new IDP $s$ and stores internally all details required to send data to the address $Addr$ B (e.g., MAC Address, label). It returns the resolve call to FB1 by specifying $s$ as the IDP corresponding to the resolved address.

8. FB1 can now use the send call with the IDP $s$ to send data to the FB1 with $Addr$ B.

![Figure 2.2: Protocol Stack Setup in ANA (one layer)](image)

Protocol stacks with more functional blocks are built recursively, using the steps described above for every layer. Similar to the two layer stack, the application starts with a resolve request in its lower layer compartment. This compartment in turn will issue a resolve request to the underlaying compartment and so on, until on every compartment a communication channel is established to the functional block on the other node.

In the next section, we look at how the protocol stack can be changed, after it is setup.
2.3 Node Local Protocol Stack Adaptation

The node local adaptation relies on the rebinding of IDPs to functional blocks. The Minmex holds a mapping of all IDPs to functional blocks in a table. If the protocol stack should be changed, the only thing that needs to be changed is this table. This process is shown in Figure 2.3. In this scenario FB1 sends data to IDP \( a \). In the upper picture IDP \( a \) is bound to FB2, whereas in the lower picture IDP \( a \) is bound to FB4. This rebinding happens transparently for FB1, which therefore continuously sends data and does not realize the change in the protocol stack.

(a) Functional block FB1 sends packets to IDP, a which is bound to functional block FB2.

(b) Functional block FB1 sends packets to IDP a, which was rebound to FB4.

Figure 2.3: Basic IDP operation.

However, this remapping applies to all functional blocks that send data to IDP \( a \), which might not always be appropriate. To this end, ANA introduces the concept of private and public IDPs which is illustrated in Figure 2.4. In the upper picture, two functional blocks

\[ \text{Figure 2.4: Private and Public IDPs.} \]
FB1 and FB2 send data to a public IDP $a^*$ which is bound to FB3. If the protocol stack for the data sent by FB2 should be changed, a new private IDP $a_{FB2}$ is created which is bound to a new functional block. With this distinction within the Minmex, both FB1 and FB2, can continue sending data to IDP a, while the protocol stack is only changed for data sent by FB2. Only the Minmex has to check whether for a given $<$sending functional block, destination IDP$>$ pair a private IDP exists and possibly forward it to the functional block bound to the private as opposed to the public IDP.

With this distinction between public and private IDPs, FB1 is still sending data through the old protocol stack, whereas FB2 sends data through the new protocol stack.

Figure 2.4: Advanced IDP operation.

Changing the protocol stack on a single node is valuable in the following situations:
• Adding monitoring blocks to the protocol stacks. This could include blocks that obtain simple statistics, but also blocks that analyse the packets for potential security threats, such as intrusion detection or intrusion prevention functional blocks.

• Deploying updates and bug fixes. As long as the underlaying functionality is not changed, a functional block can be replaced by another functional block during data transmission without synchronizing the change with a communication peer.

However, as soon as the functionality for the actual data transmission should be changed, both sides of the communication channel have to change the protocol stack simultaneously. In ANA, this situation was not studied and we explain in the next section why it is not straightforward to add it to the ANA architecture.

2.4 Inter Node Protocol Stack Adaptation

While a lot of thoughts were put into the node local protocol stack adaptation, no work was performed that looked into how the protocol stack could be adapted between nodes. This is also not a straightforward follow up work, due to the following reasons:

• A protocol would be required to renegotiate the protocol stack and trigger the adaptation on the remote node. It is not clear over which protocol stack this renegotiation would be performed.

• The IDPs are managed internally of the Minmex and no control interface is foreseen. Therefore it is unclear how the protocol stack could be adapted upon receiving a reconfiguration request.

• The nodes have to be able to distinguish between packets that were sent over the old protocol stack and over the new. Since each functional block includes the label of the destination functional block in the protocol header, the sending functional blocks would require to know the new label of the new functional block on the destination node. However, the idea of ANA is that the protocol stack can be changed, without that the sending nodes realize it.
Those drawbacks required us to continue the research in flexible network architectures and we developed a second flexible network architecture that can fulfill all basic concepts described in Chapter 1.2. This architecture is described in the next chapter.
3 The Dynamic Protocol Stack Architecture

The Dynamic Protocol Stack Architecture (DPS) was developed as part of the EPiCS [43] project that investigates proprioceptive computing systems, e.g., computing systems that monitor themselves and their environment and adapt their behaviour based on the observation.

For the DPS architecture, ANA was used as a starting position. With respect to ANA, the DPS architecture is simpler and addresses the shortcomings of ANA. The focus of the DPS architecture lies in providing a network architecture that includes a monitoring framework and that can easily adapt the protocol stack over several nodes. A comparison of DPS with ANA is provided in Chapter 4.

3.1 Node Architecture

The DPS architecture (depicted in Figure 3.1) roughly consists of two parts: a) the networking core that is responsible for processing packets and b) the monitoring framework that is responsible for adapting the protocol stack to the current network conditions. In the monitoring framework models are used by the configuration daemon to build network stacks based on sensor input.

In the following we describe each block in more detail.

• Models The models abstract the physical world in a way that a computer system can reason about it. In the DPS architecture, we foresee three different kind of models. Protocol models abstract the functionality provided by each functional block. Typical
models would state that a given functional block provides reliable communication, privacy, or delay tolerant communication. **Network models** abstract the physical communication characteristics. A typical example for a wireless communication is how the measured link quality results in packet loss or throughput. **Predictive models** try to foresee the future. Typical models are the expected battery lifetime or the expected periodicity in network traffic.

- **Sensors** The sensors provide up to date information of the network conditions and of the current status of the local node. Sensors can be either passive (just observing data) or active (inserting probes in the network and observe the reaction). Typical information provided by sensors is the current signal to noise ratio, available energy, measured latency or throughput. The data collected by the sensors is aggregated by the sensor daemon.

- **Configuration daemon** The configuration daemon determines the actual protocol stack based on the goals set by the user, the requirements specified by the applications and the models and

---

**Figure 3.1:** Overview of the node architecture.
sensor data. It consists of a strategy finder that decides which strategy to use (e.g., minimize power, maximize throughput, etc.) and the stack builder that determines the best stack and configures the networking core accordingly.

- **Networking Core** The networking core processes the packets. Similar to ANA, the network protocols are represented as functional blocks that can be combined individually and that are centrally managed. However, unlike in ANA, each protocol stack has its own instance of a specific functional block. The data processing function of each instance is identified with an information dispatch point (IDP). In order to let one instance know to which other instance it should forward the packets, a destination IDP can be configured. Similar to ANA, one functional block does not directly forward the packet to the next functional block, but the packets are forwarded through the core which does the mapping between IDP and the actual data processing functions. Figure 3.2 gives an overview over the interfaces to be provided by a functional block. When a functional block is loaded, it registers its functionality (fb_demo_factory) with the core (init_fb_demo_module). In this step, this is not the packet processing functionality, but its name (.type), the functionality it provides (.properties) and a constructor (.ctor) and destructor (.dtor) that will be used for the creation of the instances. If a new instance should be generated, the constructor (fb_dummy_ctor) allocates the required resources and registers the packet processing function (fb_dummy_netrx) and a function handling events (fb_dummy_event) with the core. The event function is typically used for setting the destination IDP of this instance.

### 3.2 Protocol Setup

The DPS architecture offers two possibilities to setup a protocol stack: manual and automatic. In the manual system, a user creates the instances of the functional blocks from the command line and connects them with each other as desired. This is especially interesting for situations where the protocol stack is not expected to change often, or
The Dynamic Protocol Stack Architecture

```c
/* process the packet /*and write the next IDP to the packet */

/* sets the IDPs */

/* allocate resources for this instance */ /*register the event and data handling function */ /*for this instance with the core */

/* free resources from this instance */

static struct fblock_factory fb_demo_factory = {
    .type = "ch.ethz.csg.demo",
    .ctor = fb_dummy_ctor,
    .dtor = fb_dummy_dtor,
    ...,
    .properties = {[0] = "demo_property"},
};
static int __init init_fb_demo_module(void) {
    // register the functional block with the core 
}
static void __exit cleanup_fb_demo(void) {
    // unregister the functional block with the core 
}
```

Figure 3.2: Interfaces for a functional block (as used in our Linux kernel implementation).

for debugging purposes. The command line interface offers the following functions:

- **add / remove**: Create an instance of a functional block or delete it.
- **set option**: Set an option for a functional block. The options provided depend on the functional block. E.g., an encryption functional block could allow for setting an encryption key.
- **bind / unbind**: Connect two functional block instances with each other.
3.2 Protocol Setup

- **replace**: Replace an instance of a functional block with another instance of another (or the same) functional block.

In the automatic system the protocol stack setup is done without human intervention. It is based on a set of properties, such as *reliability*, *privacy*, etc. that are used by functional blocks and applications. For functional blocks, properties correspond to the offered functionality, while for the applications the properties correspond to the requested functionality. Upon receiving a request for setting up a new connection, the configuration daemon will evaluate the requested characteristics and tries to build a corresponding protocol stack.

```
struct bind_msg{
    char name[FBNAMSIZEx];
    char app[FBNAMSIZE];
    char props[MAX_PROPS][20];
    int flags;
};
int main(void) {
    ...
    char buff[512];
    memset(buff, 0, sizeof(buff));
    sock = socket(PF_DPS, SOCK_RAW, 0);
    ...
    bmsg = (struct bind_msg *) buff;
    strcpy(bmsg->app, "chat");
    strcpy(bmsg->props[0], "reliability");
    strcpy(bmsg->props[1], "privacy");
    bmsg->flags = TYPE_CLIENT;
    ...
    ret = bind_config(bmsg);
    ...
    ret = sendto(sock, data, len, 0, NULL, 0);
    ...
    ret = close(sock);
    ...
    return 0;
}
```

**Figure 3.3**: Dynamic protocol stack architecture API example.

The current Application Programming Interface (API) uses simple key words both for protocols and requirements (see Figure 3.3 for an example). For the communication between the application and the
networking stack, a new BSD socket family was developed (PF_DPS) that offers the following system calls: `open`, `close`, `sendmsg`, `recvmsg`, `ioctl`. In addition to the socket calls, a library provides the `bind_config` function with which the application can specify the application to which it wants to send data as well as the properties the network connection needs to satisfy. The `bind_config` function sends the configuration message to the configuration daemon which finds all protocol stacks that match the requirements. Therefore, it uses the following input:

- The request from the application.
- The properties from the loaded functional blocks.
- Input from sensors and from the models to determine the current network characteristics.
- A layering constraints list. This list specifies constraints on the ordering of functional blocks. An example of a constraints list is shown in Figure 3.4. This example constraint file specifies that the `pf_dps` functional block has to be on top of any other functional block. And that the `iir` and `aes` block have to be on top of the `eth` block. The constraint file can either be human generated or, in a more advanced system, it could be learned. Initial ideas that could lead to such a learning algorithm are presented in Part IV of this dissertation. However, it is outside the scope of this dissertation to develop such algorithms. Therefore, we used a human generated constraint file for our experiments.

```plaintext
d :: ch.ethz.csg.pf_dps;
e :: ch.ethz.csg.eth;
i :: ch.ethz.csg.iir;
a :: ch.ethz.csg.aes;

d ^ {e i a};
i ^ {e};
a ^ {e};
```

Figure 3.4: Example layering constraints file.
Once all possible stacks are known, a connection to the destination node has to be established. The destination node might not have the required protocols available; therefore, before the communication starts, a protocol stack negotiation phase is executed (shown in Figure 3.5). First, all possible protocol stacks are sent to the destination node. The destination node decides which protocol stack to use depending on internal optimization criteria, sets up this protocol stack and sends the chosen configuration back to the source. If the source never receives a reply from the destination, which could happen on a lossy link, the source re-sends the configuration message and waits for the confirmation. After the completion of the negotiation phase, the actual data transmission starts.

There is a “chicken and egg problem” when it comes to the selection of the protocol that is initially used in the protocol negotiation phase. Since the communication partners did not yet negotiate a communication protocol they cannot communicate with each other, but they need a protocol for the exchange of the negotiation messages. To solve this problem, we assume that all nodes in a given network segment use the Ethernet protocol for the initial communication.

Similarly, if a connection to a node in another segment should be established, the intermediate nodes have to use the same internetworking protocol. In this case, in a first step, a network stack between the nodes on the path between source and destination would be established, and in a second step, the negotiation between the source and the destination node would be performed on top of the negotiated internetworking protocol.

In order to distinguish between data messages and control messages (for the stack negotiation) a one byte header is introduced.

Upon receiving a data packet, a node has to decide how to process it. In the Internet architecture this decision is based on next header fields that are part of each protocol header. For example, in the next header field of the Ethernet protocol it is specified whether the next protocol is IPv4, IPv6, ARP, etc. If the protocol stack is negotiated upfront, this step by step resolution of the next protocol is not necessary, instead, a single identifier per connection can be used. This identifier is calculated by the stack builder as follows: Every functional block has a unique name. In order to obtain a unique name the inverted url that is associated with the developer is used. This url is extended
with the name of the functional block, and if applicable with a version number. This is similar to the convention for package names in the Java programming language. The unique identifier for the overall protocol stack is then obtained by concatenating the individual names and hashing them. If the identical protocol is implemented by several developers, and their implementations pass an interoperability test, a special interoperability name should be used.

Upon packet reception, the Ethernet functional block checks the hash and forwards the packet to the corresponding stack “pipeline” (see Figure 3.6).

### 3.2.1 Packet Format

The packet format of a normal data-packet is shown in Figure 3.7. It shows a packet that is sent over the Ethernet protocol. Therefore, it starts with the Ethernet header, consisting of the destination MAC address, the source MAC address and the EtherType. As EtherType we chose 0xABBA, which is currently not assigned by the Internet Assigned Numbers Authority (IANA) [44]. This allows us to run the DPS architecture in parallel to the normal Internet architecture. After the
3.2 Protocol Setup

Ethernet header follow the DPS architecture specific headers. It starts with the eight byte long hash that identifies the protocol stack, followed by higher layer headers. Between the headers and the payload is a one byte control header, identifying whether this packet is a data packet or a control packet. Data packets are identified with 0x1 and control packets with 0x2.

![Ethernet Frame](image)

**Figure 3.7:** Packet format for a data message over Ethernet

Figure 3.8 shows the packet format of a control packet that is used to setup or change the protocol stack. Similar to the data packet, it starts with the Ethernet header, the hash value and the higher layer headers.
However, then it is identified as a control packet. The actual control header consists of a two byte sequence number, a two byte acknowledge number, and a one byte type field, identifying whether the packet is a request for changing the protocol stack \( \text{SUGG} = 0x1 \) or the confirmation that the protocol stack should be changed \( \text{COMP} = 0x2 \). In a request packet, the payload consists of an array of the possible protocol stacks, where each functional block is specified with its unique name. In the acknowledge packet (see Figure 3.9), the payload consists of one byte identifying the protocol stack to use by indicating the index in the array with the offered protocol stacks.

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
\text{CTRL} & \text{seq} & \text{ack} & \text{SUGG} \\
\text{has} & \text{possible stacks} \\
\text{high} & \text{(variable length)} \\
\text{soure mac} & \text{EtherType} \\
\text{dest} & \text{ination mac} \\
\end{array}
\]

**Figure 3.8:** Packet format for a control packet that is sent over the Ethernet protocol

In our implementation, the hash is calculated as follows: The unique names of the functional blocks that compose the protocol stacks are concatenated by a hyphen (-). This is followed by two colons and the application name. Out of this string, the SHA-1 hash is computed and the first eight bytes are used as the hash value in the DPS packets. However, any other procedure could be used, as long as it uniquely identifies the protocol stacks.
### 3.3 Node Local Protocol Stack Adaptation

As for setting up the protocol stack, the adaptation can either be done manually from the command line, or automatically triggered by a protocol stack optimization algorithm. In both cases the procedure is the same. Since for each communication channel an individual protocol stack with separate instances of functional blocks is built, changing the protocol stack on the local node is simple. It only requires to change the destination IDP of a functional block. This is a functionality provided by the framework and does not have an influence on the packet processing of a functional block. Additionally, functional blocks can export and import internal status information. This is required when the new functional block requires the internal state of the old functional block in order to work properly. A typical example would be to upgrade a functional block with a new version of the same functional block without the need for bringing down the connection. This situation is shown in Figure 3.10. The exchange of the status information requires the networking core to stop sending packets to the involved functional blocks for a short period of time so that the data transfer happens atomically with respect to packet processing.

---

**Figure 3.9:** Packet format for a control ack packet that is sent over the Ethernet protocol

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>+-----------------------------------------------+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>destination mac</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>source mac</td>
<td>EtherType</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>hash</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>higher layer headers</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>(variable length)</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CTRL</td>
<td>sequence</td>
<td>ack</td>
<td>COMP</td>
<td>select</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-----------------------------------------------+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.4 Inter Node Protocol Stack Adaptation

Including new functionality in the protocol stack usually requires changing the protocol stack on both communication end points. In the DPS architecture, negotiating a change in the protocol stack is done just like negotiating a new stack, except that renegotiation is executed over the currently used protocol stack. During the adaptation of the protocol stack, packets might be reordered on their way from source to destination. Therefore, special care has to be taken that packets still belonging to the old stack are not processed by the new stack and vice versa. Since the hash that identifies a given stack will change when the protocol stack is changed, also the packets sent over the new stack will
be identified with a different hash. This hash is used to dispatch the packet either to the new or the old protocol stack.

Figure 3.11: Updating the dynamic protocol stack over time.

Figure 3.11 shows what happens during a protocol stack change. First, a source node sends raw packets to a destination node. This 'empty' protocol stack is identified with hash1. Then, the source nodes wants to re-negotiate the protocol stack by sending possible changes to the destination node. The destination node selects a protocol stack, which includes the functional block FB1. The destination node sets up the new protocol stack, which is identified with hash2, and informs the source node about its selection. Then, the source node also configures the new protocol stack and starts to send packets using the new protocol stack. For some transition time, the destination node supports both protocol stacks. The destination node can therefore correctly process packet 2, although it arrives later than the first packet that uses the new protocol stack.
4 Comparison of ANA and the DPS Architecture

In this section we compare the ANA architecture with the DPS architecture.

• Focus
  – **ANA**: The focus for ANA was in providing the concepts of an autonomic network architecture and in developing the machinery for the node local protocol stack adaptation. However, the project ended before concepts were developed to adapt the protocol stack between several nodes.
  – **DPS**: The focus of the DPS architecture lies in providing an architecture for inter node protocol stack adaptation as well as including a monitoring system that can trigger the adaptation right into the architecture.

• API
  – **ANA**: The API introduced by ANA is used for applications and functional blocks, as well as for setting up the protocol stack on a local node and between nodes. On the one hand this makes it easy to learn the API, on the other hand several tasks could be done simpler with a dedicated API. Additionally, it is unclear how this API could be used to adapt the protocol stack on the fly.
- **DPS**: The DPS architecture offers a dedicated API for each functionality, which allows for using well known interfaces, such as for example the BSD socket interface.

- **IDPs**
  - **ANA**: In ANA, the IDPs are generated upon a resolve request. Once an IDP is resolved, it cannot be changed anymore. For changing the protocol stack, the Mimex internally changes the binding of the IDP to functional blocks. However, this requires the introduction of public and private IDPs, in case two functional blocks send data to one IDP, but only the protocol stack of one functional block should be changed.
  - **DPS**: In the DPS architecture, the IDPs are generated when an instance of a functional block is created. The IDPs to which a functional block should forward the data is written into the functional block from the core through a dedicated interface. This allows to easily change the IDP and therewith the protocol stack. The change of the IDP happens without that the functional block itself has to do anything actively, and therefore we argue the requirement that a protocol stack change is transparent to the functional block is still fulfilled. Within the DPS architecture each IDP is only used by exactly one functional block and therefore no distinction between public and private IDPs has to be made.

- **Protocol stack setup**
  - **ANA**: In ANA, the protocol stack is setup recursively, sending messages between the nodes on every layer. Since the functional blocks are only identified by a key word, a protocol stack could be built that is not working (e.g., if two functional blocks use the same key word but are not interoperable).
  - **DPS**: In the DPS architecture, the protocol stack is setup with a dedicated configuration message, which reduces the number of packets required for setting up the protocol stack as compared to ANA. In addition to the keyword describing the functionality, each functional block is identified by a unique name. This avoids building non-working protocol
stacks. If a node has several functional blocks that implement the same key word, it sends a protocol stack request to the destination node that allows the destination node to chose a protocol stack.

- **Inter node protocol stack adaptation**
  - **ANA:** Within the ANA project, no solution for adapting the protocol stack between different nodes was developed.
  - **DPS:** Within the DPS architecture, a protocol stack reconfiguration message is sent to the destination node. This reconfiguration message is always sent over the currently used protocol stack. If packets sent over the old and the new protocol stacks are re-ordered during transmission, the destination node is able to correctly forward the packets to either the old or the new protocol stack. This approach works fine for protocols that are packet oriented, however, if stream oriented protocols are used, unintentional side effects might appear. This situation should be analyzed in future work.

- **Implementation**
  - **ANA:** The ANA implementation was designed to be as flexible as possible, offering the possibility to implement functional blocks in user space or in kernel space. In order to isolate the functional blocks from each other, packets are passed by copy and not by reference. Those design choices lead to an implementation that is far from being competitive with implementations of the Internet architecture.
  - **DPS:** The focus on the DPS architecture was to provide an architecture whose implementation offers the same performance as an implementation of the Internet architecture. We present two possible implementations of the DPS architecture in the next part of this dissertation.
To summarize, in both architectures the protocols used for communication are negotiated when an application initiates a communication with another node. This allows for the coexistence of several network architectures in parallel. For each application and network condition, the best suitable architecture can be chosen, without the need that the application knows the available network architectures.

The mechanism for selecting the protocols differs between ANA and the DPS architecture: While in ANA a publish/subscribe mechanism is used, in the DPS architecture a dedicated negotiation protocol is used. This has several benefits: (i) the number of control packets can be minimized, since all protocols can be negotiated in one step and the recursive negotiation used by ANA can be omitted, (ii) the initiating node can specify a set of accepted protocols from which the destination node can select, and (iii) the negotiation protocol can also be used to change the used protocols during communication. Especially the last point is important when the set of involved protocols should be adapted to time-varying network conditions, which is one of the goals stated in the introduction.
Part II

Execution Environments for Dynamic Protocol Stacks

In this part we look at execution environments on which the DPS architecture can be implemented. We first provide an overview of the available options and then we present an implementation of the DPS architecture on two platforms. First, on a general-purpose CPU, and second on an FPGA-based system-on-chip.
Network packets can be processed by a variety of execution environments, ranging from general-purpose CPUs to dedicated ASICs. In this chapter we introduce the various possibilities.

- **General-Purpose Processor** The most basic execution environment is a general-purpose processor. In addition of executing the operating system and applications, the processor also processes the networking packets. The benefit of using a general-purpose processor for processing network packets is that no additional hardware is required. However, since the general-purpose processor is not optimized for network packet processing, its performance is not good enough to process packets at a high packet rate, or packets that are very time critical.

- **Ethernet Controllers** In order to transmit or receive packets from the network, an Ethernet controller on a networking card is required. The most basic cards just receive and transmit packets, but more advanced cards [45] offer CRC computation for IP, UDP and TCP packets, TCP segmentation or even support for the IPSec protocol. The benefit of those advanced cards is that they offer a better performance as compared to a general-purpose CPU for the functionality they advertise. Ethernet controllers are highly specialized devices, therefore, it is not possible for researcher or end user to program them to speed up new functionality.
• **Network Processors** Network processors are processors especially designed to process network packets. As compared to general-purpose processing units they do not include a floating point unit, but they offer dedicated mechanisms for operating on network packets. Many companies have developed network processors for networks with different characterisations. Hence, there is a huge diversity of network processor architectures [46–49]. Network processors offer a better performance than general-purpose processors, while keeping the flexibility of reprogramming them freely. The main application area of network processors are tasks that require header processing, but no payload processing. While most network processors would be a suitable platform for the implementation of the DPS architecture, we decided against it since the implementation would be very processor dependent, which would make it difficult to port the implementation to other processor families.

• **ASIC** ASICs (Application Specific Integrated Circuits) are hardware circuits that are designed to perform a given task with the best possible performance (in terms of throughput, latency or power). In the networking area, ASICs are parts of dedicated network processing devices such as routers or Intrusion Detection Systems (IDS) where payload processing is necessary. They are designed to perform their tasks at a very high rate and with as little power as possible. However, after fabrication it is no longer possible to change or adapt their functionality.

• **FPGA** Similar to ASICs, FPGAs (Field Programmable Gate Arrays) are hardware circuits that are optimized for a given task. Unlike ASICs however, FPGAs provide programmable logic cells. This means the hardware can be reprogrammed after fabrication. Commodity FPGAs require the whole FPGA to be reprogrammed at once, however, modern FPGAs allow for a partial reconfiguration at runtime. This has the benefit that the FPGA can still process data in some areas of the FPGA while other areas are reconfigured. This “area sharing” can be used to execute more code than would fit on a single FPGA or to include new code into the FPGA that was developed while the FPGA was running. Modern FPGAs provide a reconfigurable area that is big enough to fit complex logic. It is even possible to implement
a whole system on chip on an FPGA that consists of a general-purpose processor and several hardware accelerators. However, the flexibility of FPGAs comes at a price in that FPGAs have a lower performance as compared to ASICs.

For the implementation of flexible network architectures, only execution environments that provide the flexibility for reprogramming are suitable, which only applies to general-purpose processors, network processors, and FPGAs. We chose general-purpose processors and FPGAs as execution environments for the DPS architecture. In Chapter 7 we describe the implementation of the DPS architecture on a general-purpose processor and in Chapter 8 we describe the implementation of the DPS architecture as a system-on-chip implemented in an FPGA. The source code of those implementations is available in github at the following address: https://github.com/EPiCS/reconos/tree/v3.0_dev [50].
7 DPS on a General-Purpose Processor

In this chapter we describe how we implemented the DPS architecture on a general-purpose processor. We used the Linux operating system as a basis and therefore the implementation should run on any processor on which Linux is able to run.

7.1 Architecture

The basic idea of our implementation of the DPS architecture on a general-purpose processor is shown in Figure 7.1. The packet processing framework is implemented in the Linux kernel space, whereas the reconfiguration framework, consisting of the manual configuration interface and the autonomic configuration interface (with configuration daemon and sensor daemon) is implemented in the user space.

In the following, we first describe the packet processing framework and then the reconfiguration framework in more detail.

7.1.1 Packet Processing Framework

Applications in user space communicate over BSD sockets with the functional blocks in kernel space. We implemented a new BSD socket class \textit{PF\_DPS} which implements the functions discussed in Section 3.2. In kernel space, the packets are forwarded between the individual
functional blocks with the help of the packet processing engine (PPE). Therefore, the PPE maintains a mapping of Information Dispatch Points (IDPs) to the packet processing function of the individual functional blocks. There are two special functional blocks: The **PF_DPS** functional block and the **virtual link** functional block. Those blocks build the interface to either the application (**PF_DPS**) or to the device driver (**virtual link**). These two functional blocks directly call the packet processing engine if they have a packet that should be further processed. Additionally, if a normal functional block generates a packet (such as an acknowledgment packet) they can insert the packet in the PPE.

All functional blocks as well as the packet processing engine are implemented as Linux kernel modules. As described in Section 3.1, a protocol stack is built from instances of the functional blocks. In our implementation, upon inserting a functional block module into the Linux kernel, it registers its name, its properties, as well as a constructor for generating instances of the functional block with the PPE. Upon creating an instance of a functional block, the constructor
registers a function that will process data packets, as well as a function that processes configuration messages with the PPE. The configuration messages are used for setting the IDPs to which the instance will forward the data packets. Additionally, an interface to the /proc file system is registered. This interface can be used by user space tools to configure the functional block instance, for example for setting encryption keys.

### 7.1.2 Reconfiguration Framework

There are two possibilities to setup and reconfigure the protocol stack. In the first possibility, the manual configuration interface is used. This involves building the desired protocol stack manually on all communication nodes. Therefore, we implemented the functions described in Section 3.2. In the second possibility, the autonomic configuration interface is used. This system builds (and reconfigures) the protocol stack without human intervention, solely based on application requirements, monitoring input and internal models. The autonomic configuration interface consists of two parts: The sensor daemon which is responsible for collecting monitoring information, and the configuration daemon which initiates the protocol stack setup. In contrast to Figure 3.1, the implementation of the configuration daemon currently only consists of the stack builder, but not the strategy finder. The sensor daemon as well as the stack builder are shown in Figure 7.2.

The whole reconfiguration framework is implemented in user space and uses netlink sockets to communicate with the packet processing engine.

**Sensor Daemon**

The sensor daemon is a user space tool that collects and aggregates monitoring data. The sensor daemon itself does not perform the actual measurements, but it offers a plugin system. This allows the sensor daemon to be extended to support arbitrary measurement data, such as temperature, signal to noise ratio (SNR), latency, throughput, etc.

The sensor daemon queries the plugins for new measurement values at time intervals that are configurable per plugin. For each plugin, there exists an internal data base in which the measured values are stored.
Figure 7.2: Architecture of the sensor daemon and the stack builder.

The stack builder (or other tools) can register threshold values for each plugin with the sensor daemon and will be alerted when a measurement exceeds that threshold.

The plugins are implemented as shared object files. For the communication between the sensor daemon and its clients, Linux signals (to send the alerts), shared memory (to provide the measurements) and Unix domain sockets (to start a query or to register thresholds) are used.

Configuration Daemon

The automatic configuration of the protocol stack is done by the configuration daemon. As mentioned, currently only the stack builder is implemented. The stack builder has an interface to the sensor daemon and the application and the PPE. Internally, the stack builder consists of four parts:

- The *APP IPC server* implements the interface towards the application. It implements the counter part of the `bind_config` call described in Figure 3.3.
• The *functional block property fetcher and parser* obtains the descriptions of the properties of the functional blocks currently available on that node. Therefore, it uses an interface provided by the PPE.

• The *functional block dependency fetcher and parser* reads the constraints file that specifies the dependencies between the functional blocks. The format of this file is described in Figure 3.4.

• Finally, the *reconfiguration engine* combines the request from the application with the available protocols and the specified dependencies as well as with the data obtained from the sensor daemon. It performs the protocol stack setup on the local node and initiates the configuration of the destination node.

The timeline for setting up a protocol stack on the node initiating a communication is shown in Figure 7.3. An application sends its communication requirement to the stack builder over Unix sockets. In order to start the negotiation of the protocol stack, the stack builder initiates the building of a minimal protocol stack consisting of the Ethernet functional block and the fb_pf_dps functional block. It builds the appropriate hash value and configures the Ethernet block accordingly. The API for this communication is implemented with netlink sockets. After everything is setup, the stack builder builds the actual protocol setup request message for the destination node and sends this to the PPE, with the help of a device file. The PPE forwards the packet to the fb_pf_dps functional block. The fb_pf_dps functional block adds its header and forwards the packet to the Ethernet functional block which broadcasts the message through the physical interface.

Upon receiving a message, the Ethernet functional block of the source node checks the hash value and forwards the packet to the fb_pf_dps functional block. The fb_pf_dps functional block recognizes the packet as a control packet and forwards it to the PPE which in turn sends it to the stack builder (with a read operation on the device file). The stack builder evaluates the answer, builds the chosen protocol stack, and sets the appropriate hash value. Finally, it returns to the application which now can start sending data packets over the dps socket.
Please note, that while in Figure 7.3 there are direct arrows between the functional blocks, the actual path always goes via the PPE. The direct arrows are only used for better readability.
Figure 7.3: Timeline for setting up a protocol stack.
7.2 Performance Evaluation

We evaluated the maximum packet reception rate of the Dynamic Protocol Stack architecture on commodity hardware (Intel Core 2 Quad, 2.40GHz, 4GB RAM, Linux 3.0 amd_64, e1000e with a NAPI NIC driver and PCIe interconnect). Figure 7.4 compares the maximum packet reception rate of the normal Linux protocol stack with the packet reception rate of the Dynamic Protocol Stack architecture. We compared a) the packet reception rate if the kernel receives the packets and drops them, and b) the packet reception rate if a user space application receives the packets and drops them. The packet sizes where chosen to match the suggestions from RFC2544 [51], which specifies a benchmark methodology for networking devices. From Figure 7.4 we see that for long packets, the DPS architecture is able to process as many packets as the Linux kernel. For short packets the DPS architecture offers a slightly lower performance.

![Figure 7.4: Performance comparison between receiving packets by the Linux kernel, by the DPS architecture in the kernel, by a PF_Packet socket in user space, and by PF_DPS sockets in user space.](image)

Additionally, we evaluated the per functional block overhead by building a chain of one up to ten functional blocks. The maximum packet reception rate drops for minimum sized packets as the number
of functional blocks increases. However, for packets longer than 128 bytes the overhead is negligible (see Figure 7.5). Furthermore, we also assume that a “pipeline” of 10 or more functional blocks is used rarely. In our experience, actual stacks usually have no more than five chained functional blocks.

![Kernel space DPS versus Linux, Receive and Drop](image)

**Figure 7.5:** Packet reception rate for a functional block chain up to ten blocks.

We also measured the protocol stack reconfiguration time, i.e., the time it takes from an event that triggers a reconfiguration until data can be sent over the new protocol stack. This time is composed of (i) the time to determine and reconfigure the stack on both sides of the communication and (ii) the time to send the reconfiguration messages. We measured a protocol stack reconfiguration time of 806μs whereof 286μs were required for the transmission of the packets (round trip time).
In this chapter we look into the acceleration of individual functional blocks by using dedicated hardware components. Whereas in the Internet architecture all protocols are known, and therefore static hardware accelerators can be built, this is not the case with dynamic protocol stacks. With dynamic protocol stacks new protocols can be included in a system at any time. Therefore, we need a reconfigurable computing system that allows us to include arbitrary protocol stacks at any time. Our proposed system, EmbedNet, is based on Field Programmable Gate Arrays (FPGAs) that offer partial reconfiguration, i.e., the reconfiguration of a part of the FPGA while the rest of the FPGA remains fully operational.

By using FPGAs, we can not only include hardware acceleration for arbitrary functional blocks at run time, but we can also dynamically schedule functional blocks to be executed in hardware (faster) or in software (slower). Due to the limited FPGA area, we cannot simply map all functional blocks to the hardware, but we need to weigh the benefits of a hardware mapping of one set of functional blocks against the benefits of a hardware mapping of another set of functional blocks.

Figure 8.1 shows a high-level comparison between the Internet architecture implemented on a network interface card (NIC) and the EmbedNet system. NICs for the Internet architecture come in two varieties: the first only has the low level protocols in hardware whereas the second has a hardware TCP Offload Engine (TOE), which contains all protocols up to and including TCP. In the EmbedNet architecture, the system adapts the mapping of functional blocks to hardware...
or software at runtime—even when sending or receiving data. We use this adaptability to continuously optimize the packet processing performance to the current workload.

![Diagram of NIC with dynamic hardware](image)

**Figure 8.1:** NIC with dynamic hardware.

### 8.1 EmbedNet Architecture

The complete EmbedNet execution environment is shown in Figure 8.2. It consists of two parts: a) the packet processing framework for the actual data processing, and b) the reconfiguration framework that decides which functional block should be executed in hardware, and that performs the reconfiguration of the FPGA.

#### 8.1.1 Packet Processing Framework

The packet processing framework consists of a software part and a hardware part. The software part corresponds to the architecture for dynamic protocol stacks on general-purpose processors. The hardware parts consists of a set of functional block placeholders, in which arbitrary functional blocks can be configured. All functional blocks are connected by a hardware interconnect that uses an individual address for each functional block placeholder that is connected to it. Three functional blocks are statically configured: two for the communication between hardware and software (h2s, s2h) and the Ethernet block that implements the interface to the physical network interface.

If a packet is sent from an application through a PF_DPS socket, or, from a functional block in software, it is inserted in the packet
processing engine, which forwards the packets to the correct functional block, no matter whether the block is currently mapped to software or hardware. Therefore, the packet processing engine maintains a mapping of IDPs to the receive function of the corresponding functional block. If the functional block is currently executed in software, a pointer to its function is used, if the functional block is executed in hardware, the current hardware address of the functional block is used.

If a packet is sent from a functional block in hardware no centralized IDP to address lookup mechanism can be used for performance reasons.

**Figure 8.2:** Overall system allowing for a self-aware hardware/software mapping.
Therefore, each hardware functional block has a dedicated interface over which the mapping of IDPs to addresses can be set.

When adapting the packet flow due to either a new protocol stack or a new hardware / software mapping, conceptually, only the mapping of the IDP to the actual execution environment needs to be changed. Practically, the procedure is somewhat more complicated, since the FPGA also needs to be reconfigured with the new functional block. This procedure is described in Section 8.1.3.

8.1.2 Reconfiguration Framework

There are two possibilities to change the mapping of functional blocks to either hardware or software: manual configuration and autonomic configuration.

For the manual configuration, we have extended the command line interface presented in Section 3.2 with new commands:

- **flag / unflag trans**: Configures the packet processing framework to queue all packets that are addressed to a given functional block.

- **flag / unflag hw**: Configures the packet processing framework to send all packets addressed to the given functional block to hardware.

The hardware address of a functional block could be configured with the **set option** command discussed in Section 3.2. However, this is not yet implemented, since we did only examples with one reconfigurable hardware region. In addition to the configuration of the packet processing framework, the actual configuration of the FPGA also needs to be changed manually. This is done by manually sending the partial FPGA configuration file to the device driver over a device file (/dev/icap0).

The autonomic reconfiguration framework consists of

- a **scheduler** that determines the hardware / software mapping for the next time slot. In addition to packet processing statistics, it could also make use of other statistical data such as CPU utilization, expected battery life time, etc.
8.1 EmbedNet Architecture

- a **reconfiguration daemon** that waits for a new configuration from the scheduler and initiates the partial reconfiguration of the FPGA. It selects the appropriate FPGA configuration file and forwards it to the reconfiguration infrastructure.

- the **reconfiguration infrastructure** that does the actual reconfiguration of the FPGA.

Architecturally, it does not matter where the individual parts are implemented, as long as the scheduler has an interface on which it can collect statistical data and the reconfiguration daemon has access to the file system in order to transfer the partial configuration files to the reconfiguration infrastructure. The reconfiguration infrastructure itself can also be implemented in several ways, but all implementations have in common that they use the internal reconfiguration port of the FPGA in order to perform the reconfiguration.

8.1.3 Changing the Hardware/Software Mapping

For the explanation of the required steps to adapt the hardware/software mapping of a protocol stack, let us consider the following scenario: There are two protocol stacks $\text{eth} \rightarrow B \rightarrow C$ and $\text{eth} \rightarrow D \rightarrow E$ from which currently $\text{eth}$ and $B$ are mapped to hardware and the rest is implemented in software. The scheduler decides that in the new mapping $D$ should be implemented in hardware and $B$ in software.

This results in the following steps, which are also described in Figure 8.3:

I. **Move B from hardware to software.**

The transition of a functional block from hardware to software does not take much time. A functional block that is currently used in a protocol stack is always present in software, regardless of whether it is currently processing packets or not. Therefore, the following three steps need to be done (compare steps 1 to 3 in Figure 8.3):

(a) **Stop forwarding packets to B.** Functional block $\text{eth}$ is configured to forward all packets to the software where they are buffered.
(b) **Transfer state.** The state is gathered from the hardware block and transferred to the software block.

(c) **Forward packets to $B$.** The core forwards the packets again to functional block $B$.

This completes the mapping procedure for functional block $B$ and the hardware module is not required anymore and can be reconfigured.

II. **Move $D$ from software to hardware.**

The transition of a functional block from software to hardware requires more time, since the FPGA needs to be reconfigured. However, this reconfiguration can be done, while the functional block is still processing packets in software. The following steps are required:

(a) **Partial reconfiguration.** Perform the partial reconfiguration of the hardware module with the bit file required for functional block $D$ (step 4).

(b) **Stop forwarding packets to $D$.** This involves buffering all packets which can either be done in software or in hardware (step 5).

(c) **Transfer state.** The state is gathered from the software module and transferred to the hardware module (step 5).

(d) **Forward packets to $D$.** Functional block `eth` is configured to forward the packets to the hardware module and the packet processing engine forwards the buffered packets to the hardware. In this step the packets might arrive out of order in the hardware module if they were buffered in software, however, if a hardware buffering is implemented the packet order is preserved (step 6).

This completes the re-mapping procedure. The functional blocks that are re-mapped only need to be stopped for a short period of time, where the state is transferred from the software to the hardware or vice versa. The functional blocks not involved in the re-mapping are running continuously.

When employing such a system it needs to be considered whether for the given situation it is tolerable to have a system that reorders packets
(e.g., because the packet order is not important or because the packets get ordered on a higher protocol layer that is always implemented in software) or, whether the local node is not allowed to re-order packets (e.g., because some functional blocks that will be executed in hardware require in-order packet processing). In the first case, the packets can be buffered in the software, whereas in the second case the packets need to be buffered in hardware.

Figure 8.3: Dynamic hardware / software mapping.
8.2 EmbedNet Hardware Design

8.2.1 Platform

The hardware design of our FPGA-based EmbedNet prototype is depicted in Figure 8.4. We use a reconfigurable System-on-Chip (rSoC) architecture that combines a soft-core MicroBlaze CPU [52] with several hardware modules and peripherals on a single device. The Linux operating system runs on the MicroBlaze CPU and the required file system is stored on a compact flash disk. The functional blocks can either be executed on the MicroBlaze or directly in the FPGA. In order to implement our architecture on an FPGA board, we assume that it has external SDRAM, a physical Ethernet interface and a compact flash disk reader.

![Figure 8.4: EmbedNet FPGA design.](image)

There are three modules that are always present in the FPGA. One is the Ethernet functional block (ETH) that interfaces with the physical interface (PHY), and two modules which are responsible for transmitting the packets over the hardware/software boundary (called H2S and S2H, respectively). In addition to these statically configured modules, there are also dynamic modules (PR). Those modules can be reconfigured at run time with the functionality of arbitrary functional blocks.

Functional blocks are connected by a Network on Chip (NoC) that forwards packets between them and also supports pipelined packet processing. The NoC consists of switches in a ring topology, where
each switch connects to a configurable number of functional blocks. The total number of modules is a design-time parameter; this allows for the throughput of the NoC to scale appropriately by increasing the bandwidth between the switches and by allowing for more hardware modules to be connected to one switch. Run-time reconfiguration of the modules is done with the help of the Xilinx core XPS HWICAP [53]. The configuration files for the partial reconfiguration are stored on an external flash card.

While a given area on the FPGA is reconfigured, it may emit spurious signals. In order to prevent packet processing errors during reconfiguration, we added a dedicated enabling block between the NoC and each dynamic module. During reconfiguration, this block sets the signals that go into the NoC to a known value.

To aid implementation of a functional block in hardware and software, we provide wrappers: in hardware, this is a VHDL entity and in software, this is a Linux kernel module. The wrapper consists of the code required for receiving and sending packets and configuration data as well as transferring internal state between a hardware and a software module.

Since there is no automatic translation from a functional block in software to one in hardware, it is the responsibility of the functional block’s author to make sure that the respective implementations are equivalent, and also to provide the state that is required when resuming a hardware block after a re-mapping.

8.2.2 Reconfiguration Framework

The implementation of the reconfiguration framework is shown in Figure 8.5. The scheduler is tightly integrated with the packet processing engine, this gives a simple interface for accessing statistical and configuration data. However, it does not allow the scheduler to directly send new configuration files to the reconfiguration framework, since it cannot access the file system on which the FPGA configuration files are stored. Therefore, a user space reconfiguration daemon waits for commands from the scheduler. The reconfiguration daemon reads the corresponding configuration file from the file system and writes it to the icap device driver in the Linux kernel. The device driver forwards the data to the xilinx hw_icap core in the FPGA.
8.2.3 Network on Chip (NoC)

The communication between the functional blocks in hardware is organized by a Network on Chip (NoC). The NoC is built from switches that are organized in a ring topology. The overall integration of the NoC in the system is shown in Figure 8.4. In Figure 8.6 it is shown how two functional blocks are connected to a switch.

The packets that are transmitted by the NoC have an additional header with the following format (also shown in Figure 8.7):

- **Address** The address of the hardware block to which the packet should be sent. The address is split into two parts: 4 bits for the identification of the switch and 2 bits for the identification of the port on the switch.
- **Priority** A 2 bit field that specifies the priority of this packet. The switches select the packet with the highest priority waiting on any of the ports to be processed.
- **Direction** One bit that specifies whether the packet is processed in INGRESS or EGRESS direction (whether a packet is being received or sent).
- **Delay sensitive** One bit that specifies whether this packet is delay sensitive. This can be used by functional blocks that aggregate packets before they are further processed.

**Figure 8.5:** Implementation of the reconfiguration framework.
8.2 EmbedNet Hardware Design

Figure 8.6: Two functional blocks connected to a switch in the NoC.

- **Destination IDP** Four bytes, identifying the destination IDP of a packet. This can be used by functional blocks to determine the next address to which the packet should be sent.
- **Reserved** Four bytes that are reserved for future use (new flags, or new routing concepts).

The interface between the switches and the functional blocks is implemented with the xilinx local link interface [54] that was extended to conveniently set the NoC header. The local link interface consist of a handshaking algorithm that is implemented by the signals shown in Table 8.1.

### 8.2.4 Hardware Functional Blocks

Each functional block implemented in hardware has at least two different interfaces, a data processing interface and a control interface. The data processing interface is connected to the NoC and is described in Table 8.1. The control interface is connected to the software and
allows the software to configure the hardware functional block, read internal state, or read statistical data. This interface is implemented with the help of ReconOS [55] which is described later in this section.

There are three functional blocks that have additional interfaces. The Ethernet functional block has an interface to the PHY mounted on the evaluation board. This interface is a serial gigabit media independent interface (SGMII) which is provided as a Xilinx IP Core [56] in order to send and receive data from the network. The two functional blocks h2s and s2h are responsible for transmitting the packets from hardware to software and vice versa. Therefore they use shared memory in combination with message boxes, both provided by ReconOS and described later in this section. In the software, the packets are processed by a kernel module which interfaces with the part of the protocol stack that is implemented in software. In the current implementation each packet is transferred separately over the hardware/software boundary, however, other implementations (e.g., using a ring buffer) could be done.
Table 8.1: Xilinx local link interface for sending packets.

<table>
<thead>
<tr>
<th>signal</th>
<th>direction</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Out</td>
<td>The data to be transmitted</td>
</tr>
<tr>
<td>sof</td>
<td>Out</td>
<td>Signals the start of a packet</td>
</tr>
<tr>
<td>eof</td>
<td>Out</td>
<td>Signals the end of a packet</td>
</tr>
<tr>
<td>src_rdy</td>
<td>Out</td>
<td>Signals that the block is ready to send data (and therefore the value on the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data signal is valid)</td>
</tr>
<tr>
<td>dst_rdy</td>
<td>In</td>
<td>Signals the destination is ready to receive packets in the current cycle (a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nd therefore the source can update the value on the data signal)</td>
</tr>
</tbody>
</table>

8.2.5 Hardware / Software Interface

We make use of ReconOS [55] for implementing the hardware / software interface. ReconOS is an operating system for reconfigurable hardware that is built as an extension to existing embedded operating systems such as Linux [57], eCos [58], or the Xilkernel [59]. ReconOS brings the multi-threaded programming model well established in software systems to systems with hardware and software modules. It abstracts hardware modules as hardware threads that can interact with each other and with software threads over inter process communication methods such as shared memory and synchronization primitives. In EmbedNet the following system calls provided by ReconOS are used:

- Software
  - `reconos_init`: Initializes the ReconOS environment.
  - `reconos_hwt_setresources`: Maps message boxes, semaphores etc. to a hardware thread.
  - `reconos_hwt_create`: Initializes a specific hardware thread.
  - `reconos_cache_flush`: Flushes the cache of the processor.

- Hardware
- `mbox_put`, `mbox_get`: Transfers a 32 bit value between two processes. `mbox_put` blocks when the message box is full and `mbox_get` blocks when the message box is empty. The configuration of the functional blocks is performed with those two functions.

- `memif_write`, `memif_read`: Writes (or reads) a configurable amount of data from the hardware module to the main memory. This is used by the h2s and s2h functional blocks to transfer the packets over the hardware/software boundary. After the hardware completes the write call, it lets the software know that new data arrived. This can be done by a call to `mbox_put`.

The architecture of the ReconOS/Linux variant is shown in Figure 8.8. Each hardware thread is represented in the software with a delegate thread. This delegate thread is responsible to execute the operating system calls (such as semaphores and message boxes) on behalf of the hardware threads. The communication between the delegate thread and the hardware thread is implemented with processors streaming links (FSLs) and interrupts. For transferring data between threads shared memory is used. Since Linux uses a Memory Management Unit (MMU) for accessing the memory, this unit has to be mapped to the hardware as well. For the synchronization of the hardware MMU unit with the MMU unit of the Linux operating system, FSL links are used.

While the original ReconOS implementation provides transparent communication between Linux user space and the hardware, we have extended it to also support communication between Linux kernel space and the hardware. This extension is required because many software parts of our network architecture run in the kernel space for performance reasons. This extension required the following two steps:

- All ReconOS library functions were implemented in the Linux kernel. This includes functions for initializing and cleaning up the ReconOS environment as well as the individual hardware threads and the delegate threads that represent the hardware threads in the software.

- The ReconOS hardware MMU was extended to distinguish between user space and kernel space addresses. For kernel
8.3 Performance Evaluation

We implemented the EmbedNet architecture as a combination of our own VHDL code and readily available IP cores provided by Xilinx and ReconOS. We synthesized the code to run on a Xilinx Virtex-6 FPGA ML605 evaluation board. For the synthesis we used the Xilinx Platform Studio version 14.5 [60].

We implemented a design with a microblaze CPU, two switches, the h2s and s2h functional block, the Ethernet functional block, and one functional block that can be dynamically reconfigured. The area reserved for this functional block is big enough to implement the aes

Figure 8.8: ReconOS Architecture with control and data path.

space addresses (addresses that are higher than OxC0000000) it subtracts the constant offset OxC0000000, for user space addresses, the MMU performs a page table lookup.

8.3 Performance Evaluation
encryption algorithm [61]. The characteristics of the microblaze CPU are shown in Table 8.2. Please note that this CPU cannot be compared to a modern standalone processor which is highly optimized, clocked at an order of magnitude faster, and consists of several cores. Therefore, the evaluation shown in this section cannot be compared with the evaluation shown in Section 7.2.

**Table 8.2: CPU characteristics.**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU frequency</td>
<td>100 MHz</td>
</tr>
<tr>
<td>Data cache</td>
<td>16 kB</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>16 kB</td>
</tr>
</tbody>
</table>

The resources required for the complete system (including the microblaze CPU) are only a small fraction of the overall available resources as shown in Table 8.3. This shows that the implementation could easily be extended to more functional blocks with more complex functionality.

**Table 8.3: Design Summary, with AES block configured.**

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
<th>% (total = Virtex-6 LX240T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>25706</td>
<td>3%</td>
</tr>
<tr>
<td>LUT</td>
<td>34638</td>
<td>8%</td>
</tr>
<tr>
<td>Block Memory</td>
<td>47</td>
<td>11%</td>
</tr>
<tr>
<td>DSP48</td>
<td>4</td>
<td>1%</td>
</tr>
<tr>
<td>Clock Manager</td>
<td>1</td>
<td>8%</td>
</tr>
<tr>
<td>Tri-Mode Ethernet MAC</td>
<td>1</td>
<td>25%</td>
</tr>
<tr>
<td>Gigabit Transceiver</td>
<td>1</td>
<td>5%</td>
</tr>
<tr>
<td>Global Clock Buffer</td>
<td>10</td>
<td>31%</td>
</tr>
<tr>
<td>IO</td>
<td>129</td>
<td>22%</td>
</tr>
</tbody>
</table>
Figure 8.9 shows how the individual hardware blocks are distributed on the FPGA area. The purple area implements the microblaze CPU together with the memory management unit. The orange area is the network on chip and the green areas are functional blocks. By far the biggest functional block is the aes encryption functional block which is placed at the top left. In contrast to the other blocks, it is shaped as a rectangle. This is because the area of a partially reconfigurable FPGA region needs to be specified by an engineer before the tool starts the placement phase. All other modules are placed by the tool without any constraints.
Figure 8.9: *Routed FPGA Design.*
8.3.1 Maximum Sending Rate

We evaluated the maximum sending rate of network packets on the ml605 in four scenarios. The result is depicted in Figure 8.10.

1. **Linux Raw Sockets, default hw/sw interface** We used the Xilinx base system builder, which is part of the Xilinx Platform Studio, to build a normal Linux system and sent packets over raw sockets. This is the base line evaluation and shows that the maximum sending throughput is around 14.4 MBit/s.

2. **PF_DPS Sockets, default hw/sw interface** On the same system we sent packets with the PF_DPS socket family instead of raw sockets. This leads to a maximum sending throughput of 3 MBit/s which is considerably slower than the sending rate achieved by the raw sockets. The performance degradation is independent of the packet length. Therefore we assume it is a constant, per packet overhead. We attribute this overhead to the memory allocation scheme used in PF_DPS sockets which is not optimized.

3. **Kernel, EmbedNet hw/sw interface** In this scenario we sent packets directly from a Linux kernel module over the hardware/software interface and the NoC described in Section 8.2 to the Ethernet interface. This offers with 24 MBit/s the highest possible throughput of all scenarios. Interestingly, the number of packets the system can sent is nearly independent from the packet size. This can be attributed to the fact, that the hand shake mechanism used for transmitting the data between hardware and software is more expensive than actually transmitting the data. It also suggests that with an optimized hardware/software interface, where several packets could be transmitted at once, the overall throughput could be further increased.

4. **PF_DPS, EmbedNet hw/sw interface** This scenario is the full EmbedNet scenario where an application sends data through the PF_DPS sockets over the EmbedNet hardware. It has a maximum throughput of 14 MBit/s and has the same characteristics than the scenario above. Similar to the comparison of the scenario 1 with scenario 2, the comparison of scenario 3 with scenario 4 shows that it can process in average 1000 packet less per second.
This can be again attributed to the overhead introduced by the PF_DPS Sockets.

![Maximum Send Packet Rate](image)

**Figure 8.10: Maximum sending rate comparison.**

We also measured the maximum throughput of the hardware only system, by sending packets from an external node to EmbedNet where it was forwarded to a dummy FB (which does nothing) and back to the Ethernet FB. The maximum throughput is currently 0.8 Gbit/s which corresponds to the forwarding rate of the switches (8 bits at 100 MHz). The design could easily be adapted to clock the switches with 125 MHz which would allow for line rate (1Gbit/s) forwarding.

### 8.3.2 Reconfiguration Overhead

During the partial reconfiguration of the FPGA, required for changing the hardware/software mapping of functional blocks, the area that is reconfigured cannot process any packets. We measured the time it takes to reconfigure two typical FPGA area sizes on our system using
the Internal Configuration Access Port (ICAP). The reconfiguration times are shown in Table 8.4. The partial bitstreams are stored on the compact flash memory. Hence, a bitstream file either has to be loaded from the compact flash disk, or it is already cached in the RAM, in which case the reconfiguration is considerably faster.

The reconfiguration times shown in Table 8.4 are considerably higher than those shown in related work. This can be attributed to the overhead introduced by the Linux operating system. For instance, in [62] the authors neither use an operating system, nor are the bitstreams stored on an external memory. Additionally, they show that the HWICAP (hardware internal configuration access port) core provided by Xilinx can be optimized to perform an order of magnitude faster while not requiring CPU resources during the reconfiguration. Using their optimized core would also improve the reconfiguration time in our system.

<table>
<thead>
<tr>
<th>FPGA area</th>
<th>bitstream size</th>
<th>initial reconfiguration</th>
<th>subsequent reconfigurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>7%</td>
<td>0.65 MB</td>
<td>850 ms</td>
<td>110 ms</td>
</tr>
<tr>
<td>15%</td>
<td>1.3 MB</td>
<td>1600 ms</td>
<td>220 ms</td>
</tr>
</tbody>
</table>
In the second part of this dissertation we introduced two execution environments for the DPS architecture. The first execution environment is a software only system that is based on a standard Linux system and therefore, can run on all hardware that is supported by the Linux operating system. The second execution environment is implemented as a system-on-chip design on an FPGA. Here, the functional blocks can be executed in software (on a soft-core CPU), or directly in the FPGA.

We showed that the software only implementation has a similar performance than an implementation of the standard Internet architecture on the same hardware, which is important if the DPS architecture should be applied in real systems.

With our system-on-chip implementation we showed how a flexible network architecture can make use of hardware acceleration, even though the network functionality might not be known at the production time of the hardware. While the presented implementation focuses on a system-on-chip approach the general concept could also be applied to other hardware architectures that might have a dedicated CPU. Having a dedicated CPU would be interesting since it promises to offer better performance which would be required for a general acceptance of hardware / software co-designs for flexible network architectures.
Part III

Case Studies with Dynamic Adaptation

In this part we provide two case studies that show the benefits of a flexible network architecture. We first show how we can adapt the functionality provided by the protocol stack to minimize protocol overhead, and then we show how we can adapt the hardware / software mapping of network functionality to improve the system performance.
10 Autonomic Adaptation of the Protocol Stack Functionality

In order to show the usefulness of a flexible network architecture, we implemented a use case, in which the protocol stack is changed depending on the current network environment. Therefore, we developed a simple application that mimics a sensor that sends measurement data periodically to a server. We argue that transmitting a packet over a wireless interface costs energy, and therefore should only be performed when necessary. In this use case the protocol stack can be changed to include an idle repeat request (IRR) reliability protocol, if the sensors report low link quality. If the sensor report high link quality, the IRR protocol is not used and the packets are sent directly over the Ethernet protocol. If the IRR protocol is used, the receiver sends an acknowledge packet for every data packet. This allows a reliable communication if the sender repeats the sending of data messages, for which it did not receive an acknowledgement. On the other hand, the sending and receiving of the acknowledge packets consumes energy, which would not be required, if the communication channel would be reliable by itself.

In order to build this system, we implemented a stack builder that includes an idle repeat request (IRR) reliability protocol in the protocol stack only when sensors report low link quality. The link quality is determined by a sensor that divides the current with the
maximum possible wireless link quality. Our link-quality-aware network architecture is shown in Figure 10.1.

![Diagram of network architecture](image)

**Figure 10.1:** Implementation of the node architecture for the link-quality-aware protocol stack.

We evaluated our architecture on commodity notebooks. In order to obtain reproducible results, we used a wired connection between the test machines and used the Linux traffic control tool `tc` with the `netem` discipline [63] to emulate packet loss. We recorded the link quality (shown in Figure 10.2) between two nodes while walking around in our office building. Generally, the link quality was good when the two nodes were close to each other and poor when the two nodes were far away from each other. However, the reported link quality fluctuated considerably, even when the two nodes did not move. Simultaneously, we measured that packets got lost when the link quality was below 35%. The link quality was obtained by a call to `ioctl(2)` with `SIOCGIWSTATS` which reports a device driver specific metric of the observed link quality. We have used this recording as realistic input for our emulation.

Our stack builder requests to be notified by the sensor daemon when the signal strength falls below a threshold of 40% or increases beyond 50%, which is shown as a grey bar in Figure 10.2. Upon such an event, it either inserts or removes the reliability module and renegotiates the protocol stack with the neighboring node. The lower threshold
for renegotiation ensures that the reliability protocol is inserted to the protocol stack before the link quality reaches the critical value of 35%. The upper threshold is used to avoid frequent adaptations of the protocol stack.

For evaluation purposes, we compared the data loss rate and the total number of packets sent for different configurations; see Table 10.1. The configuration labeled ‘unreliable’ never uses reliability, ‘reliable’ always uses reliability, and ‘autonomous’ dynamically adapts itself to the link quality.

The configuration with no reliability lost on average 31% of the packets, whereas we didn’t observe packet loss in the other two configurations. However, this reliability comes at a price. The overhead (in terms of sent packets) for achieving reliability was 128% for the configuration that was statically configured to use the reliability protocol. The total overhead for the dynamic configuration was 100% split in 60% for sending acknowledgement and retransmission packets and 40% for sending the protocol stack reconfiguration messages. This clearly shows that adaptive protocol stacks can reduce the total communication overhead in dynamic scenarios. However, the adaptation algorithm has to be designed carefully to avoid increasing the total overhead by sending too many stack reconfiguration messages.
Table 10.1: *Comparison between static and autonomous configurations over 140 seconds.*

<table>
<thead>
<tr>
<th>config.</th>
<th>data packets</th>
<th>reliability</th>
<th>reconfig.</th>
<th>total packets</th>
<th>packet loss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>unreliable</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td>100%</td>
<td>31%</td>
</tr>
<tr>
<td>reliable</td>
<td>100%</td>
<td>128%</td>
<td>-</td>
<td>228%</td>
<td>0%</td>
</tr>
<tr>
<td>autonomous</td>
<td>100%</td>
<td>60%</td>
<td>40%</td>
<td>200%</td>
<td>0%</td>
</tr>
</tbody>
</table>
11 Autonomic HW/SW Mapping in EmbedNet

Figure 11.1 shows our approach for mapping the functional blocks to either hardware or software autonomically. The overall architecture corresponds to the DPS architecture shown in Figure 3.1. The configuration daemon consists of a mapping algorithm that determines the hardware / software mapping of the individual blocks. Therefore, it obtains information from three different sources: goals, sensors, and models. The goals are specified by the user and might be “no packet loss”, “minimize CPU load caused by network traffic”, etc. The sensors collect statistical information such as “packets per second per flow” or “CPU load”. The models describe the overall system and can either be known or learned at run time. Examples of models are “a packet is processed faster in hardware than in software”, or “packets per second per flow does not change between two measurement intervals”. Based on this input, the mapping algorithm determines the hardware/software mapping and also initiates the reconfiguration of the hardware, should that be required. Some specific mapping algorithms are described in Section 11.1.

To evaluate the autonomic hardware/software mapping, we used two simple protocol stacks that are made up from functional blocks of type privacy or security. The functional block of type privacy implements an AES (advanced encryption standard [61]) encryption module that uses electronic codebook (ECB) mode. This encryption algorithm requires rather complex operations on the packet and can therefore be seen as a representative of all functional blocks that require heavy packet processing. We do not advocate using ECB mode,
because it reveals patterns in plaintext, but it has the same processing characteristics as more advanced modes and is thus well suited for a performance evaluation. The functional block of type security implements a simple intrusion prevention system (IPS) by detecting non-shortest-form UTF8-encoding attacks [64]. This operation requires examining the whole payload once and is therefore representative of all functional blocks that require access to the whole network packet. This is often required, e.g., for calculating a checksum.

The overall system consists of two applications. One application receives packets over a protocol stack that is built from the Ethernet and from the AES functional block, the other application receives packets over a protocol stack that is built from the Ethernet and the IPS functional block. The Ethernet functional block is always mapped to hardware, whereas the applications are always in software in Linux user space. The AES and the IPS blocks can be mapped either to

Figure 11.1: Self-aware hardware/software mapping.
hardware or to software. However, only one of the two blocks can be mapped to hardware at a given time.

The focus of this evaluation is not to show how our architecture compares with other computing platforms, but to show how different mapping algorithms compare on our platform. To this end, we tested our system with three different mapping algorithms and three different network traffic mixes.

### 11.1 Mapping Algorithms

We have implemented the following three mapping algorithms M1–M3 that map functional blocks to either hardware or software.

**M1 Optimal static mapping:** This mapping is obtained by an engineer who has god knowledge. He knows how long it takes to process a packet in software and in hardware and he knows the traffic from each scenario. With this knowledge he builds an optimal, but static mapping.

**M2 Simple autonomic mapping:** This mapping algorithm measures the number of packets per second that arrive in software for each protocol stack and puts the functional block in hardware that has to process more packets.

**M3 Smart autonomic mapping:** This mapping algorithm senses the number of packets per second for each flow and also senses how long it takes to process a packet for each protocol stack. As a decision basis it uses the moving average of the packet rate weighted by the packet processing time. Additionally it only changes the mapping if this number of one flow exceeds 110% of the other flow. Those measures should help to avoid fluctuations of the mappings during periods in which both applications receive similar amounts of traffic or during time periods with short spikes in the traffic.

For a fair evaluation, we compare the results of the optimal static mapping with the results obtained by the two autonomic mapping algorithms for varying scenarios.
11.2 Network Traffic Mixes

We have tested the three mapping algorithms with three different network traffic mixes T1–T3, which are representative of network traffic seen by an end node (as opposed to an intermediate node such as a network router) in the Internet.

**T1 Non overlapping constant bit-rate traffic:** This traffic mix could correspond to a user who first streams a video that requires one protocol stack and after this he makes a phone call that requires the other protocol stack. We sent first 60,000 packets at a rate of 1,000 packets per second for the protocol stack with IPS and then we sent 60,000 packets at a rate of 1,000 packets per second for the protocol stack with AES.

**T2 Overlapping constant bit-rate traffic:** This traffic mix corresponds to a long running application which receives packets over a protocol stack with IPS configured and a short running application that requires packet encryption. We sent 120,000 packets with a rate of 1,000 packets per second to the protocol stack with IPS and in the middle of this transmission we sent 20,000 packets at a rate of 500 packets per second to the protocol stack with AES.

**T3 Congestion-controlled traffic:** In order to avoid congestion in the Internet, several protocols implement congestion control algorithms. Simplified, the traffic that results from those algorithms can be characterized as follows. The traffic rate linearly increases until the algorithm detects congestion. Then the traffic rate is reduced to half of the maximum traffic rate and then increased again until congestion is detected. This results in a typical sawtooth packet rate sequence. We sent 31,275 packets with a maximum packet rate of 200 packets per second to both protocol stacks. Both traffic flows were shifted, so that the peak traffic rate for both flows alternate.

In all scenarios we send maximum sized Ethernet packets (1500 bytes).

11.3 Experimental Results

Figure 11.2 gives an overview over the resulting hardware/software mapping over time for the nine combinations of mapping algorithms.
and network traffic mixes. The presented values were measured while executing the different scenarios.

Figure 11.2: Traffic mixes and resulting hardware/software mappings for the three mapping algorithms over time. The static algorithm always puts the AES block in hardware. The dynamic algorithms start with no functional block mapped to hardware, but adapt the mapping to the traffic.

The optimal static mapping implements the AES block in hardware and the IPS block in software. This can be explained by the fact that it takes the AES block 3.5 times longer to process a packet than the IPS block. The simple autonomic algorithm might lead to non-optimal solutions since it does not take the time required to process a packet into account, nor does it prevent frequent changes in the mapping, which lead to increased reconfiguration overhead. Finally, the smart autonomic algorithm adapts the hardware / software mapping, if this is beneficial, but it avoids frequent changes in the mapping that would decrease the overall performance.
In order to better evaluate the differences in the algorithms, we measured the CPU utilization while processing packets (Figure 11.3), and the packet loss in percent for each traffic mix, for each application separately (Table 11.1), and also for the overall scenario (Figure 11.4).

![CPU utilization for three algorithms with three traffic mixes](image)

**Figure 11.3:** Measured CPU utilization for the three traffic mixes for the three algorithms.
Figure 11.4: Observed packet loss for the three traffic mixes for the three algorithms.
Table 11.1: Measured packet loss for all combinations of mapping algorithms and network traffic mixes.

<table>
<thead>
<tr>
<th>scheduling algorithm</th>
<th>T1: constant bit-rate</th>
<th>T2: overl. constant bit-rate</th>
<th>T3: congestion controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>AES</td>
<td>IPS</td>
</tr>
<tr>
<td>M1: static</td>
<td>4.5%</td>
<td>0.0%</td>
<td>9.0%</td>
</tr>
<tr>
<td>M2: simple autonomic</td>
<td>1.4%</td>
<td>1.6%</td>
<td>1.2%</td>
</tr>
<tr>
<td>M3: smart autonomic</td>
<td>1.8%</td>
<td>2.2%</td>
<td>1.3%</td>
</tr>
</tbody>
</table>
We can see that the autonomic algorithms require the least CPU resources to process the network traffic mixes with constant bit-rate traffic. However, the simple autonomic algorithm requires the most CPU resources for the congestion-controlled traffic. This is because it reconfigures often, which requires CPU resources as well. This behaviour is also reflected in the highest packet loss for the simple autonomic algorithm for the congestion controlled traffic.

If the static algorithm and the smart autonomic algorithm are compared with each other the autonomic algorithm always performs better. On average it requires 15% less CPU resources and loses 1.7% fewer packets. The result of the autonomic algorithms could be improved by minimizing the overhead required for the partial reconfiguration. In Chapter 19, where we discuss future work, we present a possible architecture for such an optimized reconfiguration core.
12 Section Conclusion

In the third part of this dissertation we presented two case studies that show the benefits of the DPS architecture. In the first example, we showed that protocol overhead can be lowered when the protocol stack is adapted to time-varying network conditions. In the second example, we showed that mapping network functionality dynamically to either hardware or software can increase the performance as compared to a similar system with a static mapping.

With the implementation of those two use cases, we layed the foundation for more advanced use cases where more nodes, a bigger hardware system, and / or more advanced optimization algorithms are used. In our future work section (Section 19) we present our initial work towards such a more advanced use case.
Part IV

Towards Self-Awareness in Networking

In this part we discuss initial steps towards more autonomic optimization algorithms. We present how repeated network traffic pattern can be learned which can be used by algorithms that pro-actively optimize the provided network functionality. Furthermore, we look in the research area of self-awareness and show how the DPS architecture could benefit from further insights obtained in this research field.
Generally, an adaptation algorithm can be reactive or proactive. A reactive algorithm (Figure 13.1 top) makes an observation in time slot $t-1$, calculates an optimal mapping or an optimal protocol stack for this traffic distribution and uses this mapping in time slot $t$. This approach is especially useful for traffic changes introduced by a single user where the protocols and traffic mixes depend on user behavior, and which usually contain long periods of inactivity.

A proactive algorithm (Figure 13.1 middle and bottom) changes the hardware/software mapping or the protocol stack before the traffic mix changes. This requires knowledge from past traffic mix changes that can be applied to the current situation. On end nodes this might be a regular pattern for checking emails or storing a backup to a server. On intermediate nodes this might be the variation of network traffic that occurs on a daily basis. We can distinguish between algorithms that know the period and algorithms that learn the period of repetitive traffic. Algorithms that know the period could assume for example that tomorrow’s traffic will be much like today’s. This approach is particularly useful for aggregated traffic.

Algorithms that learn the period, could use the partial autocorrelation function (PACF). The PACF determines repeating patterns based on data from a time series. For a given lag $\ell$, the PACF is the correlation of the traffic at time $t-\ell$ with the traffic at time $t$, considering possible linear dependencies due to lesser lags. In periodic signals, the PACF will be significant at lags that are multiples of the period.

We used the PACF to analyse ordinary network traffic. Figure 13.2 shows the PACF for Email traffic. We can clearly see a significant correlation at lags 60, 120, 180, and 240 seconds. This indicates that
the observed traffic has a period of one minute, which corresponds to the configuration of the email client which checked for emails every minute. This insight would allow an optimization algorithm to already setup the protocol stack required for checking emails, or to adapt the hardware / software mapping to the expected traffic, before the application actually checks the emails. Obviously, this is not as useful for email traffic, since this is usually not a big data volume, but the same principle could also be used to learn other traffic patterns.

There are also significant negative autocorrelations just before and after these large peaks. This means that traffic just before and just after a peak is likely to be much less, and resources required for handling the periodic traffic can be freed up immediately after the peak.

In a similar analysis for a protocol without periodic traffic, the PACF did not show any significant periods.
Figure 13.2: Partial autocorrelation of IMAP traffic, $x$ is lag, dashed line is significance level $10^{-4}$. 
14 Self-Awareness in Networking

Since the development of computers, researcher try to make them as intelligent as possible. One aspect is the desire that computing system can react to changes in the environment independently without human intervention. To this end, several approaches were elaborated. The most well known are the IBM autonomic computing initiative [65], research in artificial intelligence [66], biologically-inspired computing [67], or even standard control theory.

Within the scope of this dissertation, we looked at proprioceptive computing systems. The word proprioception is built from a combination of two words: The Latin word "proprious", meaning one’s own and "perception". Proprioception is therefore the enabler for building awareness about one self which is required to express one self by adapting the behaviour.

Our collaborators in the EPiCS project defined a proprioceptive framework as well as five levels of self-awareness [68]. Those levels were defined in analogy to psychology [69], where the term self-awareness has a long tradition. In this chapter we show how our DPS architecture fits in the proprioceptive framework and how the five levels of self-awareness can be applied in networking.

14.1 Proprioceptive Framework

Figure 14.1 shows the proprioceptive framework as defined in the EPiCS project.
14.1 Proprioceptive Framework

The relation of the building blocks to the DPS architecture is explained in the following:

- **Goals and Controller:** Those two blocks determine for what the node should be optimized. There might exist several (also conflicting) goals, from which the controller selects the goal (or set of goals) that currently should be optimized.

  In the DPS architecture goals might correspond to maximize throughput, minimize energy consumption, minimize packet loss, maximize privacy, etc. The goals can be determined by an application or by the user. The controller corresponds to the strategy finder which is foreseen in the architecture, but not yet implemented.

- **Sensors and Actors:** Sensors collect information about node internal as well as node external information. This information is used to determine how the system should adapt. This adaptation is done through the actors, which can again be internal to the node, or external.

  In the DPS architecture sensors might collect information on network conditions, such as the signal to noise ratio, latency, throughput, or node local conditions, such as the available battery power or the current energy consumption. The actors
correspond to the partial reconfiguration of the FPGA when the hardware/software mapping is changed, or to the adaptation of the core machinery, when the protocol stack should be changed.

- **State and context:** The state and context block builds the knowledge of the state of the node. It consists of models and a private and public self-awareness engine. The former evaluates information from node internal sensors, whereas the latter evaluates information from the environment.

In the DPS architecture the private and public self-awareness engine are combined in the sensor daemon which collects and aggregates the data obtained from the sensors. In our implementation the sensor daemon is rather basic, but it could be enhanced to gain more advanced knowledge. Depending on the actual optimization goals, many models are required: For example, a model of the functionality provided by the functional blocks, a model of the network conditions and how they affect communication, and a model of the available system resources.

- **Self-expression engine:** The self-expression engine is responsible for triggering the reconfiguration of the node based on the knowledge gained from the state and context block. While the self-expression engine determines how the node should be adapted, the actors do the actual adaptation.

In the DPS architecture the self-expression engine corresponds to the stack builder. It determines the hardware/software mapping of the functional blocks, and it negotiates the protocol stack to be used with its communication peers.

To summarize, the DPS architecture fits nicely in the proprioceptive framework. This gives us the reason to expect that algorithms defined to optimize a proprioceptive node could be easily adapted for the DPS architecture. This would lead to a better adaptation of the DPS node.

### 14.2 Levels of Self-Awareness

In this section we describe the five levels of self-awareness as defined within the EPiCS project. The awareness level increases with each
new level and should allow a node to make better decisions on how to adapt itself. We also show how the general concept can be applied to a networking scenario.

1. **Stimulus-aware:**

   **General** A stimulus-aware system can react to stimuli, but it does not know anything about past/future stimuli. The stimuli might result from internal or external sources.

   **Networking** A networking system that is stimulus aware can adapt its behaviour based on stimuli. This might be the adaptation of the packet sending rate based on network congestion, or the change of the hardware/software mapping based on the current processing requirements.

2. **Interaction-aware:**

   **General** An interaction-aware system is aware that other nodes exist and that it can interact with them through stimuli and its own actions.

   **Networking** All networking systems interact with other nodes, however, to be interaction-aware the system has to be able to change its behaviour based on other nodes. A typical example would be the negotiation of a protocol stack.

3. **Time-aware:**

   **General** A time-aware system has knowledge of past stimuli and might guess future stimuli.

   **Networking** A time-aware networked system can analyze the received packets over time and draw conclusions from it. For example, it might be able to learn repeated traffic patterns and adapt the protocol stack already in advance.

4. **Goal-aware:**

   **General** A goal-aware system knows the current goals and constraints explicitly which allows it to change its behaviour if the goals change.
In a networked system, several goals can be achieved. On the one hand there are goals that characterize the packet transfer, such as reliability, privacy, low latency, etc. that might require the adaptation of the protocol stack, on the other hand there are goals that characterize the local node, such as maximize throughput or minimize the energy consumption.

5. **Meta-self-aware:**

**General** A meta-self-aware node knows which level of self-awareness it implements, and can reason about the benefits and costs involved. It is also able to change the level of self-awareness in order to achieve its goals.

**Networking** Even though nothing speaks against introducing meta-self-awareness in networking, it was not targeted in this work.

On each of those levels, specific algorithms are required to achieve the described level of self-awareness. We expect that with the wider adaptation of the concept of self-awareness in computing systems, a wide set of algorithms to implement those levels will be developed. Those algorithms can then be applied to networked systems and enhance their capability for self-optimization.
In the fourth part of this dissertation we looked into how more advanced optimization algorithms could be developed.

First, we showed how it is possible to autonomically learn repeated network traffic patterns. This knowledge could be used to proactively change the protocol stack which could minimize the impact of the reconfiguration to the packet processing performance.

Second, we introduced the novel research area of self-awareness and we showed that the DPS architecture fits well within the proposed framework. This would allows us to directly benefit from new optimization techniques developed for self-aware systems which hopefully would increase the overall system performance.
Part V

Conclusions

We start this part by setting our work into relation with the work presented by other research projects. Then we review our contributions and we critically assess the chosen research area. Finally, we provide an outlook to the most interesting topics that could be investigated in the future.
16 Comparison with Related Work

16.1 Future Internet Architectures

As introduced in Chapter 1.1 in many projects researcher work on future, clean slate network architectures. In this section, we present those architectures that have a similar focus like our DPS architecture. For each architecture we present the principles and show how it relates to the DPS architecture.

16.1.1 Recursive InterNetwork Architecture: RINA

RINA [70, 71] is based on the realization that networking is nothing more than Inter-Process Communication (IPC). The basic structure behind RINA is a Distributed IPC Facility (DIF). A DIF provides connectivity between all nodes that are a member of a given DIF, and provides functionality such as naming, access control, address resolution and data transfer. In order to divide the global network into smaller parts, DIFs can be layered on top of each other. This ideas is similar to the idea of compartements in the ANA architecture. However, each DIF offers exactly the same functionality, and only policies have to be used to adapt a DIF to a certain scope. This also implies that only one data transfer protocol, called EFCP (Error and Flow Control Protocol), is available in the architecture. This protocol is based on Delta-T [72] which was developed in 1978. Delta-T is a timer based
protocol which does not require any handshake mechanisms. The EFCP protocol is split into two protocols: The Data Transfer Protocol (DTP), for addressing, fragmentation, etc. and the Data Transfer Control Protocol (DTCP), for flow control, retransmission control, etc.

This repeated use of the same DIF, with the same data transfer protocol on each layer, is fundamentally different from the current Internet architecture, where a layer is strongly coupled with the protocol that implements that layer. In order to evaluate how well RINA works in comparison to the Internet architecture, a dedicated project was funded: IRATI: Investigating RINA as an Alternative to TCP/IP [73]. IRATI is a European Union research project which started in 2013. Within the IRATI project, RINA will be implemented from scratch on a Linux based system. However, since the project only started recently, no concrete insides are available yet, but it will be interesting to see their conclusions at the end of the project.

As compared to our dynamic protocol stack architecture, RINA has a different focus. In the DPS architecture we focus on time-variant network conditions and how we can provide optimal service by adapting the protocol stack (the layers) to the current situation. In RINA, time-variant network conditions are not considered. With respect to the Internet architecture, in the DPS architecture we keep the diverse sets of protocols developed over the last years and we simply suggest to dynamically adapt the set of protocols used. RINA in contrast, argues that only one protocol is required that can be configured for different layers. It would be desirable that the IRATI project shows that the RINA approach works well in real scenarios, since this would facilitate network operation considerably.

16.1.2 Recursive Network Architecture (RNA)

Similar to RINA, RNA [74, 75] is a recursive network architecture that reuses a protocol on all layers of the protocol stack. However, in RNA, this protocol is not an actual protocol but a metaprotocol. The metaprotocol offers a static set of basic services, such as discovery, negotiation and template matching. Additionally, it offers hooks to configure the actual data processing functionality and a basic protocol to negotiate the required functionality between two nodes. The functionality is composed from individual modules such as buffering,
reordering, encryption, etc. The combination of those modules can change over time.

RNA and the DPS architecture are similar with respect to the possibility of changing the protocol stack at run time. While in RNA a metaprotocol that offers hooks for arbitrary network processing is used, the DPS architecture has a more traditional approach and simply combines the individual network protocols.

16.1.3 Net SILOs

In contrast to recursive network architectures, the Net SILOS project [76] keeps a traditional understanding of layering. However, unlike in the Internet architecture, where the functionality on each layer is well defined, in Net SILOs, each application can define its own protocol stack. Each protocol stack is composed from a set of services. In addition to data processing, each service provides a set of knobs over which other services can interact with the given service. These knobs allow for easy cross layer optimization. With cross layer optimization it is possible that higher layer services can adapt their behaviour based on the observed characteristics of lower layer services.

Both, Net SILO and the DPS architecture provide the possibility that each application can have its own protocol stack. While Net SILO focuses on cross layer optimization, the DPS architecture focuses on the run time optimization of the protocol stack.

16.1.4 4WARD Node Architecture

In the 4WARD project [77,78] they built a node architecture to *let 1000 networks bloom* [77]. Therefore, a framework in which many network architectures can co-exist was developed. This allows for exploiting the possibilities of network virtualization. The authors introduce the concept of a Netlet, which corresponds to a protocol stack. When an application wants to communicate, the node selects the Netlet (the protocol stack) that best fits the requirement of this application. Each Netlet is built from a set of functional blocks that are characterized by a set of properties. This allows the node architecture to evaluate the utility of a Netlet with respect to the application needs. The node can switch between different Netlets at run time.
Both, the 4WARD node architecture and the DPS architecture, allow for different protocol stacks to co-exist in one node. While the 4WARD node architecture focuses on the selection of entire protocol stacks, in the DPS architecture we focus on building protocol stacks from individual components and adapting those components at runtime.

16.2 Network Processing with FPGAs

All projects presented in the last section have prototype implementations that are developed solely in software. Another interesting research line is to investigate in implementations for flexible network architectures that make use of hardware acceleration. This is not a straightforward task, since the introduced flexibility requires the ability to change the hardware dynamically. Therefore, several (unrelated) projects looked at how FPGAs can be used to provide hardware acceleration for flexible network architectures. The projects tackle different network processing devices, such as switches dedicated for active networking, network processors, or network interface cards. A summary of all described projects can be found in Table 16.1.

16.2.1 Network Processing for Active Networking

Active networking poses two challenges to network processing: a) the functions to be executed are potentially computation intensive, and b) the functions to be executed might only be known when a packet is received. On the one hand general-purpose processors would be flexible enough for active network processing, but for some application scenarios they did not offer the required performance, and on the other hand a dedicated ASIC implementation would offer high performance but could not provide the required flexibility.

Therefore, researcher who wanted to build network processing devices for active networking were the first to realize the potential of FPGAs in network processing. Already in 1998, where FPGAs were rather small and could only be reconfigured as a whole, the first devices were built.

For example, Hadzic and Smith introduced the Programmable Protocol Processing Pipeline (P4) architecture [79]. It uses several
FPGAs and a switching array that decides which packet will be processed by which FPGA. They can add new functions to the system by reconfiguring an FPGA, and they implement different protocol stacks by changing the path of a packet through the FPGAs. At the same time, Decasper et al. introduced the Active Network Node (ANN) [80]. The central component of their system is a switch which has, for each input port, a CPU and an FPGA. Performance-critical functions are executed on the FPGA, which can be reconfigured by the CPU. The Plato architecture is an FPGA-only architecture for Active Networks [81]. In Plato it is not foreseen to reconfigure the FPGA at runtime.

All three projects implement network devices that are located within a network, e.g., they receive packets from a network interface and send the packet out through another network interface. They use Asynchronous Transfer Mode (ATM) for data transfer as opposed to Ethernet, which is in common use today. The use of ATM cells makes packet processing somewhat easier, since all packets have a fixed length.

In contrast to those architectures, EmbedNet is an implementation for end nodes, where packets are forwarded between an application running on top of an operating system in software and a network interface in hardware. This implies completely different design choices and leads to a completely different architecture.

16.2.2 FPGA-based Projects for IP Processing

Packet processing devices with FPGAs were not only built for active networking, but also for normal IP packet processing.

In 2000, researchers at the Washington University in St. Louis introduced an FPGA-based system called FPX [82] that enhances the Washington University Gigabit Switch [83] with reprogrammable features. They used two Xilinx FPGAs on a custom board. Several research projects were done on the FPX in the areas of IP routing, video processing, and partial FPGA reconfiguration.

The RiceNIC project [84] developed an open network interface card for prototyping and educational purposes. Their project is built on an off-the-shelf FPGA board with a PCI interface to a host system and two embedded processors on the FPGA. After a packet is received by the network interface (in RX direction) or by the DMA controller
(in TX direction) it is mostly processed in one of the CPUs, but also additional hardware accelerators can be built. The general idea is to build the system once and then configure the FPGA. It is not foreseen to reconfigure the FPGA at run time.

Kachris et al. [85] evaluate run time reconfigurable network processor designs based on FPGAs. Their architecture receives packets on an embedded processor and is able to dynamically reconfigure hardware accelerators. Their design uses three different hardware accelerators. Moreover, these accelerators could be implemented multiple times. In order to find the best mapping of accelerators, they use Integer Linear Programming (ILP) on five different traffic mixes at design time. At run time, their system analyzes the actual traffic mix and chooses the one configuration corresponding to the closest pre-analyzed traffic mix.

From 2002 to 2009 the German research association DFG [86] funded a priority program entitled *Reconfigurable computing systems* [87]. Within this program two projects were using FPGAs to build more powerful network processors. In the DynaCore project [88] the network processor receives packets and executes the protocols that are computationally cheap. Packets that need processing through computationally expensive protocols are forwarded to an FPGA that is connected to the network processor over a Gigabit Ethernet interface. Partial reconfiguration of the FPGA can be used to change the functionality provided by the FPGA. There is a limited set of protocols that can be executed on the FPGA and the transition condition from one configuration to the next is statically defined.

In contrast to this architecture the FlexPath NP [89] architecture receives packets in the FPGA and processes them completely in hardware if all required protocols are available. Only packets requiring different protocols are forwarded to a CPU cluster. In this project, the FPGA cannot be reconfigured at runtime. However, the FlexPath and the DynaCore project can be combined, where the FlexPath board would first receive the packet and if required, it could forward packets to the DynaCore board.

In contrast to FPX, Kachris et al., DynaCore, and FlexPath the EmbedNet architecture is concerned with end nodes and not with intermediate nodes. This is similar to the RiceNIC project. Both, RiceNIC and EmbedNet use embedded CPUs, but in RiceNIC they are solely used for packet processing, whereas in EmbedNet the whole
operating system is executed on the embedded CPU. This makes EmbedNet a standalone system, whereas RiceNIC can only be used in combination with a host system. Additionally, in EmbedNet the network functionality can be reprogrammed at run time, whereas this is not possible with RiceNIC.

16.2.3 Current FPGA Boards

While all projects described in the last sections use outdated hardware, we describe here a few FPGA boards that are currently supported.

The main objective of the NetFPGA project [90, 91] is to build FPGA boards that can be used for research and education. The NetFPGA project provide already their second board which offers four 10Gbit/s Ethernet interfaces, a PCI Express interface to the host system, and a Xilinx Virtex-5 FPGA. The community around NetFPGA offers several reference designs that make it easier to start working with the board.

A commercial alternative are the COMBO FPGA Board [92] from INVEA-TECH. The company offers several boards that support 10Gbit/s Ethernet interfaces with Xilinx Virtex-5 or Xilinx Virtex-7 FPGAs and a PCI express interface to the host system. INVEA-TECH also offers a software suite that facilitates the development of networking programs for their board.

In addition to specialized networking boards, general-purpose FPGA evaluation boards, such as the Virtex-6 FPGA ML605 Evaluation Kit from Xilinx [93] or the Arria V GX FPGA Development Kit from Altera [94], can also be used to develop networking applications. They do not have frameworks dedicated to network programming but instead have a wide range of supported peripherals, have professional documentation, and are relatively cheap. This is especially interesting for mixed research groups that share a single hardware platform and that have a diverse range of requirements on their platform. For these reasons, we used the ML605 Evaluation Kit from Xilinx in this dissertation.
**Table 16.1: Comparison of Network Processing Devices built on FPGAs.**

<table>
<thead>
<tr>
<th>Year</th>
<th>Project</th>
<th>Type</th>
<th>Design</th>
<th>Reconfiguration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998</td>
<td>P4</td>
<td>ATM network processor for active networking</td>
<td>Custom board with several FPGAs</td>
<td>whole FPGA</td>
</tr>
<tr>
<td>1999</td>
<td>ANN</td>
<td>ATM switch for active networking</td>
<td>One CPU and one FPGA for each input port</td>
<td>whole FPGA</td>
</tr>
<tr>
<td>2001</td>
<td>Plato</td>
<td>ATM switch for active networking</td>
<td>Custom FPGA board</td>
<td>no</td>
</tr>
<tr>
<td>2000</td>
<td>FPX</td>
<td>Network card for the Washington University Gigabit Switch</td>
<td>Custom FPGA board</td>
<td>partial reconfiguration</td>
</tr>
<tr>
<td>2007</td>
<td>RiceNIC</td>
<td>Network Interface Card with a PCI interface</td>
<td>Off the shelf FPGA board with embedded processors</td>
<td>no</td>
</tr>
<tr>
<td>2006</td>
<td>Kachris et al.</td>
<td>Network processor</td>
<td>Of the shelf FPGA board with embedded processor</td>
<td>partial reconfiguration</td>
</tr>
<tr>
<td>2006</td>
<td>DynaCore</td>
<td>Network processor</td>
<td>Of the shelf FPGA board</td>
<td>partial reconfiguration</td>
</tr>
<tr>
<td>2008</td>
<td>FlexPath NP</td>
<td>Network processor</td>
<td>Of the shelf FPGA board</td>
<td>no</td>
</tr>
<tr>
<td>ongoing</td>
<td>NetFPGA</td>
<td>4 port Ethernet NIC</td>
<td>Custom board with one FPGA</td>
<td>project dependent</td>
</tr>
<tr>
<td>ongoing</td>
<td>INVEA-TECH</td>
<td>1 - 4 port Ethernet NIC</td>
<td>Custom board with one FPGA</td>
<td>project dependent</td>
</tr>
</tbody>
</table>
The contributions in this thesis can be summarized as follows:

- **DPS architecture**: We tackled the problem of the difficulty to add truly innovative changes to the Internet architecture, by proposing a new, clean slate network architecture. Our DPS architecture allows for the inclusion of novel protocols at run time and even for the dynamic adaptation of the protocol stack during communication. On the one hand this allows researchers to easily build a system where they can test their newly developed protocol (or integrate their protocols in an already running system) and on the other hand it allows for minimizing the protocol overhead when a dedicated protocol stack is used as opposed to a one size fits all protocol stack. With our implementation of the DPS architecture on a general-purpose CPU, we have shown that the proposed architecture is a) feasible, b) performs similarly to the standard networking functionality provided by the Linux operating system, and c) protocol overhead can actually be minimized by dynamically adapting the protocol stack.

- **EmbedNet execution environment**: A raise in the performance of computing systems is often obtained by deploying hardware acceleration. However, this is usually only possible, when the tasks to be computed do not change over time. This is in contrast to the DPS architecture, where the tasks (the protocols) may change at any point in time. Therefore, we developed EmbedNet, an execution environment for the DPS architecture,
which provides dynamically configurable hardware accelerators. This is achieved by building a system-on-chip design on an FPGA. The hardware accelerators can be (re-)configured into predefined areas on the FPGA while the rest of the system is processing data. With our implementation and evaluation we have shown a) that it is feasible to provide hardware acceleration for flexible network architectures and b) that the dynamic mapping of network functionality to hardware or software might increase the overall system performance.

During the work on this dissertation the following peer reviewed papers were published:

- Main author


• Co-author


Additionally, we enhanced the Xilinx hwicap driver of the Linux kernel with support for virtex 6 FPGAs. The following two patches where accepted to the official Linux source code:

- [http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=commitdiff;h=1790625feb3904f54c82e469a2e5997c3a01f4fa](http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=commitdiff;h=1790625feb3904f54c82e469a2e5997c3a01f4fa)
- [http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=commitdiff;h=73eb94a094e54cb81c41c64e59eb5d6a05ecb045](http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=commitdiff;h=73eb94a094e54cb81c41c64e59eb5d6a05ecb045)
18 Critical Assessment

While thinking about the overall project, the following two key observations have to be made, before suggesting to other researchers to work in this area:

- **Real world deployment:** While it is tempting to propose a clean slate network architecture, this comes with the drawback that a large scale deployment in a network spanning around the world or even further, is not realistic. Therefore, one cannot claim to have an immediate impact from which the general public could benefit. However, in the Network Management Research Group (NMRG) [95] of the Internet Research Task Force (IRTF) [96] they are actively discussing how to deploy concepts from autonomic systems in the current Internet architecture. And the TAPS bof [97] (an Internet Engineering Task Force (IETF) [98] pre-working-group effort) considers, how to change the socket API to allow applications to more precisely state their communication needs. Both initiatives are only at their beginning and therefore could make use of the insights gained in this dissertation. With those initiatives, the ideas from this dissertation could find their way in the current Internet.

- **Keeping up with technology:** Even though the DPS architecture itself, and the concepts of the EmbedNet execution environment can be applied to any technology, the actual implementation of EmbedNet is heavily linked to the Xilinx Virtex-6 FPGA which was newly introduced during the course of this dissertation. While at the beginning of the dissertation we still used an older FPGA platform, today, we would already chose
a newer platform again. Currently, the most promising FPGA platform in the Xilinx Family is the Zynq FPGA [99] which was introduced in 2012. The Zynq board combines an ARM processor with configurable logic cells which is exactly what is required for the EmbedNet architecture. However, in order to make use of the new features introduced by the new FPGA platforms, the porting of projects implemented on older platforms takes a considerable amount of work. While this certainly would increase the observed system performance, the attractiveness of the system to other research groups, and the applicability in real world scenarios, it would rarely lead to new insights that are valuable for research. Therefore, performing research on devices that are highly technology dependent needs to make a trade off between development time and technology. We think, our decision of changing the development platform once, but not twice was optimal, since a) the old FPGA was too small to fit our design and b) there was only preliminary tool support for partial reconfiguration.
19 Future Work and Outlook

We see the following research directions as interesting future work:

- **Advanced optimization algorithms:** During the course of this dissertation, we have focused on the development of the actual architecture and execution environment. In a next step, one could focus on the optimization algorithms, both, for determining the best protocol stack, and the best hardware/software mapping.

- **Simulation Environment:** To ease the process of finding advanced optimization algorithms, in a first step a simulation environment for the DPS architecture could be built. This is of special interest for algorithms that are concerned with the hardware/software mapping, as it is time consuming to build an actual system in which more advanced algorithms could be tested. We currently see two parts of such a simulation environment: a) a simulation of the expected throughput, and b) a simulation of the expected reconfiguration overhead.

  For the throughput estimation, the EmbedNet architecture could be divided into three parts: the hardware blocks, which can all be executed in parallel and allow for a pipelined packet processing, the software blocks, which have to be executed serially, and the hardware/software interface which has a limited bandwidth.

  For the simulation of the reconfiguration overhead the parameters shown in Table 19.1 could be taken into account. With those
parameters, the reconfiguration overhead $A$ could be estimated as follows: $A = m + c + \sum_{i=1}^{n}(s_i + r_i)$.

Table 19.1: Hardware parameter limiting the adaptation frequency.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>Time required to transmit sensor data to the reconfiguration application</td>
</tr>
<tr>
<td>$c$</td>
<td>Time to compute the next hardware/software mapping</td>
</tr>
<tr>
<td>$s_i$</td>
<td>State relocation time for block $i$</td>
</tr>
<tr>
<td>$r_i$</td>
<td>Reconfiguration time for block $i$</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of blocks to reconfigured</td>
</tr>
</tbody>
</table>

- **Real world demonstrator:** It would be interesting to see how the architecture behaves in bigger scenarios. To this end, we are currently working on a smart camera demonstrator. The goal of this demonstrator is to track an object autonomously over the field of view of several cameras. To this end, the cameras need to coordinate themselves which is done over the DPS architecture. The cameras are connected to an FPGA on which EmbedNet is running. The image processing as well as the networking is split between the software as well as the hardware. It will be interesting to evaluate the benefit of using the DPS architecture over the standard Internet architecture for this scenario.

- **Improving the EmbedNet implementation:** Currently, the EmbedNet implementation is a proof of concept implementation which could be improved in a number of ways. For example, a system with more hardware accelerators could be built, the architecture could be ported to the Zynq platform, or the actual reconfiguration time for a hardware accelerator could be improved. We are currently working towards the latter. Our approach is to replace the reconfiguration core provided by Xilinx with our own core that does not require the CPU during the reconfiguration. Initial results are promising and show a reduction of the reconfiguration time by a factor of almost twenty.
This reduced reconfiguration time allows us to do a finer grained adaptation of the hardware/software mapping which would be again interesting for new optimization algorithms.

- **Enhancement to the DPS architecture:** It would be interesting to see how the DPS architecture could be enhanced so that DPS nodes which are only connected over the standard Internet architecture could communicate. One approach would be to focus on an IPv6 [18] connection and using the fact that in IPv6 a single interface can have multiple IPv6 addresses. In DPS we could use this as follows: While the first part of the IP address is used for routing the packets over the IPv6 network, the second part could be used to identify the DPS protocol stack on the end nodes. The implications of this idea would have to be analyzed in detail.

- **Combination of Software Defined Networking (SDN) with the DPS architecture:** Within this dissertation we have shown how the protocol stack can be adapted at run time to time-varying network conditions. However, we only showed this on end nodes that were directly connected with each other. It would be interesting to show how the DPS negotiation protocol could be used in a multi-hop communication scenario, in which the intermediate nodes also perform packet processing operations (e.g., routing). One approach could be to use an SDN-based approach to configure those intermediate nodes, which would allow the negotiation packets to be forwarded directly to the end host.
List of Figures

1.1 Comparison of the Internet architecture (a) with a flexible network architecture (b). ........................ 5
1.2 Comparison of the Protocol Stacks for the Internet architecture (a) and the flexible protocol stacks architecture (b). ................................................................. 6
1.3 Dynamic reconfiguration of the protocol stack due to changing network conditions. ......................... 7
1.4 Dynamic reconfiguration of the protocol stack to deploy a bug fix. .................................................. 7

2.1 Node Architecture within ANA. ................................. 13
2.2 Protocol Stack Setup in ANA (one layer) ............... 16
2.3 Basic IDP operation. ............................................. 17
2.4 Advanced IDP operation. ...................................... 18

3.1 Overview of the node architecture. ......................... 22
3.2 Interfaces for a functional block (as used in our Linux kernel implementation). .............................. 24
3.3 Dynamic protocol stack architecture API example. . 25
3.4 Example layering constraints file. .......................... 26
3.5 Negotiation phase for setting up a protocol stack between nodes. ................................................ 28
3.6 Dispatching a packet to one of several protocol stacks. ................................................... 29
3.7 Packet format for a data message over Ethernet . . 29
3.8 Packet format for a control packet that is sent over the Ethernet protocol ................................. 30
3.9 Packet format for a control ack packet that is sent over the Ethernet protocol 31
3.10 Replacing a functional block in the DPS architecture 32
3.11 Updating the dynamic protocol stack over time 33
7.1 DPS implementation on a general-purpose processor 43
7.2 Architecture of the sensor daemon and the stack builder 45
7.3 Timeline for setting up a protocol stack 48
7.4 Performance comparison between receiving packets by the Linux kernel, by the DPS architecture in the kernel, by a PF_Packet socket in user space, and by PF_DPS sockets in user space 49
7.5 Packet reception rate for a functional block chain up to ten blocks 50
8.1 NIC with dynamic hardware 52
8.2 Overall system allowing for a self-aware hardware/software mapping 53
8.3 Dynamic hardware / software mapping 57
8.4 EmbedNet FPGA design 58
8.5 Implementation of the reconfiguration framework 60
8.6 Two functional blocks connected to a switch in the NoC 61
8.7 Packet format in the NoC 62
8.8 ReconOS Architecture with control and data path 65
8.9 Routed FPGA Desgin 68
8.10 Maximum sending rate comparison 70
10.1 Implementation of the node architecture for the link-quality-aware protocol stack 75
10.2 Measured link quality over 140 seconds. Packets got lost when the link quality was below the dashed line. The protocol stack is adapted when the graph crosses the gray bar 76
11.1 Self-aware hardware/software mapping 79
11.2 Traffic mixes and resulting hardware/software mappings for the three mapping algorithms over time. The static algorithm always puts the AES block in hardware. The dynamic algorithms start with no functional block mapped to hardware, but adapt the mapping to the traffic. 82

11.3 Measured CPU utilization for the three traffic mixes for the three algorithms. 83

11.4 Observed packet loss for the three traffic mixes for the three algorithms. 84

13.1 Reactive versus proactive algorithms. 90

13.2 Partial autocorrelation of IMAP traffic, $x$ is lag, dashed line is significance level $10^{-4}$. 91

14.1 Proprioceptive Framework as defined in D2.1. 93
List of Tables

8.1 Xilinx local link interface for sending packets. . . . . . . . 63
8.2 CPU characteristics. . . . . . . . . . . . . . . . . . . . . . . . . 66
8.3 Design Summary, with AES block configured. . . . . . . . 66
8.4 Overhead of partial reconfiguration. . . . . . . . . . . . . . 71

10.1 Comparison between static and autonomous configurations over 140 seconds. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

11.1 Measured packet loss for all combinations of mapping algorithms and network traffic mixes. . . . . . . . . . . . . . . . . . . . 85

16.1 Comparison of Network Processing Devices built on FPGAs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

19.1 Hardware parameter limiting the adaptation frequency. 114


[44] Internet Assigned Numbers Authority (IANA), “IEEE 802 Numbers.”


[54] W. Y. Wei and D. Huang, “XAPP691: Parameterizable LocalLink FIFO.”


[56] Xilinx, “Tri-Mode Ethernet Media Access Controller (TEMAC).”


[60] Xilinx, “Xilinx Platform Studio.”


[68] University of Birmingham, “D2.2: Mechanisms and Algorithms for Online Learning.”


[95] IRTF, “Network Management Research Group (NMRG).”


Abbreviations

AES  Advanced Encryption Standard
ANA  Autonomic Network Architecture
ANN  Active Network Node
API  Application Programming Interface
APP  Application
ARP  Address Resolution Protocol
ASIC Application Specific Integrated Circuits
ATM  Asynchronous Transfer Mode
BSD  Berkeley Software Distribution
CIDR  Classless Inter-Domain Routing
CPU  Central Processing Unit
DMA  Direct Memory Access
DNS  Domain Name System
DIF  Distributed Inter process communication Facility
DPS  Dynamic Protocol Stack
DSP  Digital Signal Processor
ECP  Electronic Code Book
EFCP Error and Flow Control Protocol
EMEA Europe, Middle East and Africa
EPiCS Engineering Proprioception in Computing Systems
ETH  Ethernet
EU  European Union
FB  Functional Block
FIA  Future Internet Assembly
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FIRE</td>
<td>Future Internet Research and Experimentation</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSL</td>
<td>Processor Streaming Links</td>
</tr>
<tr>
<td>hw</td>
<td>hardware</td>
</tr>
<tr>
<td>HWICAP</td>
<td>Hardware Internal Configuration Access Port</td>
</tr>
<tr>
<td>h2s</td>
<td>hardware to software</td>
</tr>
<tr>
<td>IANA</td>
<td>Internet Assigned Numbers Authority</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>ICAP</td>
<td>Internal Configuration Access Port</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communication Technology</td>
</tr>
<tr>
<td>IDP</td>
<td>Information Dispatch Point</td>
</tr>
<tr>
<td>IET</td>
<td>Institution of Engineering and Technology</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>ILP</td>
<td>Integer Linear Programming</td>
</tr>
<tr>
<td>IO</td>
<td>Input Output</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol or Intellectual Property</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter Process Communication</td>
</tr>
<tr>
<td>IPS</td>
<td>Intrusion Prevention System</td>
</tr>
<tr>
<td>IPv6</td>
<td>Internet Protocol version 6</td>
</tr>
<tr>
<td>IRATI</td>
<td>Investigating RINA as an Alternative to TCI/IP</td>
</tr>
<tr>
<td>IRR</td>
<td>Idle Repeat Request</td>
</tr>
<tr>
<td>IRTF</td>
<td>Internet Research Task Force</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MINMEX</td>
<td>Minimal INfrastructure for Maximum EXtensibility</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>NMRG</td>
<td>Network Management Research Group</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>PACF</td>
<td>Partial AutoCorrelation Function</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical Ethernet Interface</td>
</tr>
<tr>
<td>PPE</td>
<td>Packet Processing Engine</td>
</tr>
<tr>
<td>PR</td>
<td>Partial Reconfiguration</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ReconOS</td>
<td>Reconfigurable Operating System</td>
</tr>
<tr>
<td>RFC</td>
<td>Request For Comment</td>
</tr>
<tr>
<td>RINA</td>
<td>Recursive InterNetwork Architecture</td>
</tr>
<tr>
<td>RNA</td>
<td>Recursive Network Architecture</td>
</tr>
<tr>
<td>rSoC</td>
<td>reconfigurable System on Chip</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Networking</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous Dynamic Random Access Memory</td>
</tr>
<tr>
<td>SGMII</td>
<td>Serial Gigabit Media Independent Interface</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>sw</td>
<td>software</td>
</tr>
<tr>
<td>s2h</td>
<td>software to hardware</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TOE</td>
<td>TCP Offload Engine</td>
</tr>
<tr>
<td>UCS</td>
<td>Universal Character Set</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>US</td>
<td>United States of America</td>
</tr>
<tr>
<td>UTF8</td>
<td>UCS Transformation Format 8bit</td>
</tr>
<tr>
<td>VHDL</td>
<td>Programming language for FPGAs</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>XPS</td>
<td>Xilinx Platform Studios</td>
</tr>
</tbody>
</table>
Acknowledgements

First, I would like to thank Prof. Dr. Bernhard Plattner for giving me the possibility to do my PhD in his group. It was an interesting time in which I learned a lot. I am also grateful that I could work within two European Union research projects (ANA (FP6-IST-27489) and EPiCS (FP7-256906)) which gave my research a bigger context. Especially, I would like to thank the two project coordinators, Martin May for ANA and Marco Platzner for EPiCS, who also agreed to be the co-examiner for my PhD.

Special thanks go to my close collaborators and people who encouraged me over the course of this PhD: Rainer Baumann, Daniel Borkmann, Stephan Neuhaus, Markus Happe and Tilman Wolf. I would also like to express my gratitude towards the developer of the ReconOS project, especially Enno Lübbers and Andreas Agne. ReconOS is a great framework in which I could embed my own research. Many thanks goes also to everyone who was involved in ANA and EPiCS, and everyone who was with the communication systems group for many interesting discussions.

I would like to thank the following students who did their projects in the context of either ANA or EPiCS (in chronological order): Andreas Traber, Yujiao Huang, Stefan Kronig, Yiheng Yang, Richard Huber, Florian Deragisch, Remo Niedermann, Stephan Lienhard, Beat Gebistorf, Mathias Fischer, Diego Adolf and Martin Kos.

I’m also grateful for the great administrative and technical support I got from the following people: Thomas Steingruber, Beat Futterknecht, Friederike Bruetsch, Tanja Lantz, Caterina Sposato, Frank Gürkaynak and his colleagues from ETZ J60.1.
Last but not least I would like to thank my family: my parents for supporting me throughout my academic career, and my husband, Brian Trammell, for all the time we spent together - regardless of whether we talked about work or enjoyed our spare time.