RWTH AACHEN
RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

Distributed Systems

Adaptable and Flexible Protocols and Protocol Implementations

Rheinisch-Westfälische Technische Hochschule Aachen
LuFG Informatik 4 Verteilte Systeme

Diploma Thesis

Tim Just

Advisors:

Dipl.-Inform.	René Hummen
Prof. Dr.-Ing.	Klaus Wehrle

Registration date:	19	November	2009
Submission date:	19	May	2010

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, den 19.05.2010

Abstract

Computer networks evolved from being static structures consisting of uniform hosts, to dynamic systems with varying usage scenarios and many heterogeneous clients. Due to the large number of devices in contemporary networks, the integration of new network protocols is challenging. For this reason, new protocol designs tend to be *flexible* in order to be able to fulfill future requirements without the need for new protocols and *adaptable* in order to be applicable in new usage scenarios. Hence, network protocols developed from comparatively simple data communication services to highly adaptable and flexible, distributed software.

The mentioned evolution of network protocols at the conceptual level is not consequently applied to at the implementation level. Various protocol implementations utilize a monolithic software approach. This thesis discusses the design and implementation of adaptable and flexible protocols and argues that the application of monolithic software concepts is not sustainable in the observed domain. Instead, we examine the application of software modularity to network protocols by creating a concept for modular protocol implementations. We design and implement the modularization library `libmod` and prove its feasibility by integrating it with an existing implementation of the Host Identity Protocol as an example for flexible protocols. Based on qualitative and quantitative analysis, we conclude the superiority of modular software in comparison to monolithic software regarding the overall software quality in the field of network protocol implementation.

Acknowledgments

Many thanks to the entire Distributed Systems team for the nice working atmosphere and the support in creating this thesis. Special thanks to René Hummen for the inspiring supervision.

Contents

1

Introduction

Early computer networks needed to provide less functionality compared to contemporary networks. They connected a few known, uniform computers and provided only basic functionality such as e-mail or simple file transfer. The employed network protocols had fixed structures with static packet formats and were typically realized as *monolithic* software. In contrast, today's networks consist of numerous differing devices and have to cope with recent challenges, such as the increasing number of mobile network participants. Since the global deployment of new network protocols is challenging, recent protocol specifications aim to be adaptable in order to allow the compliance of future needs. Protocols with this *forward compatibility* entail flexible structures; for instance, protocol headers are designed extensible as to allow the exchange of information previously not thought of.

This thesis examines the design and implementation of adaptable and flexible protocols. As starting point, we analyze the feasibility of the monolithic software concept on protocol implementations. Monolithic software persists of one continuous piece of source code and there are no defined components with well-defined interfaces, nor a layered application architecture. This approach shortens the design phase to a minimum; according to their intuition, developers can almost immediately start implementing protocol functionality.

The monolithic software approach is functional for classical protocols, as long as the specification does not change. But the loose architecture can cause various drawbacks at the implementation level, such as unpredictable dependencies in the source code. For adaptable and flexible protocols, the application of the monolithic software approach is not sustainable, because of the predictable need for modifying the protocol implementation. In this thesis, we examine the feasibility of software *modularity* in the field of adaptable and flexible protocols. In modular software each component or module has an explicitly defined interface for the interaction with other components. Therefore, the dependencies between components are well-known and described in the interfaces.

Well-designed, modular software is more **flexible** than monolithic software, because modules can be replaced by other modules with the same interface without any modifications to the remaining components. Furthermore, changes inside one module that neither change the syntax nor the semantics of the external interface, do not enforce changes outside the module, because all module interaction bases on the defined interface. This directly increases the **maintainability** of software. Developing modular software is **less complex**, because each encapsulated functionality can be implemented separately and it is not necessary to comprehend the entire system all at once.

In this thesis we prove the superiority of the modular concept over the monolithic one in the field of adaptable and flexible protocols. Our argumentation regards conceptual and implementation related issues. As mentioned above we identify the following key advantages of modular software:

- Higher flexibility of functionality
- Better maintainability
- Reduced implementation complexity

Our work presents the general modularization library `libmod` for the implementation of adaptable and flexible protocols. To prove our conceptual statement, we discuss its application to an existing protocol implementation; the Host Identity Protocol for Linux (HIPL). `libmod` focuses on two usage scenarios: Firstly, it provides functionalities for creating modular protocol implementations from scratch. Secondly, `libmod` can be used to re-engineering existing protocol implementations with the goal to create modular software. The conducted performance measurements on personal computers and embedded hardware, such as smartphones or routers, show that the overhead introduced through the modularization is negligible.

The thesis is structured as follows. Chapter 2 provides the reader with background information on the used concepts. Then Chapter 3 introduces the observed research area by presenting related approaches. The problem analysis in Chapter 4 contains the exact problem statement that is covered in this thesis. Our modular design for adaptable and flexible protocols is introduced in Chapter 5 and aims to solve the depicted problems. We prove the practicability of our design by applying it to the Host Identity Protocol for Linux. While Chapter 6 describes the implementation, Chapter 7 contains the evaluation results of this process. We complete this thesis with our conclusion in Chapter 8.

2

Background

This chapter gives an introduction into topics software engineering and network protocols. Firstly, we familiarize the reader with the basics of the used software engineering concepts. The applicability of these concepts to protocol implementations is the object of our analysis. Secondly, we introduce the *Open System Interconnection (OSI) Reference Model* and the *Transmission Control Protocol* (TCP) as well as the *Host Identity Protocol* (HIP). The reference model defines nomenclature used later and depicts the interaction of different protocols. TCP and HIP will be considered as case studies throughout this thesis.

2.1 Software Engineering

The Institute of Electrical and Electronics Engineers (IEEE) defines *software engineering*[22] as:

> **software engineering.** (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

We use the term software engineering in the sense of this definition.

2.1.1 Software Development Process

A *Software Development Process* is a sequence of activities with the goal to create functional software in an efficient way. There are several models describing software development on an abstract level, such as the *waterfall model*[39] or the

spiral model[39]. Process models generally contain the following phases: requirements analysis / specification, design, implementation, verification, integration and maintenance.

The **Requirements Analysis** phase contains the process of breaking down the customer and user needs. There are several methods to figure out which functionality the planned system should provide. For example, stakeholder interviews, use case analysis and prototyping are well-known approaches. Result of the requirements analysis is a *specification* in written form.

During the **Design** phase, software engineers work on how to realize the specification. These considerations effect in the software architecture. Through this, an abstract model of the software is built and its structure is defined. In design-focused software development approaches, software components and their interaction are also specified during the design phase.

Implementation is the transfer from design to software. Developers use a programming paradigm (for example object-orientation, imperative programming) and a programming language (for example Java, C) to realize the designed system. The result is a computer program or application that should fulfil the requirements.

There are multiple possibilities of error in a software development process. To avoid this, **verification** checks the two transitions from specification to design (design errors) and from design to software (implementation errors, bugs). These checks can be a static analysis of sources or even a dynamic execution of the application against test cases. The software is considered as functioning when all checks were successful.

Integration is the deployment of software in the customers environment. During this phase, the new software is made accessible to the customer. Today's software generally interacts with other software systems. Software rarely operates isolated. For this reason, the integration phase also contains the interconnection of collaborating software systems. If necessary, interfaces can be configured and the systems adjusted to each other.

The last phase of software development is the **Maintenance**. Software maintenance is the modification of software after its integration. Besides error correction it includes adaption to new requirements, (performance) improvement and prevention. Prevention is an important activity, because software does not abrade in classical sense. Even over years of usage, the software functionality does not change. But in most cases, the environment in which the software is used, changes. Therefore, a software becomes less usable over time. This fact is commonly known as software aging[35]. Software maintenance, and especial prevention, aims to counteract this aging. Dependent on the quality of software and the applied paradigms, software can be more or less maintainable. Software *maintainability* can be expressed as the sum of *understandability*, *modifiability*, *extendibility* and *testability*[12].

2.1.2 Monolithic Software

Monolithic software denotes software consisting of one part. The entire functionality (including user interface, access control, error handling and data processing) is

implemented in one component. Thus, the design phase for monolithic software is short. For example, the classification of components and their interfaces is omitted. The implementation can begin early and hence the development process is accelerated; but only for the short term. Due to the loose architecture, dependencies may occur anywhere in the source code. Therefore, modifications can enforce other changes at dependent positions - this complicates the maintenance of monolithic software. Thus, the last phase of the development process is more time consuming than in other approaches.

In general, the monolithic software approach is useful for small programs or programs with fixed functionality. For these programs assumptions on the program flow can be made. Hence, direct statements are applicable. This immediacy tends to result in better performance.

Throughout this thesis we use the term monolithic to denote the characteristics of software on the implementation level as stated above. There is another meaning of *monolithic*, regarding operating system kernels. A kernel is called monolithic if all operating system components (including process management, file system, device management) are built together and run in the same context; the privileged kernel space. In contrast, microkernel operating systems are built of multiple separated components. Only the minimal functionality of the operating system runs in the kernel space - everything else runs in the user space with limited privileges. Popular examples for monolithic kernels are most *UNIX* variants[7].

2.1.3 Modular Software

Modular Software is a design approach based on the design concept *Separation of Concerns*[39]. The basic idea of this concept is, to subdivide the considered problem into multiple, self-contained concerns. The complexity of realizing one concern is smaller than the overall complexity. Furthermore, the accumulated complexity for realizing all concerns can be smaller than the overall complexity, because it is not necessary to deal with the entire problem all at once. Instead the sub-problems (or concerns) are identified, solved and combined into the overall solution. This is a well-known divide-and-conquer strategy. In many cases the effort of division and assembly is minor compared to the effort of understanding the entire problem[11].

Modularity is an application of the separation of concerns approach to software design. The overall functionality of a software is divided into multiple building blocks called modules. Modularity reduces implementation complexity, because modules can be implemented one after another. The modular design needs to specify interfaces for all modules, otherwise the module integration would fail. The modules interact exclusively through the defined interfaces and therefore all dependencies are explicit. Due to this explicitness, one can use a module without knowing how the implementation is realized - only the interface needs to be known. Therefore, changes inside a module do not require changes in other modules. One can develop and maintain a module, without consideration of the dependent modules, as long as the interface is stable. This provides a high flexibility and good maintainability of the software. Multiple developers can work independently on different modules and - as long as the interface specification is fulfilled - conflicts do not appear. Modules

can be replaced by other modules with the same interface without any issues. The error correction in one module does not entail any changes in other modules.

Furthermore, well-designed modules are reusable. Modules for common tasks, such as user authentication, can be used in multiple software projects. But a modular design does not automatically contain independent and reusable components. Each dependency of a component reduces it's reusability. In general, components should have high *cohesion*, that is all parts of one component belong together[12]. In contrast, the *coupling* between different components should be loose[12], that is components should have as less dependencies as possible. These two conditions develop the component's reusability to the maximum.

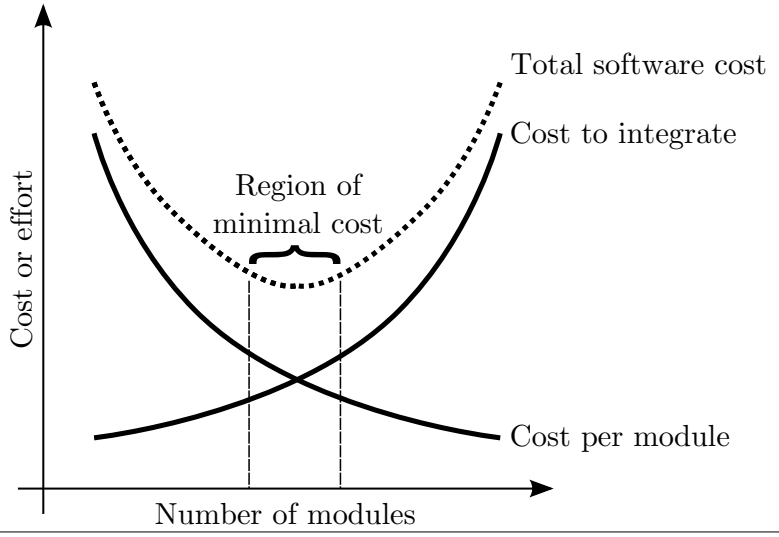

Figure 2.1 Module trade-off according to Pressman[39].

The drawback of a modular design is the introduced overhead due to the encapsulation of functionality. Modules need to be integrated and connected to offer the system functionality. The module integration, including module identification and interface specification, causes additional effort that does not occur without modularity. Furthermore, there is a trade-off regarding the optimal number of modules - that is the number of modules with minimal total costs. The more modules the lower the complexity per module, but the higher the integration effort. Figure 2.1 shows this trade-off. The total effort is the sum of the module costs plus the integration costs. In theory, there is a region with minimal costs. The number of modules defined in a modular design should be in the region of minimal cost, but there is no automatism for finding the optimal number of modules. Rather, experience in designing software yields to a good modular design.

2.1.4 Design Patterns

In each domain with modelling challenges, repetitively occurring problems exist. The original definition of *Design Pattern* is from Christopher Alexander and related to the architecture of buildings and towns[5]. Alexander describes design patterns as general solution to repetitive occurring problems. Software engineers use design patterns in the same way - to describe frequently appearing problems and their solutions in software design. Using design patterns supports software engineers in

finding reusable and understandable solutions[20]. With patterns a design can be described elegantly, because patterns abstract realization details. Furthermore, design patterns catalog the expertise of software designers. One can find an approved solution for a common problem in design pattern collections. Gamma et al. have defined various patterns for object-oriented designs on an abstract level[20]. This collection groups design patterns according their purpose in three categories: *creational*, *structural* or *behavioral*. While creational patterns address object generation problems, structural patterns deal with object composition and structural patterns apply to component interaction.

Due to the generality of these design patterns, they are also applicable for other, not object-oriented implementations. In the subsequent sections we describe design patterns that are used in this thesis. Our descriptions correspond rather to procedural-style than to object-oriented implementations.

2.1.4.1 Adapter Pattern

The *adapter pattern* is a structural pattern. It is used to enable the interaction of two components with incompatible interfaces. That is, one component (the *client*) needs to execute functionality from another one (the *adaptee*), but the interfaces of the two components do not match. By creating an adapter for the adaptee, its functionality can be used. In order to execute the designated functionality, the client executes operations of the adapter. The adapter then calls the specific functions of the adaptee. A simple example for the application of the adapter pattern are so-called *wrapper* functions. If the signature of a function is not adequate in the current context, a slim wrapper function, with an adequate signature, is built around the needed function.

2.1.4.2 Command Pattern

In software systems with multiple components, these components need to interact to provide the complete system functionality. One component (the *invoker*) may perform *request*s on another component (the *receiver*). In a naive implementation, these requests were direct function calls. But these direct connections cause coupling between components and hence a single component is less reusable and less flexible. Furthermore, the request's properties and the receiver might not be known in advance. Therefore, direct function calls are not applicable. The *command pattern* provides a better solution by decoupling invoker and receiver. All possible receivers share a general request structure - for instance, the same function signature. Thus, the invoker can abstract concrete requests and handle with general instances. During runtime the general instances are filled with concrete requests.

2.1.4.3 Publish/Subscribe Pattern

As the command pattern, the *publish/subscribe* is a behavioral pattern. The publish/subscribe (also known as *observer pattern*) is used to define dependencies between multiple components and notify all interested components about state changes.

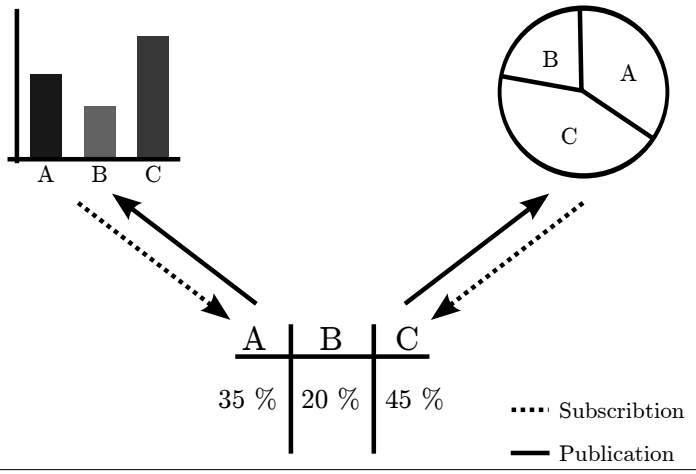

Figure 2.2 Example for the publish/subscribe pattern, according to Gamma et al.[20]

For instance, Figure 2.2 illustrates a common scenario: multiple views, such as table and chart, represent the same data. If the data changes, the views must be updated, too. In order to ensure the update of dependent components, the publish/subscribe paradigm can be used. All dependent entities (different views) subscribe to an event (change of data). Every time this event occurs, all subscribers are notified and the data of interest is published.

The publish/subscribe concept decouples the considered components on the source code level. Dependencies are not listed in the source code of the required component. Instead the dependent component registers to the required one. Removing the dependent component, does not enforce any other changes. Therefore, the communicating components do not need to be known in advance.

2.2 Protocols

A (network) protocol defines syntax and semantics for communication in computer networks. Generally, network protocols are organized in layered architectures like the *Open System Interconnection (OSI) Reference Model[23]* or the *Internet Protocol Suite*[8] (also *TCP/IP*).

2.2.1 OSI Reference Model

The *Open System Interconnection (OSI) Reference Model* is an abstract model for communication in computer networks[23]. It defines seven layers and their tasks in the communication process. The concrete protocols are not part of the specification - they are defined in separate documents.

The most characteristic attribute of the OSI reference model is its strict layering. Figure 2.3 shows the seven layers of the model. Each layer provides a service to the next higher one; and is serviced from the next lower layer. There is no communication besides this service architecture. One layer encapsulates a specific functionality and has to perform its tasks for the layer above it. Due to this strict encapsulation, a

	#	Layer	Transfer Unit
Host	7	Application	Data
	6	Presentation	
	5	Session	
	4	Transport	Segment
Network	3	Network	Packet
	2	Data Link	Frame
	1	Physical	Bit

Figure 2.3 The OSI Reference Model

service implementation (a protocol) can be replaced with another one, with the same services. Therefore, the OSI reference model allows flexibility regarding the service implementation.

The two lowest layers, that is the physical and data link layers, abstract physical circumstances and offer data transfer between network entities for layer 3. The network layer enables multi-hop connections and connections between different network types. Therefore, it uses a structured addressing scheme (for instance IP addresses). Layer 3 fragments packets to frames and reassembles frames to packets, if necessary. Furthermore, the network layer performs routing based on its addressing scheme.

The transport layer provides data transfer from end-point to end-point. That is, it connects two processes on different hosts. Layer 5 - the session layer - has the task to provide authentication and permission checks for an layer 4 connection between processes. The presentation layer abstracts syntactical differences (for instance different encoding). The application layer contains all programs, that communicate over a network connection. In practice, there is no distinction between layers 5 - 7, each application implements instead the functionality of these layers.

As depicted in Figure 2.3, only the lowest three layers need to be implemented in the network infrastructure (including hubs, switches and routers). End-hosts (including PCs and smartphones) have to implement the entire protocol stack.

2.2.2 Transmission Control Protocol

The *Transmission Control Protocol* (TCP)[8] is a network protocol on the transport layer. TCP offers reliable and ordered delivery - that is TCP assures that all packets arrive at the destination in the designated order. Many internet applications including worldwide web, e-mail and file transfer use TCP for data transmission. Before transferring data, the hosts perform a three way handshake to negotiate the connection parameters, such as the initial sequence numbers. Afterwards, a full-duplex connection is established - data can be transfered in both directions simultaneously on top of a single TCP connection.

2.2.2.1 TCP Header Format

TCP transfer units are called *segment*s. Each segment contains a header and a data section. Figure 2.4 shows the TCP header format. The data section of a TCP segment contains the payload data from the application.

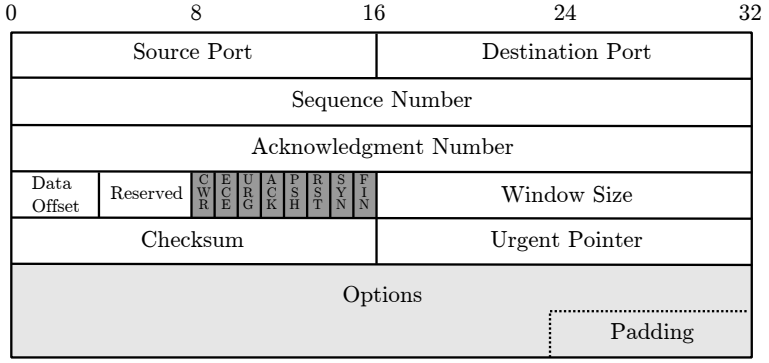

Figure 2.4 TCP header format

TCP uses a fixed set of control bits (marked dark-grey in Figure 2.4) to negotiate and maintain connection settings. In consequence, the header structure (syntax of the protocol) is inflexible. Therefore, the possibility to modify the protocol behaviour (semantic), using this header structure, is limited.

In theory, TCP can be extended using header options (marked light-grey in Figure 2.4). Options are type-length-value encoded: they consist of a fixed-length type field, a fixed-length field representing the length and a value field with variable length. This structure is highly flexible, because options may have variable length and their types are not predetermined. Furthermore, unknown options can be skipped and the rest of the packet can be parsed, because the length of each option is known. TCP options can be used for transfering further control data from one host to another. However, the original TCP specification defined only three options, of which two were needed for option handling[38]. Therefore, only one option (maximum segment size) could be used to transfer control information. Hence, the flexibility through TCP options is not used consequently.

2.2.2.2 History of TCP

The first specification of TCP was published in 1974[9]. Notably, the layered protocol architecture was added in 1981. The transportation functionality was split into TCP[38] and the Internet Protocol (IP)[37]. This fundamental modification entailed an almost complete re-implementation of existing TCP applications. Regardless of this incident, successive implementations of TCP were monolithic, too. With enlargement of the network sizes, performance problems occurred in TCP/IP networks. In particular, network congestion decreased the performance. Therefore, a discussion on congestion control and avoidance started in 1984[32]. During the subsequent years, TCP has encountered numerous improvements. Several algorithms like TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery were developed and standardized[8]. Again existing TCP implementations had to be changed radically. However, the development of new congestion avoidance mechanisms is still object of research[13][6]. So, new mechanisms will occur and must be

integrated into existing TCP implementations. Such an integration is a significant software modification and therefore very complex.

The example of TCP shows that protocols can have a very long lifetime, but nevertheless substantial protocol characteristics may change. This property of protocols should be considered in a protocol implementation design and will be examined in our problem analysis (see Chapter 4).

2.2.3 Host Identity Protocol

The Host Identity Protocol (HIP)[31] is a signaling protocol that provides end-host mobility and security mechanisms such as secure key exchange. The HIP architecture[30] has a highly extensible design. The HIP base specification[31] defines packet and parameter format as well as the basic functionality (see Section 2.2.3.4). Further HIP-related specifications, so called extensions, define additional functionalities (see Section 2.2.3.6). Extensions may define new use cases for existing packet types or parameters. In addition, extensions can define new packet and parameter types. Therefore, the functionality of HIP can be liberally increased. This extensibility is a key characteristic of HIP.

Two end-hosts communicating through HIP, establish a *HIP association*. A HIP association can be considered as signalling channel between these two hosts. Within this signalling channel, secure key exchanges are possible - even over insecure channels, such as public wireless networks. Payload data is not transfered through the HIP association. Instead an existing end-to-end security protocol is used for encryption of payload data. A common scenario is to create secret keying material with HIP and transfer payload data with IPsec and *Encapsulating Security Payload*[24].

2.2.3.1 HIP Namespace and HIP Layer

HIP was developed to resolve the weaknesses of contemporary TCP/IP networks[30]. Classical TCP/IP networks, such as the early Internet, were developed for stationary clients. The network participants use IP addresses for *identification* (each network interface should have a unique IP address) and *localization* (hierarchical routing based on IP address prefix). The increasing mobility of network clients causes issues with the dual usage of IP addresses. When a client changes his point of network attachment from one network to another, he necessarily gets a new IP address to adhere the routing scheme. This assures the correct localization of the client. But, because of the IP address change, the identity of the mobile client changes simultaneously. As a result, connected hosts cannot verify the new identity, without further ado. Opened connections that were bound to the previous IP address, will be disrupted. A new connection establishment is necessary.

Additionally, classical TCP/IP networks have no authentication mechanism on the transport layer, or below. Therefore these networks are susceptible for IP address spoofing and impersonation attacks. These attacks are based on the possibility to send messages with IP addresses that belong to another host. To counteract such vulnerabilities, authentication and encryption must be implemented in applications. The downside of this approach is that each application needs to implement the

security features on its own. In contrast, connection-related security features could be used by multiple applications without additional effort.

In order to eliminate the dual usage of IP addresses, HIP introduces a new cryptographic namespace for the identification and authentication of other network participants. Each communication partner has at least one unique *Host Identity* (HI), that is a self-generated public and private key-pair. Using the HI, each other HIP-enabled host can be identified and authenticated. Today's applications usually use IP addresses for identification and are not HI capable. Therefore, HIP defines *Host Identity Tag*s (HIT) and *Local Scope Identifier*s (LSI). HITs are compatible with IPv6 addresses and hence many applications can use them immediately. Legacy applications without IPv6 support (and therefore without HIT support), use LSIs. LSIs are compatible with IPv4 addresses. Provided with the new HIP namespace, the dual usage of IP addresses can be avoided. Applications on upper layers use the new HIP identifiers. The underlying networks continue using IP addresses for localization and routing.

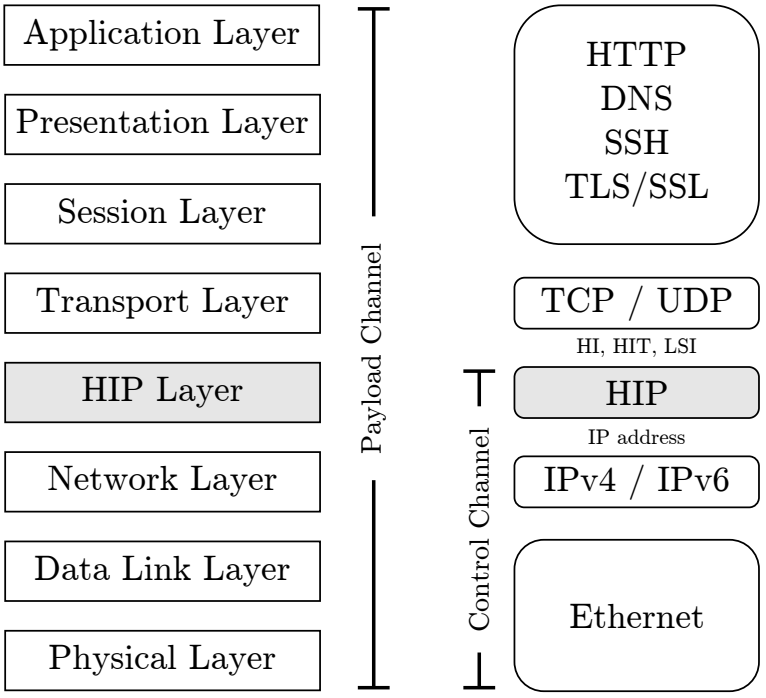

Figure 2.5 HIP in the network stack

The required mapping from HIT/LSI to IP addresses is done on a new protocol layer. This layer serves as indirection point and decouples network and transport layers, as depicted in Figure 2.5. Through this decoupling, end-host mobility becomes possible. IP addresses may change, but the connection, which is bound to the HIT/LSI, remains.

2.2.3.2 HIP Packet Format

Figure 2.6 shows the HIP packet format that is common to all HIP control messages. Each packet must contain several control information (including version, checksum and controls fields) beside the sender's and receiver's HIT. HIP parameters are optional and have variable length. The different control message types are encoded in

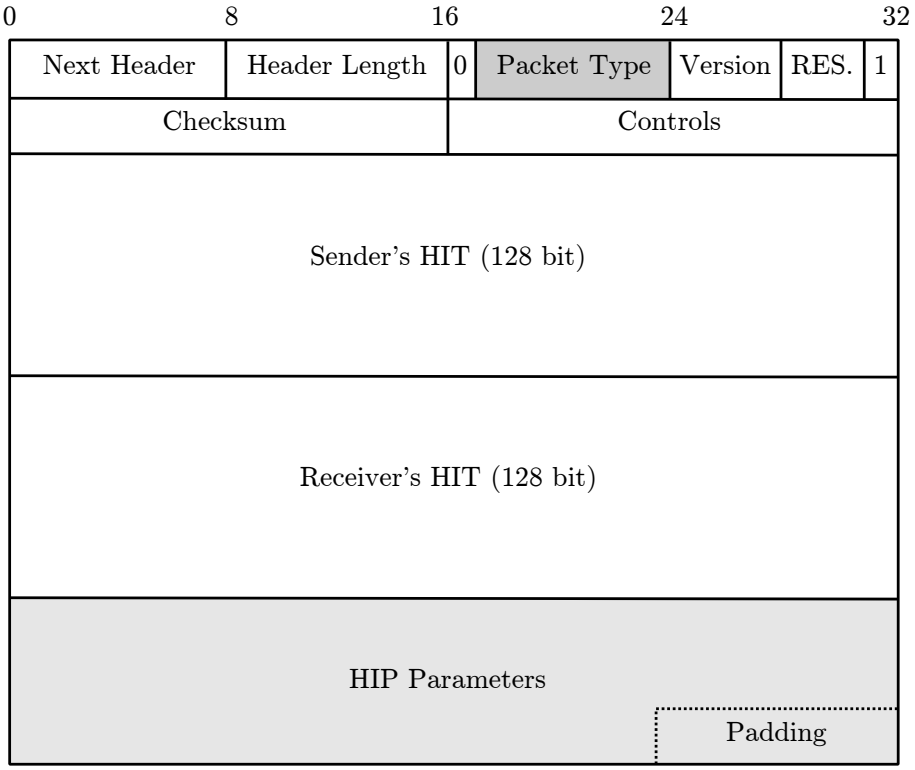

Figure 2.6 HIP packet format

the 7-bit packet type parameter (marked dark-grey in Figure 2.6). Flexibility was one of the main design goals for the HIP packet format. This can be seen in the variability of packet type and parameter fields and due to their flexibility, HIP is greatly extensible. Extensions may define additional packet types and so enlarge the functionality of HIP. However, a HIP implementation must drop control messages with unknown packet types. This allows interoperability between protocol implementations with different feature sets.

2.2.3.3 HIP Parameter Format

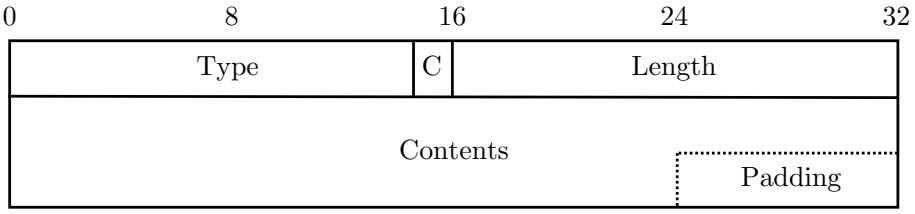

Figure 2.7 HIP parameter format

HIP parameters are type-length-value encoded and have to occur in ascending order of their type in HIP control packets. The last bit of the type code (C, also *critical bit*) denotes whether the parameter must be recognized by the recipient or not. For this reason, critical parameters have odd values and optional parameters have even values. The critical bit facilitates backward compatibility. A HIP implementation that does not understand a non-critical parameter can ignore it, this is because the type-length-value encoding allows to ignore one parameter and continue parsing the remaining packet. Using non-critical parameters, HIP hosts negotiate the feature

set for a HIP connection. Due to the flexibility of type-length-value encoding, HIP parameters can be used for numerous different purposes. Just like packet types, HIP extensions may define new parameter types.

2.2.3.4 HIP Base Exchange

HIP connections are established using a four way handshake - called the HIP base exchange[31] (BEX). The BEX is depicted in Figure 2.8 and contains an authenticated Diffie-Hellman key exchange[10]. The initiator starts the BEX by sending a minimal I1 packet, without additional parameters, containing its and the receiver's HIT. The responder answers with a signed R1 packet that contains his HI, a public Diffie-Hellman key and a *puzzle challenge* that the initiator must solve. The puzzle challenge is a number I, chosen by the responder. The initiator chooses a number J and computes the hash value of the concatenation I, HITs, and J. The challenge is solved, if the resulting hash value has zeros at the lowest order K bits. The parameter K is also chosen by the responder and denotes the puzzle difficulty. The higher K is, the more computing power is needed for solving the puzzle. The initiator sends the solution - the number J - together with his public Diffie-Hellman key in the signed I2 packet to the responder. The responder verifies the solution by performing the hash computation. If the puzzle solution is correct, the responder completes the BEX by sending an R2 packet with his signature. Due to the puzzle challenge, a potential attacker has to invest more computation power than the responder. The puzzle mechanism is necessary to avoid malicious hosts attacking hosts using the cryptography. After a successful BEX, the participating hosts are mutually authenticated and have generated secret keying material that can be used for encryption of payload data.

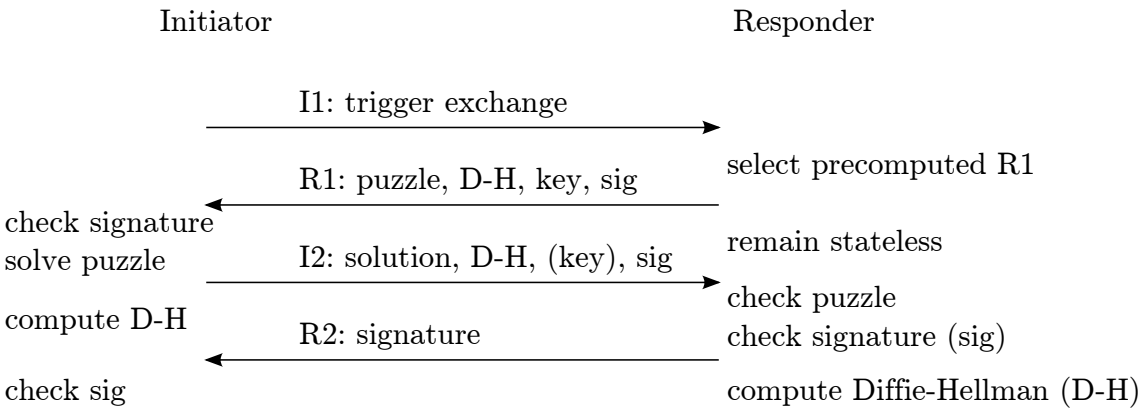

Figure 2.8 HIP base exchange

Beside authentication and key exchange, the BEX can be used for additional features. In which case, the used control packets may contain additional parameters. HIP extensions can define syntax and semantic of these parameters. Thus, the BEX is a multi-purpose operation.

2.2.3.5 Other HIP Control Packets

Two hosts use the HIP base exchange, consisting of I1, R1, I2, R2 packets, for connection establishment. In addition, HIP specifies further control packets - for instance, to maintain or close a connection[31].

UPDATE Packet

HIP UPDATE packets can be used for updating of connection parameters. There are several reasons for sending an UPDATE packet, including rekeying, creating of new security associations and mobility management (information about an IP address change). Authenticity and data integration are secured in UPDATE packets to avoid connection hijacking. In addition, sequence and acknowledgement numbers are used to enable reliable transmission.

NOTIFY Packet

HIP specifies NOTIFY packets for information purpose. Protocol or negotiation errors can be reported using NOTIFY packets. The implementation of NOTIFY packet handling is optional.

CLOSE Packet

HIP connections are teared down using CLOSE and CLOSE ACK messages. The host wanting to close the connection sends a CLOSE packet. This packet is also protected against data manipulation and eavesdropping. The corresponding host confirms the connection termination by sending an equally protected CLOSE ACK packet. Afterwards both hosts can remove the connection state.

2.2.3.6 End-Host Mobility with HIP

One actively used extension for HIP is *End-Host Mobility and Multihoming with the Host Identity Protocol*[33]. This extension enables address changes for established HIP associations (end-host mobility) and the possibility to hold multiple addresses per host (multihoming). Therefore, the term *locator* is introduced. Locator denotes a name for one or more network addresses (points of network attachment). In addition, LOCATOR is a new HIP parameter type that contains zero or more locator fields[33]. This new parameter type is transfered using UPDATE packets. The main objective of the mobility and multihoming extension is to inform associated hosts about new or changed locators. This feature allows a mobile host to change its point of network attachment without losing its transport layer connections. Layer 4 connections exclusively use HITs or LSIs for addressing. Therefore, the IP address change on layer 3 is opaque for transport layer connections. Instead, the HIP implementation changes the mapping between HIT/LSI and IP address.

In the simplest mobility case, one host changes its point of network attachment and rekeying is not performed. Figure 2.9 shows this procedure consisting of three steps.

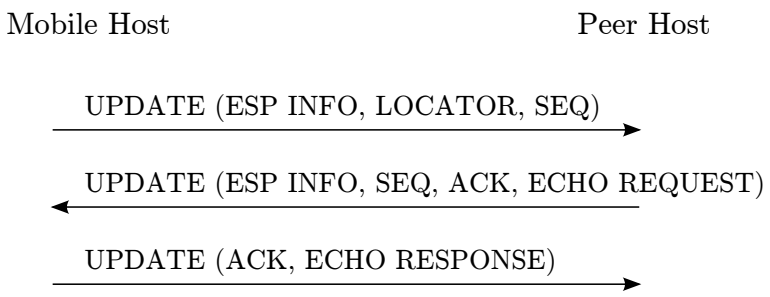

Figure 2.9 Mobility

Firstly, the mobile host informs the peer host about his new locator. In this simple case, the ESP INFO field does not contain new information. The peer host answers with an ECHO REQUEST that is send to the new locator. Finally, the mobile host confirms its new address by sending the ECHO RESPONSE. Sequence numbers (SEQ) and acknowledgments (ACK) make sure that all messages are received and the information is up to date.

The end-host mobility and multihoming extension proves the extensibility of HIP. The new parameter type LOCATOR is transfered using the existent packet type UPDATE. Furthermore, the extension specifies the semantics of the new parameter - that is how to handle LOCATOR parameters. HIP implementations without the end-host mobility and multihoming extension can ignore the LOCATOR parameters in received UPDATE packets. In this way, backward compatibility is warranted.

3

Related Work

In this chapter we present other approaches that share various aspects of our work, such as software modularity or the usage of design patterns. Firstly, we introduce approaches on modular protocol stacks that also apply the software modularity concept to protocol implementations - but at a more abstract level than in our approach. These approaches are presented in Section 3.1. Later, we present studies on software re-engineering. The presented work concerns design patterns, as well as our approach, but in a different domain. However, we present the results as indication for our work in Section 3.2. Finally, in Section 3.3, we introduce practical programming frameworks that may be used for realizing modular networking software.

3.1 Modular Protocol Stacks

The layered architecture of network protocol stacks is a well-established structure. The durability of this design is based on its flexibility with regards to protocol implementation. Each protocol on any layer can be replaced by another one, providing the same interfaces to the adjacent layers. However, one drawback of the strict layering is the isolation of each layer. Independent protocol implementations may realize similar functionalities redundantly. Thus, reusability potentials may not be utilized. Furthermore, a protocol instance can only interact with adjacent protocols. Thus, valuable communication between two non-adjacent layers is prohibited. For instance, TCP retransmissions could be reduced if the physical layer informs the transport layer about network disconnections.

In contrast to our work, *modular protocol stack* approaches do not aim to improve a specific protocol implementation, but to improve the entire protocol stack. Both approaches utilize similar concepts, including software modularity and design patterns. The key idea of modular protocol stacks is to resolve the strict layering and implement the overall protocol stack functionality in a modular fashion. Therefore, these approaches design reusable components and implement the entire protocol stack functionality in one modular software system.

3.1.1 Conduits+

Conduits+ is an object-oriented framework for network protocol software[21]. Its goal is to provide protocol independent, reusable software components for network protocol implementations. These building blocks are called *conduits* and realize the protocol logic. In contrast, *information chunks* represent the processed information, such as packet data. Conduits+ defines four abstract building blocks: *mux*, *protocol*, *conduit factory* and *adapter*. A mux connects multiple other conduits and (de-) multiplexes information chunks based on the respective addressing scheme. Protocol conduits provide the actual protocol functionality by producing, consuming and verifying information chunks. Furthermore, protocol conduits store the current communication state. existing ones. In contrast, adapter conduits connect the protocol stack with the Conduit factories are used for creating new conduits and connecting them to environment - that is the underlying hardware and the applications on top. Since implementations using Conduits+ aim to provide the entire protocol stack functionality, the hardware and the application adapter are the only external interfaces.

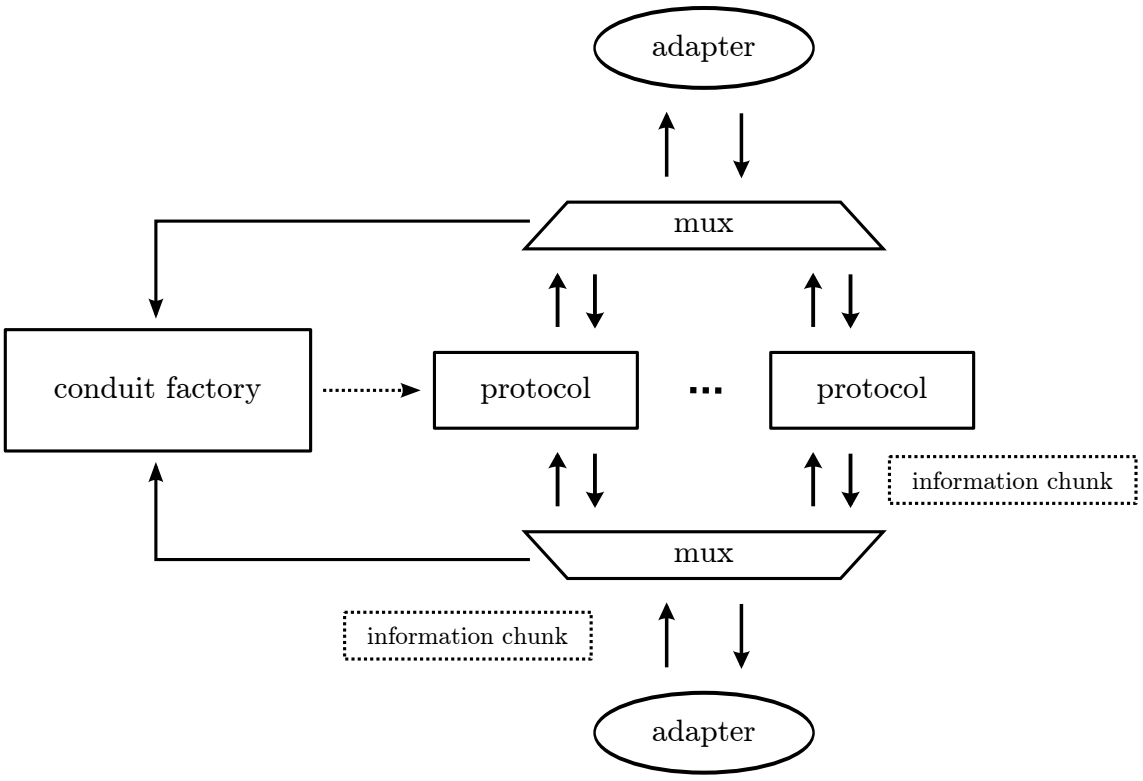

Figure 3.1 A typical Conduits+ structure according to [21]

In order to achieve the desired functionality, multiple conduits are interconnected. The resulting system processes protocol data in the form of information chunks and performs the designated activities. Figure 3.1 shows a typical system consisting of multiple conduits. As mentioned above the adapter conduits act as interface to the environment. While the lower adapter in Figure 3.1 could be an Ethernet conduit that operates hardware, the upper one could be a network socket that can be used by applications. The other conduits implement protocol specific logic. Both mux conduits are responsible for (de-) multiplexing the processed information chunks

from and to the various protocol instances. Each protocol instance manages one connection. If an information chunk without matching protocol instance is received, the respective mux informs the conduit factory, which can create a new protocol instance if necessary. For instance, a Conduits+ TCP layer implementation could have the structure depicted in Figure 3.1.

Conduits+ provides generic classes for protocol specific functionalities - that are abstract conduits. Early versions of the framework enforced the usage of inheritance in order to implement the designated protocol functionality. The general protocol class must be specialized by a concrete class fulfilling the protocol needs. For this reason, developers must know about the concrete framework implementation, which causes additional effort. This issue was solved by the application of design patterns, such as adapter and command[21]. Hence, the reusability of all framework components was increased and the usage of inheritance reduced to a minimum.

Conduits+ is implemented in C++ and therefore restricted to platforms for that a C++ compiler exists. The framework has been used for implementation of a TCP/IP stack and a multi-protocol *Asynchronous Transfer Mode* access switch. These implementations prove the practical usability of Conduits+. In our opinion, the framework brings the most benefits when the entire protocol stack is implemented using conduits, seeing as general building blocks can be reused in multiple protocols.

3.1.2 Protocol Factory

Another approach to modular protocol stacks is *Protocol Factory*[28] (ProFab). ProFab consists of two main parts: a generic protocol library and a virtual platform for hardware abstraction. The objective of ProFab is to enable reuse in two dimensions. Firstly, across platforms by introducing the a hardware abstraction layer. Secondly, across protocol layers by identifying modular building blocks that can be utilized in multiple implementations.

Generally, protocols are first verified using simulators with specific characteristics, such as special programming concepts or languages. The source code generated for simulations cannot be reused directly for implementing the end-system protocols. The end-system implementation must instead start from scratch or the simulator code must be ported. Using ProFab's virtual platform, developers can build protocols for multiple platforms including simulation environments, testbeds and classical operating systems - with the same code base. This directly improves the development process of protocols: Firstly, protocol design and implementation can be verified using simulations and testbeds. This phase could contain several iterations of design, implementation and tests. If the results are satisfying, the same code can be used to build binaries for numerous operating systems including Linux, Windows XP / CE / Mobile, network simulators, testbeds and sensor nodes.

ProFab's generic protocol library has similarities to the Conduits+ framework for network protocol software. Both approaches utilize the same concept of software modularity in order to enable software reuse, but on different abstraction levels. ProFab identifies *invariants* that can be used in multiple protocols. Invariants are building blocks with a dedicated functionality. While Conduits+ specifies four abstract types for its building blocks, ProFab offers a higher flexibility on a more

fine-grained level. ProFab does not limit the type of invariants. Instead all common functionality among different protocols may be identified and realized as invariant. An implementation of an invariant should be as general as possible. For instance, check-summing is required by multiple protocols on different layers. Therefore, it is implemented as invariant and can be used multiple times. Each protocol that needs to compute or compare checksums, can use this one implementation. Well-defined invariants enable reusability across protocols and network layers and hence simplify protocol development.

Besides software reuse, ProFab enables dynamic inside the protocol stack. Required protocol functionality can be loaded and connected on demand. This allows the realization of different functionalities for multiple connections. For instance, a TCP implementation could apply different congestion avoidance algorithms for multiple TCP connections by loading different invariants for this task. This dynamic is achieved at the expense of performance loss. The dynamic loading causes additional effort compared to static approaches.

ProFab extends the standard C syntax by adding constructs for module implementation and composition. Hence, ProFab enables developers to build modular protocol software in C. However, ProFab does not introduce a new programming language, instead ProFab uses a pre-compiler for the generation of plain C code. Thus, the generated protocols can be used on all platforms that provide a C compiler and were added to ProFab's virtual platform. Furthermore, the ProFab compiler significantly reduces the overhead introduced by the dynamic loading of protocol functionality. This is done by identifying dynamic functionalities that can be inlined and integrating these into a single building block or by replacing function pointers by direct calls.

Overall ProFab simplifies the protocol development by reducing the complexity of protocol stacks and enabling reuse of building blocks. The virtual platform for hardware abstraction allows deployment and evaluation on many different platforms.

3.2 Software Re-engineering

The functionality of software does not change, until its implementation is modified. Therefore, software does not abrade in the classical sense. But, environments software is commonly used in change. In this case, the software in question becomes less usable. In order to counter this software aging process, software can be modified. Most likely, these functional modifications were not considered during the initial design. Therefore, the current implementation does not map the original design. This fact complicates the software maintenance and this kind of software is called *legacy software*. Thus, *software re-engineering* might be necessary in order to increase the software's maintainability. Software re-engineering includes all activities to make software more understandable or changeable[39]. In general, the first step of re-engineering is the analysis of the current implementation in order to create an abstract design. Afterwards, the design can be changed to fulfill the new requirements. Finally, the new design is implemented using current software approaches. Software re-engineering with the application of modularity and thus the goal to create modular software, is called *modularization*.

Verma and Liu[43] published a case study on software re-engineering with the focus on application of design patterns. Although their work is related to software for mesh generation, the results of the case study are of interest for our thesis. Verma and Liu motivate software re-engineering by presenting the disadvantages of legacy software. Besides the hindered maintenance, they identify the absence of abstraction as a main drawback, because abstraction could reduce the system's complexity to the developer. Furthermore, the study names conditional statements, such as `if-then-else` or `switch`, as indicators for inflexible source code. Since static constructs offer only a fixed set of alternatives, their extension requires significant effort.

The case study points out that the application of design patterns increases flexibility, extensibility and understandability of a software system. In particular, Verma and Liu say that the usage of design patterns entails modularization and hence the resulting software is easier to extend. Regarding the performance of the modified system, the study provides only vague statements. It states that the performance degradation should be acceptable to users. Overall, the study concludes that the effort for the application of design patterns is exceeded by the quality improvement of the resulting system in their scenario.

3.3 Programming Frameworks

In practice, more and more programming frameworks for different domains appear. Generally, frameworks reduce the development effort by providing reusable building blocks. This section introduces frameworks that may be used for realization of modular networking software. Firstly, we present a general plug-in framework, called *C-Pluff* [29]. This framework can be used to build software that allows the dynamic loading of functionality in form of plug-ins. Secondly, we introduce the event notification library *libevent*[40]. The library provides the possibility to register callback functions to events and performs the designated functionalities when the event occurs.

3.3.1 C-Pluff

C-Pluff is a framework for realizing plug-in based software using the C programming language[29]. The objective of C-Pluff is to provide services for accessing and managing plug-ins. The framework defines how the main program and plug-ins interact. Programs implemented with C-Pluff typically consist of a thin main program and multiple plug-ins. The main program controls the plug-in framework and is responsible for the initialization and setup of the framework. Plug-ins may be loaded or unloaded during runtime, without restarting application or framework. Notably, the most application logic is found in plug-ins; allowing a high degree of flexibility. So-called *extension points* are the counterpart of plug-ins and define how main program and plug-ins may be extended. Thus, the extensibility is not limited.

Figure 3.2 shows the structure of programs based on C-Pluff. Plug-ins are connected using the framework functionality and can extend the system functionality at multiple extension points. The main program controls the framework, but does not provide the main application logic.

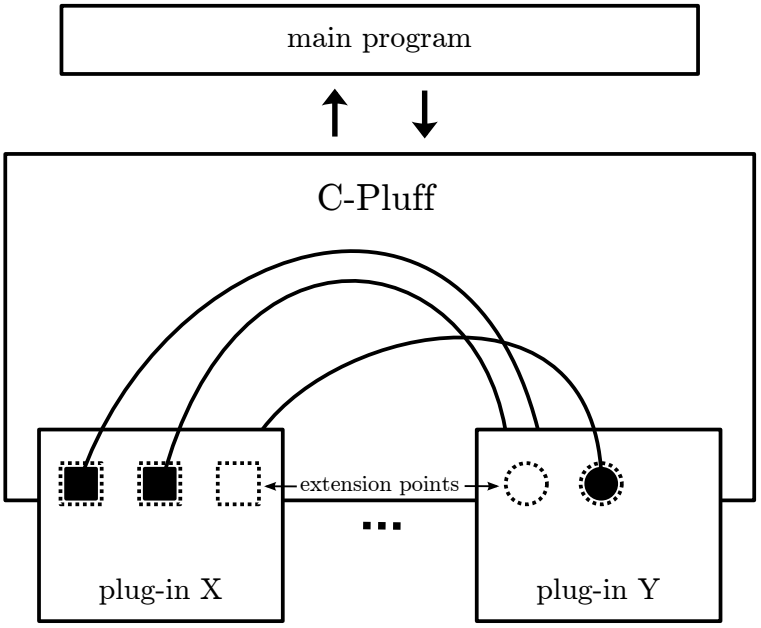

Figure 3.2 Structure of C-Pluff-based software according to [29]

In C-Pluff, a plug-in consists of a plug-in descriptor, a runtime library and static data. The plug-in descriptor provides meta information for the plug-in, such as the plug-in name and its static dependencies. The runtime library is a shared library containing all program logic provided by the plug-in. Each plug-in can provide static data to other plug-ins in form of data files. C-Pluff provides the functionality to load plug-ins during runtime. For that purpose the plug-in runtime libraries are dynamically linked when the plug-in is started, or unloaded when the plug-in is stopped. Hence, plug-ins can be developed, distributed and deployed independently. In order to represent complex relations, plug-ins may depend on other plug-ins. C-Pluff was designed to manage these dependencies, providing functionality to automatically load required plug-ins if applicable.

C-Pluff is completely implemented in C and currently available for Linux and Windows. The framework was created by a single developer and is only marginally maintained. The last release was three years ago[29].

3.3.2 Libevent

Libevent is an asynchronous event notification library for network software[40]. Typical network software, such as a http-server, has to handle multiple connections and hence creates multiple network sockets. Thus, a classical implementation manages multiple file descriptors - one for each socket. In order to observe incoming requests, the software periodically polls all file descriptors. Usually this is done in an infinite loop and the file descriptors are checked in sequence. If the program has to take action on one descriptor, this is done immediately. Therefore, the handling of subsequent file descriptors is delayed until the former activities are finished. This scenario might result in low performance, due to the blocking activities.

Libevent provides another approach. All needed file descriptors are registered to libevent under specification of callback functions. Whenever a file descriptor requires

activity, libevent executes the registered callback function. This handling is done asynchronous - that is the main program must not wait until the callback function has finished its activity. While another thread performs the desired actions, the main program can continue with polling the other file descriptors. Thus, libevent can be used to replace the classical event loops and hence increases the scalability of network software.

Since libevent allows the dynamic addition and removal of file descriptors, the flexibility of the resulting program is increased too. Furthermore, libevent can execute a designated functionality after a timeout has been reached or when the program receives a specific signal.

Libevent provides an application of the publish/subscribe pattern in the context of file descriptors, timeouts and signals. With regards to protocol implementations, libevent can be used to create a flexible packet handling mechanism. The library is implemented in C and available for Linux, *BSD, Mac OS X, Solaris and Windows[40].

4

Problem Analysis

This chapter precisely describes the challenges connected with protocol implementations and in particular, regarding adaptable and flexible protocols. Firstly, Section 4.1 highlights the characteristics of adaptable and flexible protocols. Secondly, Section 4.2 analyses the drawbacks of monolithic protocol implementations. Finally, we describe the problem statement in Section 4.3.

4.1 Adaptable and Flexible Protocols

> If your protocol is successful, it will eventually be used for purposes for which it was never intended, and its users will criticize you for being shortsighted. (Charlie Kaufman, [36])

Due to the fast and unpredictable evolution of computer networks and the applications on top of them, the protocols used must be sophisticated. Backward compatibility is a well-established requirement for new protocols and simplifies the protocol integration into existing networks. Besides this, extensibility and hence *forward compatibility* is becoming an important design goal for protocol specifications. At the time of specification, not all use cases for a protocol might have been known, making the need for changes to the protocol behaviour to be very likely. As a result, specifications of numerous new protocols are highly flexible. A common example is the need for exchangeability of cryptographic algorithms. The defined algorithms may become outdated due to security vulnerabilities. Then an additional specification defines the usage of a different algorithm. For instance, HIP defines RSA and DSA as obligatory, cryptographic algorithms[31]. However, it is intended to use newer algorithms in future specifications, because the present algorithms may become outdated. Therefore, the used algorithms inside a HIP implementation should be exchangeable with as less effort as possible.

4.2 Issues of Monolithic Protocol Implementations

Due to the characteristics of adaptable and flexible protocols, monolithic implementations of these protocols do not make sense. Subsequently, we present the main drawbacks caused by monolithic protocol implementations.

4.2.1 Source Code Dependencies

The examples of TCP and HIP show that protocol implementations need to be changed over time. Either because the specification was revised or because the protocol specification is flexible and extensible per se. In the case of a monolithic protocol implementation, these predictable modifications are extensive. The reason can be found in the strong source code dependencies caused by the loose architecture in monolithic software. Any statement in the code can depend on code at other positions; without having criteria or explicit rules for these dependencies. A dependency might be an expected variable modification. If this modification is missing, the dependent code will behave unexpectedly. For instance, protocol state may be modified at multiple positions. Code depending on one of these state modifications cannot function anymore if the state modification is removed. But, in monolithic software, there is no explicit reference between the related positions and a code modification can entail unexpected errors, making **maintainability** (see Section 2.1.1) of monolithic protocol implementations is poor.

In addition, source code dependencies decrease the **reusability** of code. On the one hand, monolithic functionality is hardly reusable inside a software project, due to the lack of interfaces to it. On the other hand, usage of code blocks from monolithic software in other projects requires the same conditions as in the original project to be valid. Otherwise, significant modifications have to be done. In practice, this results in a so called *copy and paste* approach: the needed code is copied from one project and pasted into another. Afterwards the code is modified to fulfill the exact requirements of the new project. This is code duplication and not reuse. Both code bases must be maintained separately, because they are no longer identical. In contrast, a well-defined component with an explicit interface could be used in multiple projects, but needs to be maintained only once.

4.2.2 Implementation Complexity

Hosts use network protocols for interaction. Therefore, a defective protocol implementation on one host can impact other, unrelated hosts or the entire network. Thus, protocol implementations must be error-free and conform exactly according to their specifications. An incorrect HIP implementation, for instance, may cause security vulnerabilities. Several HIP control packets are signature protected. It is essential that the signature of a received packet is verified, before the further packet processing is done. Otherwise the host would be highly vulnerable. The implementation complexity of monolithic software provokes **implementation errors**. There are no different abstraction levels in monolithic software and a developer has to understand the entire architecture all at once. In big software projects, direct communication

between developers might not be possible due to a separation in time and space. Subsequent developers might not have knowledge about assumptions made by the original developers. However, modifications require an exhaustive knowledge and a vast mental model of the software. This knowledge is essential, because the loose architecture of monolithic software does not point out these assumptions.

4.2.3 Reduced Portability

Beside classical personal computers also servers, middleboxes and smartphones participate in today's networks. Therefore, protocols must be implemented for more and more platforms. The porting of a monolithic implementation to another platform is extensive, because platform specific code might occur anywhere in the code base. In contrast, a layered software architecture would increase the portability. Hardware specific functionality can be realized on one layer and the protocol logic on another, higher one. Porting to another platform only requires changes in the hardware specific layer.

4.2.4 Organizational Issues

Like software development in general, protocol development becomes a more iterative process. Regarding flexible protocols, first the basic functionality could be implemented and tested. In a subsequent iteration additional extensions may be defined, implemented and verified. Monolithic implementations are not qualified for iterative processes, because of their poor maintainability.

Furthermore, the absence of components complicates the separation of responsibilities between software developers or different organizations. Due to the wide-ranging dependencies in monolithic software, the code base cannot be separated into multiple, *independent* parts. A separation without considering the dependencies, would also cause issues. Let us assume each developer is responsible for a specified part of the code base. The work on one of these parts will entail changes in other parts of the software. Hence, the responsible developer must be involved in order to perform the modifications. This is an extensive organizational effort.

The usage of a version control system with the possibility to independently work on the same code base in different branches, might be an approach to reduce this problem. Each developer can perform the required modification in his branch. But the synchronization of multiple branches will be very extensive.

4.3 Problem Statement

The loose architecture of monolithic software entails multiple issues. Namely, the wide-ranging and implicit source code dependencies reduce maintainability and reusability of monolithic software. Furthermore, the complexity of monolithic protocol implementations provokes implementation errors. Reduced portability and organizational issues complete the list of drawbacks for monolithic protocol implementations.

These issues are not sustainable especially for adaptable and flexible protocols. Our approach (see Chapter 5) explores the applicability of modular structures on the design and implementation of adaptable and flexible protocols. In addition, we evaluate the application of design patterns in order to increase clearness and reusability of the resulting software.

5

Design

The problem statement in Section 4.3 shows that monolithic implementations of adaptable and flexible protocols are not sustainable. Therefore, our design bases on modular structures and approved design patterns. In particular, flexible protocols specify mandatory and optional features. The key idea is to implement all mandatory protocol functionality in a minimal program core. Optional features of a protocol specification are realized in so-called *extensions* or *modules*. We will use these two terms interchangeably. Each extension encapsulates a self-contained set of optional features. In order to minimize dependencies between extensions and the program core, all components are designed as self-contained building blocks. Interaction of components is realized using the publish/subscribe and command design patterns.

This chapter consist of three sections. First, we define our design goals in Section 5.1. Afterwards, in Section 5.3, we describe how a modular protocol design can be realized. Finally, we prove our modular design by applying it to HIP in Section 5.4.

5.1 Design Goals

Our problem analysis in Chapter 4 identifies multiple challenges regarding the design and implementation of adaptable and flexible protocols. With the solution described in this chapter, we aim to fulfill these challenges and hence achieve the following design goals:

- **Maintainability** of the resulting software. Maintainability includes modifiability and extensibility at the design and implementation levels as well as testability of the runtime correctness. In particular, modifiability and extensibility are important for the implementation of flexible protocols, because the specification of these protocols might change frequently.
- **Reusability** of building-blocks. Reuse of self-contained components reduces the implementation complexity and accelerates the development process. The

functionality of existing components may be integrated, instead of implemented from scratch. For this reason, we aim to design the needed functionality in a reusable shape. The generated building-blocks should be reusable within the same project and, if applicable, in other projects.

- **Simplicity and Structuredness** of the software project and source code. Simplicity means that gratuitous complexity should be avoided. Therefore, we do not aim to create an solution that contains unneeded features, but one that fulfills our requirements. In addition, source code simplicity directly reduces implementation errors, because developers understand the code faster. Structuredness denotes the existence of responsibilities on the project level and the separation of concerns on the source code level. The overall structure must allow the simultaneous co-operation of multiple developers on different extensions, without conflicts. Thus, we aim for a solution that allows indepent development of different functionalities and enables their integration to the overall system.

- **Performance** of the resulting application is an important attribute, because protocol implementations with low performance also influence the performance of dependent software. Meaning that, the performance of a modular protocol implementation must not be significantly worse than the performance of a monolithic implementation. This goal should also be fulfilled for embedded devices with low computing power, such as routers or smartphones.

We present a design that considers these goals in this Section 5.3 and evaluate its conformance in Chapter 7.

5.2 Applicability of Existing Approaches

Approaches like Conduits+ and ProFab (see Section 3.1) propose a modular concept for entire protocol stacks. This enables a high potential for reuse and may increase modifiability and extensibility of the protocol stack. The authors of both frameworks, Conduits+[21] and ProFab[28], proved that the implementation of protocol software using their respective approaches improves the protocol development. For instance, due to the reuse of existing components or the application of software modularity. We agree that both frameworks improve the implementation of network software from scratch - that is without existing code. But, considering a situation with an existing implementation that should be re-engineered, the application of both frameworks is challenging, because the frameworks strictly define the software structure. Thus, the existing source code has to be transformed in order to fulfill the framework's structure. During this process, there are no stages with functioning interim versions. Instead, the entire code base must be integrated into the framework.

Furthermore, the mentioned frameworks also reduce flexibility at the implementation level, because they predetermine the software structure and hence the control flow of the resulting program. This situation is not problematic, as long as the framework structure maps the requirements exactly. But if framework structure and the

actual protocol design diverge, this demands high effort to compensate. For this reason, we argue that of adaptable and flexible protocols should not be implemented using frameworks that predetermine the control flow, because the flexibility of the considered protocols could conflict with the structural heaviness of frameworks. In contrast to frameworks, software libraries provide independently usable functionalities. Developers may utilize only the actual needed features and hence decide on using a subset of the functionality that is offered by the library.

We appreciate the mentioned solutions as being reasonable for the implementation of the entire protocol stack functionality from scratch, but our problem corresponds to a single protocol instance on one layer. So, our approach for modular protocol implementations will not utilize the mentioned frameworks.

5.3 libmod - a Protocol Modularization Library

In this section we describe our concepts for modular protocol implementations. Firstly, we examine the practicability of existing approaches to our problem. Afterwards, we give an overview of our solution before we explain our concept in detail.

5.3.1 Architecture Overview

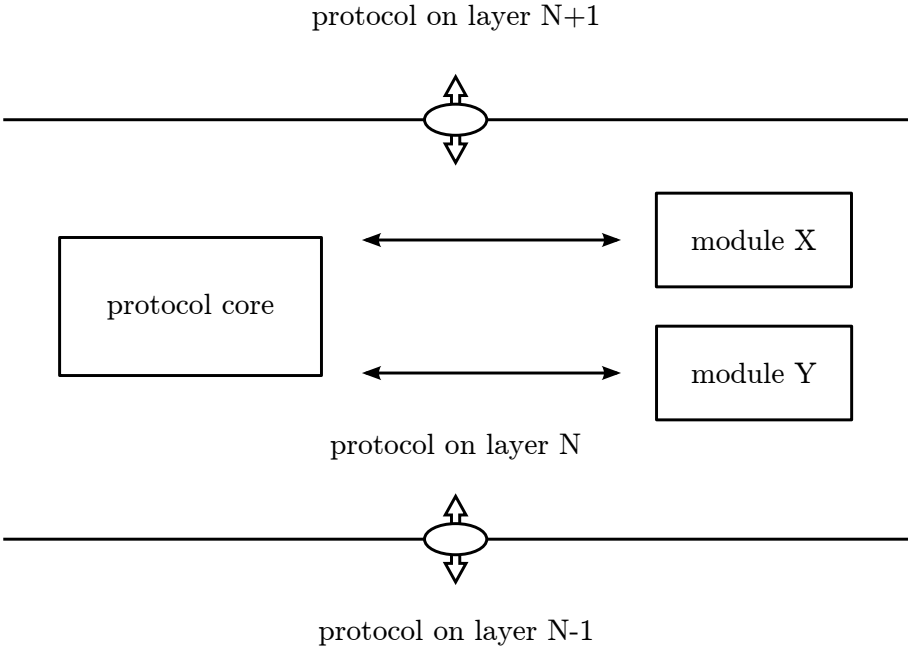

Figure 5.1 Modular protocol structure on layer N with interfaces to layer $N+1$ and $N-1$

Based on the above argumentation, we designed a modular architecture that resides on a single layer. Figure 5.1 depicts the considered environment. We want to create a modular solution for one protocol on a specific layer. As usual in layered protocol stacks, protocols on different layers communicate through the defined

interfaces between their layers. Within the regarded protocol, the overall functionality is split into multiple components; namely protocol core and multiple modules. Modules interact directly with the core and the protocol functionality is provided by collaboration of protocol core and the modules. The ultimate design goal is to minimize dependencies between each of these components. This directly facilitates maintainability and reusability.

Adaptable and flexible protocols usually consist of mandatory and optional features that are defined in multiple specifications. Our design subdivides the features according to their obligation: Mandatory features, such as security checks, are implemented in the protocol core and optional features, such as additional features for individual cases, are encapsulated into modular extensions. Thereby, the minimal coupling and maximal cohesion principle applies. Minimal coupling denotes that each module should have as few external dependencies as possible. In contrast, all features inside one module should be semantically related and may have internal dependencies at the source code level.

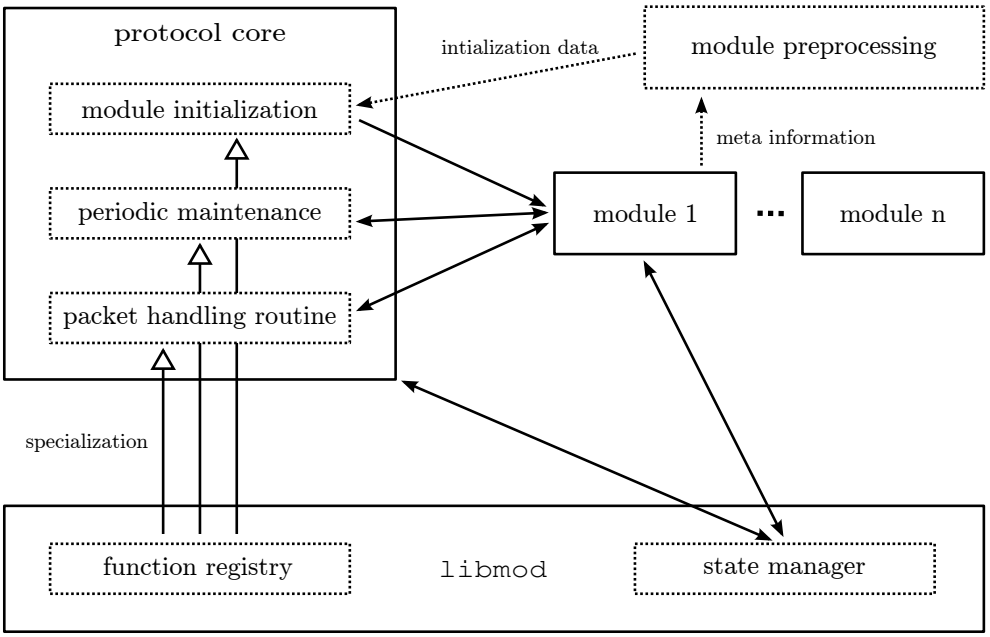

Figure 5.2 Design overview (for clarity reasons only module 1 is connected)

Each module should thereby be a self-contained building block that can be added or removed to an existing protocol implementation without enforcing changes to the protocol core. This is only feasible, if the protocol core does not know how many or which modules exist. Therefore, we need a mechanism that integrates protocol core and modules. The integration mechanisms will be encapsulated in a reusable modularization library called `libmod`. This library provides general functionality for modular protocol implementations, which may be specialized to fulfill the actual needs. Figure 5.2 shows the involved components and their relationship. Modules are connected to the protocol core using functionalities from `libmod`, the protocol core as well as the modules depending on the modularization library.

Since modules may need to save state information that are not provided by the protocol core, they must be enabled to create own state variables. In order to realize this feature without creating redundant functionality nor dependencies, we define

a *modular state* mechanism. Protocol core and modules can use this mechanism to register and manage their state information. As mentioned above, the core is not aware of the existing modules, because we aim to create independent building blocks. The protocol core can then not call any module functionality directly. Instead `libmod` provides a *function registry* that mediates between protocol core and modules. The function registry is a general concept that allows the nomination of functions, which are executed when a specified event occurs. In order to utilize this concept, the core must specialize it. We propose three features that are based on the function registry. These features are *module initialization*, *packet handling routine* and *periodic maintenance*. In our opinion, these are the three fundamental parts, which might be influenced by modules. Therefore, modules are enabled to register their functionality.

With the program startup, the protocol core obtains exclusive control of the program flow. Since the protocol core is not aware of the existing modules, it cannot delegate the program flow to the modules and hence these could not register any functionality. Therefore, we need a mechanism that enables modules to register their functionality. We propose a *module preprocessing* routine that analyzes meta information provided by the modules and makes the required information available to the core. Furthermore, the preprocessing is responsible for checking the compatibility of protocol core and existent modules.

After this first overview, the next sections provide detailed information about the mentioned components.

5.3.2 Modular State

Especially protocols that perform a connection establishment save various association state information, such as the used address pairs or connection timeouts. The protocol core can manage this information for all mandatory functionalities. However, extensions may need to save additional information, such as optional connection options, in the association state. The protocol core cannot manage the extensions' state information, because that would entail dependencies between protocol core and modules. Alternatively, each extension could create and maintain its own state information set per host association. Then, the same functionality of creating and updating data structures as well as look-up mechanisms would be implemented and executed redundantly in core and modules.

For this reason, the data structure for storing the association state must be accessible for the protocol core and the modules. We propose a modular state structure that enables the core as well as extensions to add and modify state information in one central database.

Figure 5.3 depicts the proposed structure. The *state manager* administrates the association state. Protocol core and extensions can register new state items, that are individual state data structures, during their respective initialization. In addition, they are allowed to read and write the existing state entries. State items are clearly identified by their unique ID - that is a string identifier. IDs are chosen by the registrar (core or module) while needed to access state items. Further protection against unauthorized read or write is not intended, because core and modules may

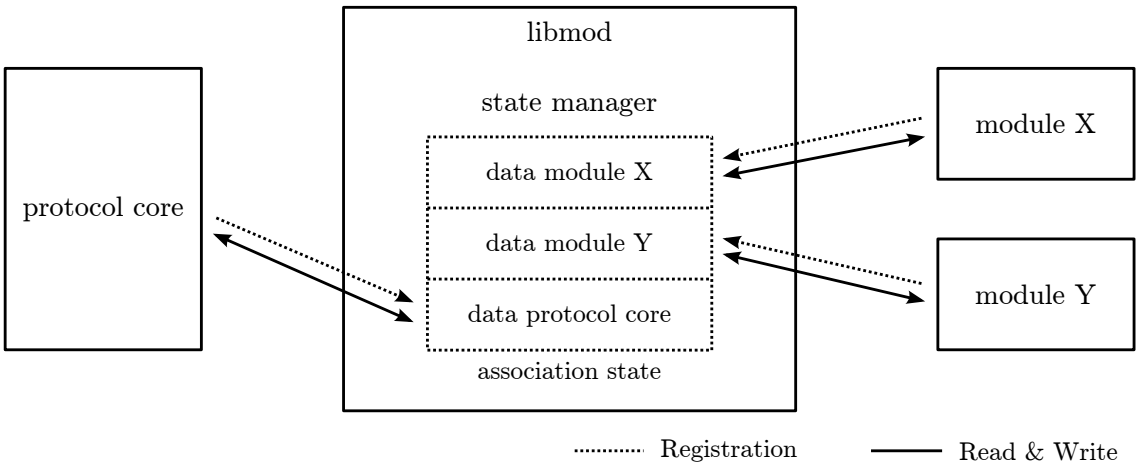

Figure 5.3 Modular association state structure

need to access data fields of other components. In this case a dependency between the two components exists and the concerned functionalities must take this dependency into account.

Due to the registration of state items, the state manager knows the structure of new association state database entries. However, the state manager does not know how to initialize the data structures for a new entry. Thus, each component that registers a state item also provides an initialization command for this item. Hence, the state manager can preallocate the state items with the desired data.

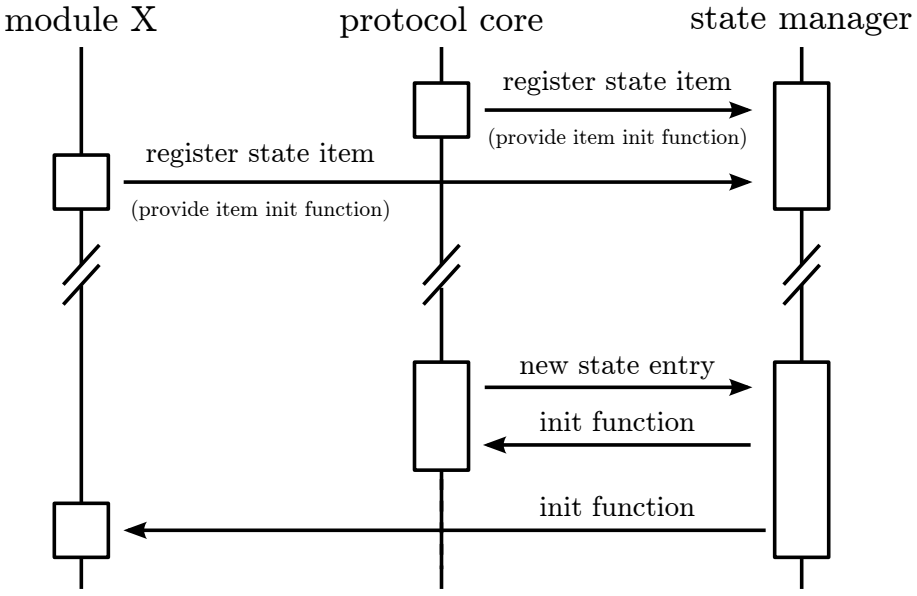

Figure 5.4 Modular state initialization

As shown in Figure 5.4, modules provide a state initialization function when registering their state item. The state manager saves all state items and the respective initialization functions. When the protocol core adds a new entry to the state association database, state items are initialized with the registered function. This is again an application of the command pattern. The state item initialization functions are

requests in the terms of the command pattern. Modules and protocol core can be considered as receivers and invoker.

5.3.3 Function Registry

Naturally, the protocol core defines the program flow during runtime. But the core has no awareness about which or how many modules are existent nor how to execute the functionality that resides in the modules. This blindness is necessary to avoid dependencies between protocol core and modules, because otherwise semantically independent components would be coupled at the implementation level. Therefore, we need an alternate integration mechanism for the module functionality. That is, the modules must be enabled to influence the control flow. For this purpose, `libmod` provides an abstract function registration mechanism. Conceptually, the function registry is an application of the publish/subscribe and command design patterns.

Each component (protocol core or modules) may subscribe to the function registry by registration to an specified event. Every time a defined event occurs, all registered components are sequentially notified and hence enabled to execute their desired functionalities. This event notification service is a realization of the publish/subscribe pattern. Additionally, it includes a command pattern structure. All registered functionalities must conform to the same request type - that is they must fit to the same interface. The component, which has registered the functionality, acts as receiver and will perform the request, when it is invoked. This trigger ultimately originates in the program core, which acts as invoker.

The function registry is a general concept for decoupling protocol core and modules. In order to utilize this concept as concrete feature, it must be specialized. Our design applies the function registry concept to multiple functionalities; including the packet handling routine (Section 5.3.3.1) and periodic maintenance (Section 5.3.3.2). Furthermore, the modular state initialization concept (see Section 5.3.2) could be realized using the function registry.

The mentioned functionalities are protocol dependent and must implemented in the protocol core. The next sections describe how these functionalities can be realized using the generic function registry feature from `libmod`. Thus, the function registry is a central element for the decoupling of protocol core and modules, because without this mechanism, functionality that is implemented in modules could not be executed.

5.3.3.1 Protocol Packet Handling

A central aspect of each protocol implementation is the processing of received packets. This packet handling mechanism is responsible for the packet demultiplexing - that is the mapping of received packets to the respective connection. The packet demultiplexing mechanism triggers the further protocol functionality, based on the received packet and the current connection state.

In particular regarding adaptable and flexible protocols, new functionality will be added over time. For instance, new packet types and their handling could be defined in additional specifications. Furthermore, the semantic of existing packet types

might change. In order to avoid numerous, heavy modifications to the packet handling routine, it should be as flexible as possible.

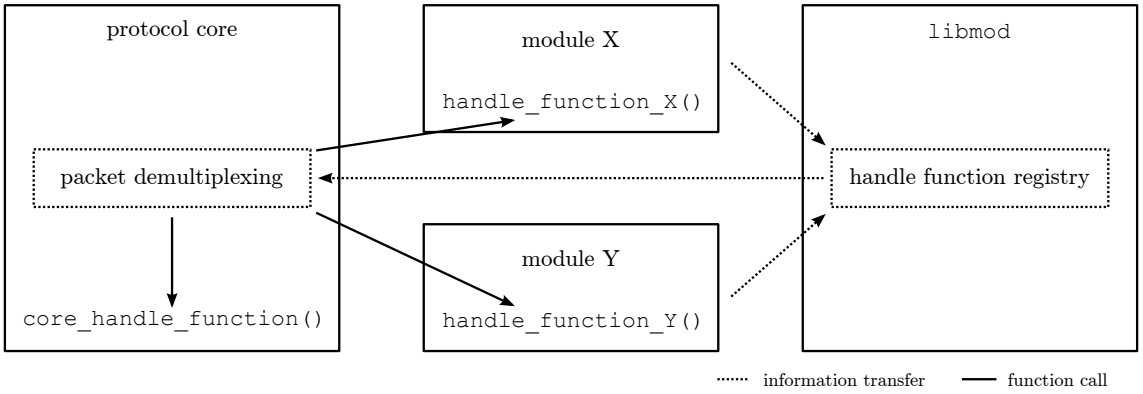

Figure 5.5 Modular packet handling

Figure 5.5 shows the structure of our approach. Protocol core and modules can register handle functions for a combination of packet type and current connection state. Since protocol core and modules may register multiple handle functions for one combination of packet type and connection state, the execution order must be specified. Therefore, we use an absolute priority number to order handle functions: the lower the priority number the earlier a handle function is executed. Priority numbers are positive integers and mandatory for all handle functions.

Protocol core and modules register handle functions with the *handle function registry*, which is a specialization of the general function registry. Based on this information, the packet demultiplexing routine executes the designated functions in the required order. This method facilitates a flexible packet handling: extensions can define handle functions for existing or new packet types without any modification in the protocol core.

Furthermore, extensions may need to change the existing packet handling mechanism. For instance, because a security functionality becomes obsolete. Thus, our design offers the possibility to deregister handle functions. Using this feature, a module can replace existing handle functionality by another one.

Regarded over time, the described process consist of two phases. Firstly, during the initialization phase, protocol core and modules register their handle functions as depicted in Figure 5.6. Secondly, each time the protocol core receives a packet, the handle function registry executes the previously registered handle functions. The execution order depends exclusively on the priority, with which the handles function were registered.

Handle Function Priority Ranges

Since independent modules can register handle functions for the same combination of packet type and connection state, a developer will not definitely know, which handle functions exist for one parameter combination. Furthermore, some handle functions must not be executed until a specific task is fulfilled. For instance, parameter parsing should not be done until all security functions are finished. Thus, the developer

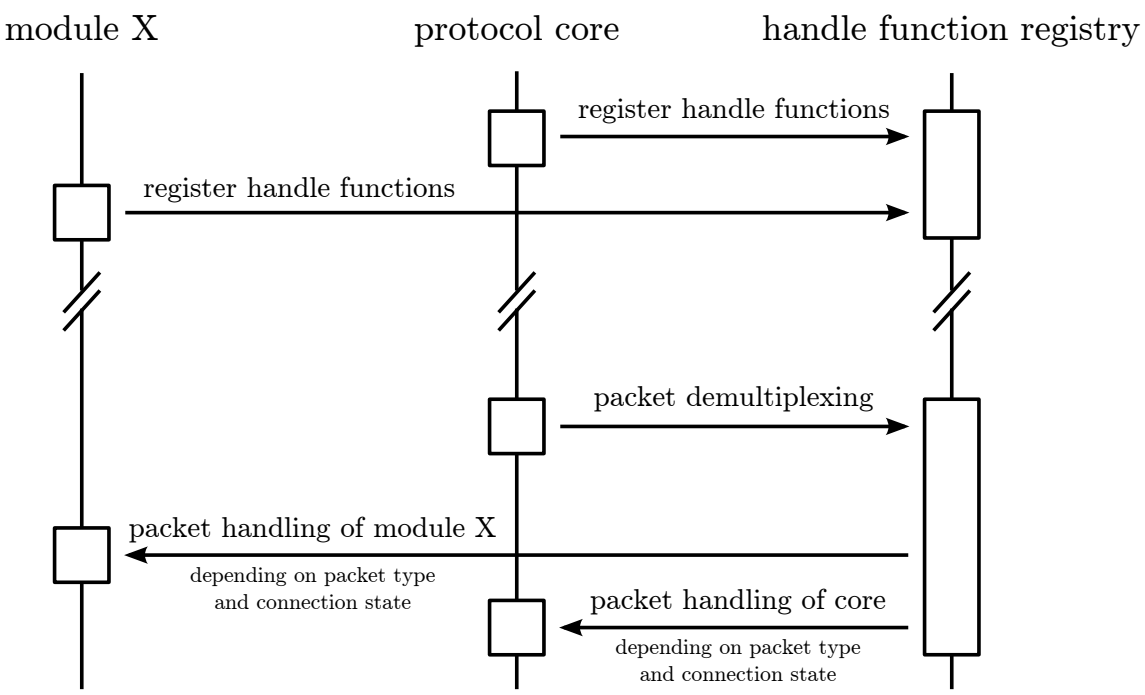

Figure 5.6 Modular packet handling over time

cannot determine reasonable priorities for new handle functions without further information.

In order to support the developer in determining priorities and to ensure the correct classification of handle functions, we divide the range of priority values in sections for specific tasks, such as security checks, parameter parsing, internal state transitions, generation of reply packets or transmission of reply packets. Notably, each phase must be finished, before the next one starts. This strict sequencing is essential for correct packet handling. If, for instance, the parameter parsing phase would start before the security phase has finished, security vulnerabilities could occur. Furthermore, one phase could influence the subsequent ones.

Usign predefined ranges for special purposes, each developer can classify the priority of new handle functions. The definition of exact priority ranges is protocol specific and hence we only describe the mechanism, but do not define real ranges.

5.3.3.2 Periodic Maintenance

Each protocol instance has to perform periodic activities, such as checks for retransmission of packets or acting on timeouts. We denote these activities with *periodic maintenance*. As protocol extensions can define new periodic activities, these must be registered with the protocol core to enable their execution. For this task, we again utilize the function registry mechanism. The protocol core defines an additional specialization of the function registry for this purpose. Hence, protocol core and modules register their maintenance functionalities, which will be executed in periodic intervals.

5.3.4 Module Preprocessing

Since the protocol core is not aware of the existing modules, it cannot initiate the registration of functionality that is implemented in modules. Thus, there is a need for a mechanism that initially passes the control over the program flow to the modules. Right at the protocol startup, during the protocol initialization phase, modules must be allowed to register their functionality.

We propose a module preprocessing routine (in the following *preprocessing*) that operates yet before the protocol initialization. The preprocessing analyzes meta information that each module must provide. Based on this analysis, dependencies of existent modules are checked. While the protocol core has no awareness about modules, the preprocessing routine has an global view. Thus, the module preprocessing provides information about the module initialization to the protocol core and hence, the core can initialize the modules, which may register as many features as needed.

In order to take advantage of the flexibility of modular systems, the preprocessing also provides a mechanism to select the modules that should be enabled and disabled. For instance, if modules are defective or incompatible to other modules, they need to be disabled. When the issue is resolved, the concerned module can be enabled again.

5.3.4.1 Module Meta Information

Modules provide meta information in a predefined structure. We distinguish mandatory fields that must be present and optional fields that may be added to the meta information. Each module must publish at least its name, version and information about its initialization process (see Section 5.3.5). Additional data fields are description, name of the developer, address for bug reports and webpage. The optional data has informative character and will help other developers to get further information. Furthermore, modules may have dependencies to other modules. These dependencies are also listed in the module meta information.

5.3.4.2 Module Dependencies

A single module should have as few dependencies as possible. With the exception of `libmod` and protocol core functionality, an extension should not depend on other building blocks. But there might be the situation, that dependencies are not avoidable, because modules are built upon each other. In this case, the module dependencies are explicitly listed in the module meta information. Thereby, the preprocessing routine can analyze the dependencies and check if all dependencies are fulfilled.

In general, we distinguish two dependency types: the *requires-dependency* and the *conflicts-dependency*. A module X requires module Y, if X cannot function without Y. For instance, a module providing user authentication could require a module providing cryptographic functionality, such as hash functions. In contrast, the modules X and Y conflict, if they cannot function together. This might be the case, if both

modules provide identical functionality in a different fashion. For instance, two developers could provide independent realizations of the same protocol extension. For each dependency, the developer adds an appropriate entry that contains the name of the required or conflicting module to the meta information.

Because modules evolve over time, each module provides a version number in its meta information. Using these version numbers, module dependencies can be refined by providing the minimal and maximal version number a dependency applies to. If, for instance, a conflict between two modules is removed, both modules can limit their incompatibility information to the concerned version numbers. In the case of one module requiring another one in a special version, it can set both the minimal and maximal version number to the required value.

The module preprocessing routine has an overall view of the existent components. All modules and their dependencies are known, making the performed dependency checks are exhaustive. As the module preprocessing routine acts before protocol core or modules are executed, that is previous to the runtime, incompatibilities can be found early and the developer can resolve them before the software integration. Thus, extensive modifications after the software integration can be avoided.

5.3.5 Module Initialization

As mentioned above, we define a module initialization phase, which is initiated by the protocol core based on information from the module preprocessing routine.

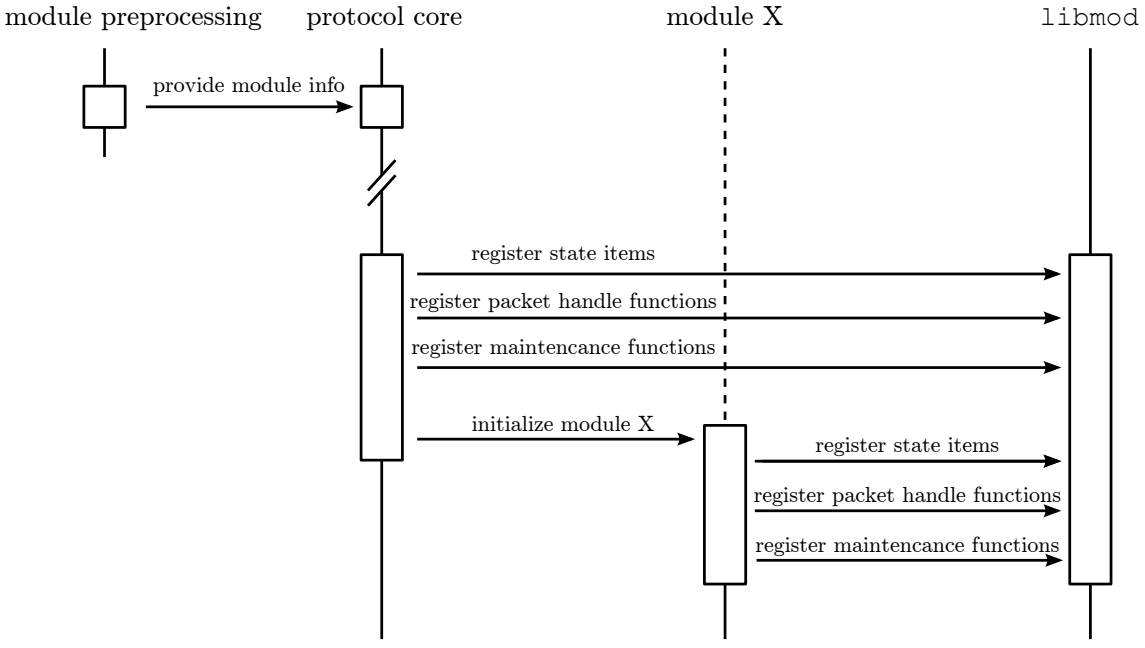

Figure 5.7 Initialization mechanism for modules

Figure 5.7 shows an exemplary initialization process. Each module names one initialization function in its meta data. The module preprocessing transfers this information to the protocol core. Afterwards, the core executes the initialization functions of all modules. Within its initialization function each module can register further

functionality. The described mechanism is again an specialization of the function registry from `libmod` and hence an application of the command pattern (see Section 2.1.4). The protocol core acts as invoker and performs requests (the module initializations) that are not known in advance. Therefore, all initialization functions must have the same interface. The modules act as receiver of initialization requests and handle them.

5.4 Modular HIP

The preceding section introduced general concepts for modular protocol implementations. In order to verify our approaches and give an example for their application, we design a modular HIP implementation. In this section, we describe the HIP specific adoptions of the general concepts in Section 5.3.

The identification of multiple self-contained building blocks is simplified by the modular specification structure of HIP. Mandatory features are defined in the base specification and implemented in the protocol core. Optional functionalities from the base specification or from protocol extensions are realized as modules. Firstly, we describe a modular design for implementing the HIP base specification with the introduced concept. Afterwards, we characterize the extensibility of the design by describing how the end-host mobility and multihoming extension can be integrated into the base implementation.

5.4.1 Base Functionality

The HIP base specification (RFC5201[31]) defines mandatory functionalities a HIP implementation must provide. This section describes how the mandatory HIP functionality can be implemented using our proposed design.

5.4.1.1 HIP Association State

Most notably, a host must save the current connection state (UNASSOCIATED, I1-SENT, I2-SENT, R2-SENT, ESTABLISHED, CLOSING, CLOSED or EXCHANGE-FAILED) for each HIP association. Additionally, various other information, such as HI, HIT, IP address or shared key, are needed in order to realize the mandatory base functionality. We utilize the modular state mechanism described in Section 5.3.2 for the management of all needed state information. The protocol registers a state item that saves the mentioned data fields for one host association. Thus, the protocol core and modules can access the basic association state data, if necessary. Furthermore, modules can register own state items containing extension specific information.

5.4.1.2 Handling of HIP Packets

The HIP base specification defines the syntax and semantic of the basic HIP packet types I1, I2, R1, R2, UPDATE, NOTIFY, CLOSE and CLOSE ACK. Except for

NOTIFY, a HIP implementation must handle all of them according to the base specification. For the modular HIP design we apply the packet handling mechanism described in Section 5.3.3.1 by adapting it to HIP. Due to the 7-bit packet type field in the HIP header, the maximal packet type number is 127 and the base specification defines eight connection states for a HIP association. For this reason, the HIP handle function registry must distinguish handle functions for 7 times 127 (889) possible parameter combinations.

Using the adopted handle function registry, the HIP protocol core registers its functionality during the initialization. Every time a packet arrives, the protocol core triggers the handle function registry to execute the previously registered handle functions. Furthermore, we specify concrete priority ranges for the HIP packet handling process.

1. **Initialization**. Prepare the packet handling and initialize required data structures.
2. **Security**. Check signatures and verify checksums of received packets.
3. **Packet Processing**. Process the received packet and trigger the desired functionality. In this phase packet type and parameters of the received packet are interpreted. Furthermore, the desired protocol functionalities are performed and a response packet may be created.
4. **Send Response**. Transmit the potential response packet.
5. **Clean-up**. Complete the packet handling, uninitialize the used data structures and manage timers.

In order to enable independent handle functions to collaborate, we need a mechanism for information transfer between the different phases and handle functions. For this purpose, we define a data structure that is called *HIP packet context* and contains information needed throughout the packet handling process. Foremost, the received packet and the data structure for a potential reply packet reside in the packet context. In addition, source and destination address as well as the current host association state are part of it. The protocol core initializes the packet context. During the packet handling process, each handle function may manipulate the packet context in order to fulfill its purpose. An important part of the packet context is the error flag. Each handle function may abort the further packet processing by setting the error flag. This might be necessary, if an error in the protocol operation occurs. For instance, if a signature or checksum verification fails.

5.4.1.3 HIP Maintenance Activities

The HIP base specification defines retransmission timers for sent control packets and for connections in the CLOSING state. The management of such timers is a typical, periodic maintenance activity. We utilize the periodic maintenance mechanism described Section 5.3.3.2 to provide the needed functionality.

Retransmission timers can be realized as follows: The protocol core registers a timer field in the modular state structure and a maintenance function that is executed periodically, for instance every second. When a control packet is sent, the function that transmits the packet starts the corresponding timer by setting the field to the desired timeout value. According to the HIP base specification, timeout values for HIP packets should exceed the anticipated worst-case round-trip time. Every time the registered maintenance is executed, it checks if the timer is set. In the case, the timer is set, the maintenance function decrements the value on each iteration. When the timer value reaches zero, the timeout has been reached and a retransmission is triggered.

5.4.2 Integration of Extensions

Now we describe how an exemplary extension for HIP can be integrated into our design for the HIP base functionality using the end-host mobility and multihoming extension (RFC5206[33]).

5.4.2.1 End-host Mobility and Multihoming

While the main consequences of implementing the mobility and multihoming extensions result from the new LOCATOR parameter, the extension does not define a new packet type. The existent base implementation neither specifies this parameter type nor defines a handle functionality for it. Therefore, all logic related to locators must be implemented in a new mobility and multihoming module. As locator parameters may occur in R1, I2, UPDATE and NOTIFY packets ([33], Section 5.3), the packet handling routine must be extended for multiple packet types. The mobility module registers additional handle functions for the affected packet types. Notably, the mobility and multihoming extension modifies the semantics of the HIP base exchange, because R1 and I2 packets must be handled in a different way. Furthermore, we need to save new state information in order to integrate the considered extension. These are the locator data itself, that is multiple addresses per host, as well as the locator states (locators may be UNVERIFIED, ACTIVE or DEPRECATED).

In order to realize the mobility and multihoming extension, we design a module that depends on functionalities provided by the protocol core and `libmod`. There are no additional dependencies to other modules. Thus, the module meta information must only contain the modules name, its version and an initialization function. Regarding our new module, the only task of the module preprocessing routine is to provide the initialization information to the protocol core. During the module initialization, the mobility and multihoming extension registers all further functionalities.

As mentioned above, we need to create a new state item in the modular state structure. This item is used to store all locators per associated host. The mobility and multihoming module registers the state item during its initialization process. Afterwards it can access the data at any designated position. Modifications to the protocol core are not necessary.

The packet handling is implemented in the protocol core using the function registry concept from `libmod`. In order to realize the handling of LOCATOR parameters,

the mobility and multihoming module has to modify the packet handling. For that purpose, it registers handle functions for the needed packet types including R1, I2, UPDATE and NOTIFY. By the usage of locators, outgoing HIP packets might be sent to addresses the protocol core is not aware of. Thus, we add the locator handle mechanism to the end of the packet processing. That is, we register a handle function with the highest priority number of the packet processing range. Therefore, the original packet handling is performed as usual. The core may create a response packet with the source address of the received packet. After the core has finished his handling, the mobility and multihoming extension parses the received packet for LOCATOR parameters. If such parameters are present, it takes the desired actions like updating the mobility and multihoming state item, which contains the locator information. When the locator handling is finished, the module checks if there is need to modify the response packet generated by the core. As the core does not know locators, it might use the wrong address. The mobility and multihoming extension can correct the address field of the outgoing packet and hence realize the desired functionality. As the mobility and multihoming extension includes new logic for the addressing of outgoing packets, the described locator handling mechanism must be added to the packet handle mechanism for all packet types. The original packet processing is not influenced by this operation.

Conclusion

The integration of the mobility and multihoming extension into the HIP base implementation shows that our modular design is extensible. Even central aspects of the protocol, such as the handling of base exchange packets, was modified without any changes in the protocol core. All new protocol logic is encapsulated in the module, achieving a minimal dependency level.

6

Implementation

The implementation of our design proposed in Section 5.3 bases on the open source software project *Host Identity Protocol for Linux* (HIPL)[4]. HIPL is maintained by the Aalto University (former Helsinki University of Technology), the Helsinki Institute for Information Technology and the Distributed Systems Group at RWTH Aachen University. The HIPL project implements an extensive features set of the HIP base specification and numerous extensions including the end-host mobility and multihoming[33], the rendezvous[26] and the registration[27] extension. Furthermore, inofficial research extensions, such as BOS (see HIPL user manual[3] for details), are realized in HIPL. The functionality of HIPL is implemented in three userspace applications, namely the HIP daemon (`hipd`), the HIP firewall (`hipfw`) and the HIP configuration tool (`hipconf`). While `hipd` realizes the actual protocol functionality, the `hipfw` provides helper functionality for the `hipd`, such as reading packets from the network stack. Furthermore, the HIP firewall allows filtering of HIP traffic and implements a HIP proxy. `hipconf` can be used for runtime configuration of the `hipd`. Common functionality for the applications is encapsulated in a library called `libhipcore`. In addition, HIPL uses standard libraries, such as OpenSSL[42] for cryptography support.

While HIP conceptually resides between the network and transport layers, the implementation (`hipd`) is arranged mostly on the application layer. Therefore, HIPL enables network packet processing in the userspace. The HIPL source code is written in C and can be build on multiple Linux-related platforms including Linux, OpenWRT[1] and Maemo[34]. For these platforms, the HIPL software compiles without issues and is frequently tested.

In the following sections, we will analyze the design of HIPL and describe how we have improved the protocol implementation by modularization of `hipd`. Section 6.1 presents the HIPL shortcomings that should be eliminated through the modularization. Afterwards, Section 6.2 evaluates the utility of existing approaches to the realization of our design. Subsequently, Section 6.3 describes the implementation of the modularization library `libmod` and gives usage instructions. Finally, Section 6.4 presents the application of `libmod` to the `hipd`.

6.1 HIPL Design Shortcomings

The `hipd` is an example for a monolithic implementation of a flexible protocol. Our problem analysis in Chapter 4 shows that monolithic implementations of flexible protocols are not sustainable.

Furthermore, the vast number of features implemented in HIPL intensifies this situation. Optional features of HIPL are enabled or disabled using precompiler statements. These statements and hence the optional code are inserted at arbitrary positions to the base implementation. Thus, related code blocks are distributed among the entire code base. Indeed, this code can be disabled, that is removed by the precompiler, but the comprehension of such scattered code is very difficult. Besides the usage of precompiler statements to remove currently unused code, reduces the code quality because the concerned code blocks are not continuously compiled. Thus, issues regarding these code blocks, may result from recent changes, but not be found, because the code is removed by the precompiler.

In addition, central parts of the `hipd`, such as the packet handling, are implemented using static `switch` statements. This reduces the flexibility, because extending the functionality requires changes to these static structures.

As application of our modular design, we will realize `libmod` and integrate it to HIPL. The goal of this process is the modularization of `hipd` and hence the elimination of the above shortcomings.

6.2 Evaluation of Existing Approaches

As presented in Section 3.3, there are programming frameworks that might be used for modular protocol implementations. Now we examine whether the introduced frameworks are applicable for the realization of our design.

6.2.1 C-Pluff

The modular design could be realized using a plug-in framework like C-Pluff (see Section 3.3.1). Such frameworks offer the possibility to load and unload functionality in the shape of plug-ins during runtime. In our opinion, this is a dispensable feature for protocol implementations. End users will not change the functionality of their systems protocol stack. For experts on protocol experimentation, it is sufficient to define the protocol functionality during build time. Nevertheless, we examine the consequences of dynamic plug-in loading for two reasons. Firstly, existent solutions like C-Pluff provide this functionality. These solutions could be used in order to reduce the implementation complexity and effort for modular protocols. Secondly, dynamic plug-in loading could be needed in future usage scenarios that are currently not predictable. With dynamic loading, the protocol functionality could be adapted to the current needs during runtime. For instance, the protocol core could dynamically load the handle functionality for a new packet type, when the concerned packet type occurs the first time.

The dynamic loading approach may cause compatibility issues. Frameworks and hence their interfaces evolve over time. If main program and plug-ins are maintained separately, it is possible that one installation contains plug-ins that are built for different versions of the framework. Hence, their interfaces may be incompatible to each other or to the main program. In the case dynamic loading, these compatibility issues are not detected until program is executed. So, the employed components must be adjusted after their integration. On embedded devices, such as smartphones or routers, the replacement of components might be especially complicated or not even possible.

Furthermore, the C-Pluff development team consist of only one person and hence the project maintenance is strongly limited. Thus, the utilization of C-Pluff could entail additional maintenance tasks related to the project-external C-Pluff source code.

Due to the mentioned drawbacks of dynamic loading and the C-Pluff project, we will not use this framework for the realization of our design.

6.2.2 libevent

In Section 3.3.2 we introduced libevent - an asynchronous event notification library. libevent may be used to realize the desired functionality of registration callback functions to specified events.

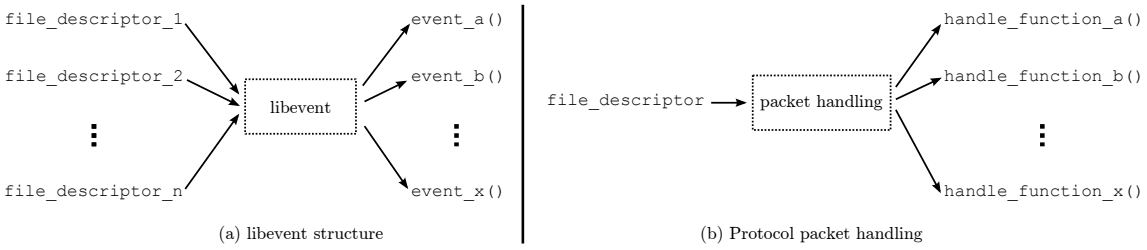

Figure 6.1 Comparison of the simplified libevent structure with a simple packet handling mechanism in network protocols

Figure 6.1 (a) depicts the highly simplified structure of the libevent callback mechanism for file descriptors. As shown, libevent focuses on a usage scenario with multiple file descriptors that must be mapped to event handle functions. Depending on file descriptor changes, libevent performs the designated functionality. The key advantage of the libevent handling mechanism is its asynchronism - the handle functionality may be executed in a new thread and hence the main program can resume its work immediately.

Considering the packet handling of network protocols (Figure 6.1 (b)), the demultiplexing does not necessarily depend on the file descriptor a packet was received on, but may also depend on the packet content. In the simplest case only one file descriptor is observed and all received data is processed by the packet handling routine. Thus a mechanism for managing multiple file descriptors is not needed. Instead we need to realize packet demultiplexing based on packet type of the received packet and the current status. Hence, the utilization of libevent for the realization of `libmod` is not sustainable.

6.3 Realization of `libmod`

As none of the existing solutions fits our requirements, we start with the implementation of `libmod` from scratch, without the utilization of existing projects. We aim to create a generic library that realizes the design described in Section 5.3.

This section presents our implementation of `libmod` using the C programming language and gives instructions how the library can be used for the implementation of modular protocol software.

6.3.1 Modular State

The modular state structure enables protocol core and modules to register and modify state items. For each associated host, the protocol core creates a dedicated modular state variable using functionality from `libmod`.

We assume that the protocol core provides a database for the management of host association data and hence concentrate on how to structure one entry of such a database. Protocol core and modules should be enabled to independently save and modify information in one database entry.

During the initialization phase of the protocol software, protocol core and the modules can register state initialization functions by providing a function pointer to `libmod`. Every time the protocol core needs to create a new association state variable, it triggers `libmod` to create a new state data structure. `Libmod` iterates over all registered state initialization functions and returns a pointer to the created data structure to the core.

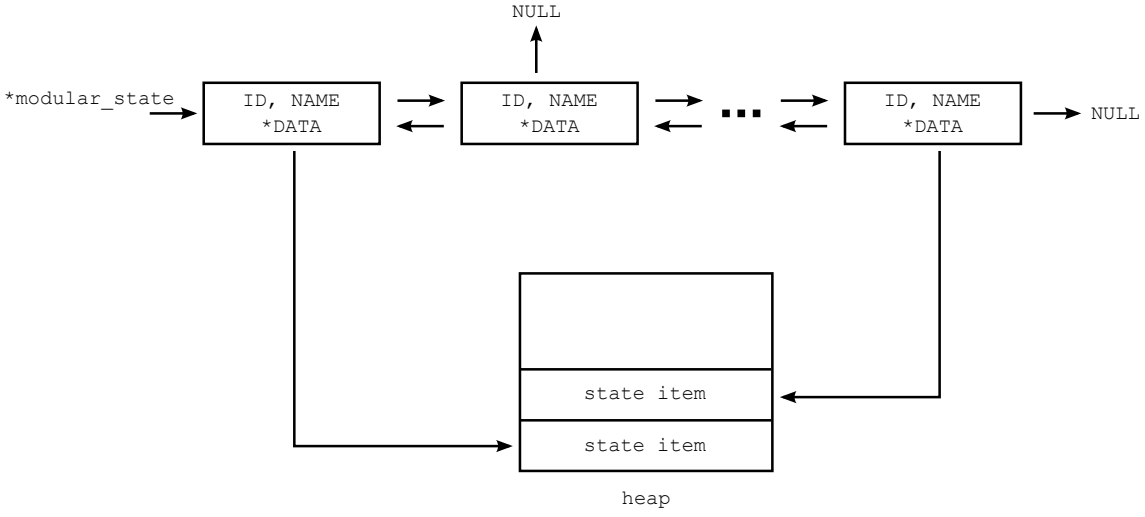

Figure 6.2 Data structure of the modular state

The modular state data structure itself is a double-linked list of state item containers. Figure 6.2 visualizes how the modular state data structure is arranged in the memory. Each container stores pointers for the previous and the next element as well as a pointer to the actual state item (`*DATA`). State item containers can be addressed by integer (`ID`) or string (`NAME`) identifiers. Both values are unique and `libmod` stores

their mapping. While IDs are dynamically assigned during the registration of state items, names are chosen by the developers. However, state items can be accessed using both values. The registering module can remember the ID and hence directly access the state item. Other modules may be aware of the assigned string identifiers for state items and hence access them using the name. In this case, `libmod` performs the mapping from name to ID and then returns the requested data. Since the number of existing modules and hence the number of registered state items will be limited, the described look-up mechanisms using linked-lists should not cause performance issues. In case, the number of state items significantly increases, the used data structure for storing state items could be replaced with a more sophisticated one.

As described above, the modular state data structure can be used to manage the registered state items. The actual state items are created individually by each component. Thus, each component can define the structure of its state items, according to its needs. This allows maximal flexibility regarding the shape of state items. Since all mentioned variables are allocated on the heap, `libmod` provides a cleanup function that frees the used memory.

6.3.2 Function Registry

The `libmod` function registry is designed to provide a general realization of the publish/subscribe mechanism (see Section 5.3.3). In order to create a reusable library that can be adapted to concrete usage scenarios, we implemented an abstract, list-based publish-subscribe functionality based on C function pointers. The `libmod` function registry provides a function for the registration of function pointers to a specified functionality, such as the packet handling or the periodic maintenance. For each of this functionalities exists one publish/subscribe list. Function pointers will be added to the list according to their priorities. Thus, all registered function pointers are sorted ascending in the respective function registry list. In addition, registered functions can be unregistered by providing the respective list and the function pointer itself. Again, the usage of lists should not cause performance issues, because of the limited number of registered functions. Otherwise the data structure could be replaced.

The implementation of further functionality, such as iterating over the list, resolving and executing the function pointers, is left to the protocol core, because this functionalities depend on the respective usage scenario. However, the general function registry provides basis for further functionalities, such as packet handling and periodic maintenance. Sections 6.4.3, 6.4.1 and 6.4.2 explain such concrete realization for the `hipd`.

6.3.3 Packet Type Registry

Protocol extensions may define new packet types that are not known by the core. Thus, the protocol core is not aware which or how many packet types exist. This situation hinders practical functionalities, such as debugging information with packet type information. We propose a *packet type registry* that stores all registered packet types. For each packet type the responsible component (protocol core or module)

makes an entry consisting of packet type number as integer value and packet type name as string identifier. All components can read the information from the packet type registry and hence all packet types are known to all components.

The string identifier might be useful as debug information, but does not add functionality to the overall protocol functionality. Components can query the packet type registry for a specific packet type number and check if the number already exist. The realization of the packet type registry bases on a linked list data structure. Each list item stores number and string identifier of one packet type.

6.4 Applying `libmod` to HIP For Linux

After we have described the implementation of `libmod` in Section 6.3, we now present its integration into HIPL. Our implementation concerns the `hipd`, as the protocol functionality is realized in this application.

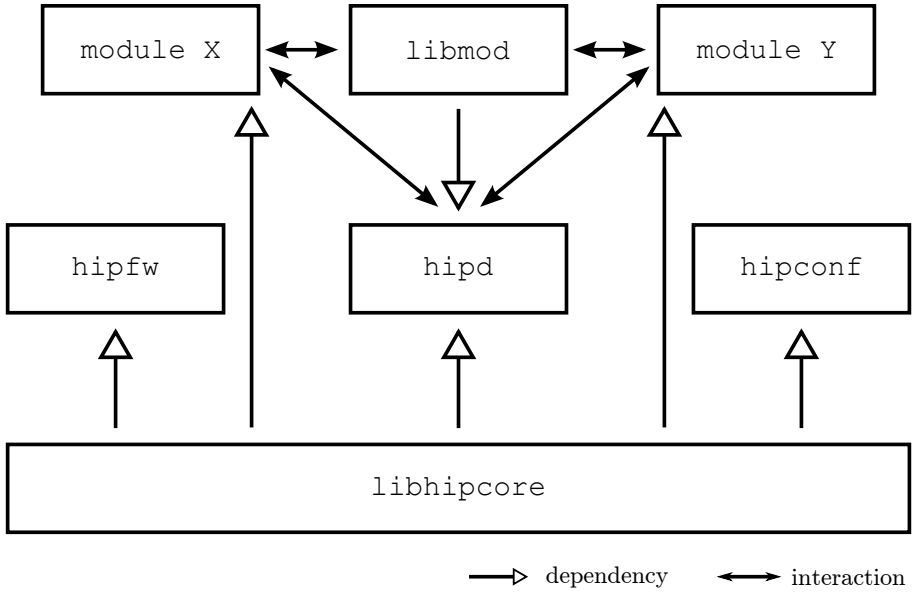

Figure 6.3 HIPL building blocks on the implementation level

Figure 6.3 shows the HIPL components after integration of `libmod`. For now, only `hipd` uses the modularization library. However, `hipfw` and `hipconf` are implemented in the same fashion as `hipd`, they could also be modularized using `libmod`. The modular `hipd` depends on `libhipcore` and `libmod`. Modules, that provide additional features, are integrated using the modularization library and may depend on functionality of `libhipcore`.

In order to realize the modular design, we re-engineer `hipd`. The first step of this modularization process is identifying functionalities that are integrated in the `hipd`, but should be realized as modules. For that task, the precompiler statements indicate which features are optional. We have removed these features as a start, minimizing the protocol core to the essential functionalities. This step has considerably reduced the complexity of `hipd`. However, the removed functionality can be inserted again - in the shape of modules that can be integrated without changes to the protocol

core. Thus, the next step is to identify useful extensions that have sufficient code quality and to integrate them again as modules that adhere to the new design.

6.4.1 Packet Handling

We realize the packet handling routine for HIP using the general function registry from libmod. Since the demultiplexing of HIP packets is based on the packet type of the received packet and the current association state, these are the two essential parameters. Hence, we need to manage a set of handle functions for each possible combination of the two parameters. Therefore, we initialize a two-dimensional array with the maximum parameter values as boundaries during the program startup. After the initialization, all array fields are NULL pointers without further functioning.

In order to ensure compatibility throughout the various handle functions, we define a mandatory data type for all handle functions. This data type specifies the return value and the expected parameters for the registered function pointers. The registration of handle functions is done, using this consistent interface. Therefore we create a slim wrapper function around the original register function form libmod. The new wrapper function expects a function pointer with the mandatory data type for handle functions, and a combination of packet type and association state. Based on this information it chooses the adequate array entry and registers the function pointer by executing registration functionality from libmod. The mentioned wrapper function is an application of the adapter design pattern. The advantage of such an implementation is its flexibility, because the internal realization is abstracted. If, for instance, the libmod interface changes, the according modification to the protocol core, must only be done once - in the wrapper function.

The actual packet handling routine must execute all functions that were registered for the received packet type in the current association state. Thus, we implement a general iteration function that executes all handle functions in a specified list. This realization is only feasible, due to the consistent structure of all registered function pointers. This is an application of the command design pattern. Notably, the protocol core can iterate over the handle functions without considering the respective priority, because libmod automatically sorts the handle functions according their priority. For consistency reasons, we also create a wrapper around libmod's deregistration function.

Thus, all utilized functions share the same interface. Based on the consistent data type for handle function pointers, the handle function registration can be checked already during build-time. The compiler knows the mandatory types and hence incompatibilities are found during compilation. Due to the static linking, incompatibilities can not occur, after the build-time.

6.4.2 Periodic Maintenance

The periodic maintenance implementation bases on the same concepts as the packet handling routine implementation (Section 6.4.1). We specialize the general function registry functionality from libmod by creating wrapper functions around the maintenance function registration and deregistration.

6.4.3 Module Initialization

During the protocol core startup, modules must be initialized (see Section 5.3.5). For this purpose, each module defines an initialization function and a header file with the adequate function prototype in its meta information. The protocol core includes a header file, which was generated from the module preprocessing routine and contains the information required for the module initialization. Hence, the protocol core iterates over the provided information and executes all listed module initialization functions. Within these functions, the modules register further functionality, such as packet handling and periodic maintenance functionalities.

Furthermore, there is the possibility to disable integrated modules at the program's startup by providing the module's name as parameter. During startup, the protocol core checks, if the provided modules can be safely, that is without violating dependencies, disabled. If so, the protocol core skips the accordant initialization functions during module initialization. Thus, the functionality remains in the software, but it is not registered and hence never executed. The module deactivation at startup depends on the provided information from the preprocessing algorithm (see Section 6.5.2).

6.5 Module Preprocessing

In our opinion, there is no need to modify a protocol implementation during runtime. Especially, end-users are not concerned about which protocol extensions are loaded, or not. Therefore, the decision which modules should be loaded can be made before runtime. Thus, we realize the module preprocessing mechanism in the build system yet before the compilation starts. Hence, the compiler can greatly improve the performance by inlining and replacement of function pointers by direct calls.

In order to enable the module preprocessing in the build system, we have evaluated various build automation tools including CMake[25], SCons[2], Make[14] and GNU Autotools (consisting of Autoconf[16], Automake[17] and Libtool[15]). Finally, we chose GNU Autootools for the following reasons. First, the Autotools suite is available as standard package on all targeted platforms Linux, OpenWRT and Maemo. Furthermore, it is widely-used in the Linux and Unix environment and hence many developers are familiar with the used concepts. Third, HIPL already uses GNU Autotools as build system - thus the integration effort is limited. The actual module preprocessing mechanism as part of the build system is realized as python[41] script, because python provides the needed functionalities, such as file operations, parsing of configuration files and sophisticated data types.

All modules are located in a designated source folder, called `modules`. Within this folder, each module has an own directory with all needed data. Hence, modules can be removed by simply deleting the respective directory and re-configuration of the build system - that is executing the `configure` script.

As described in the design (see Section 5.3.4.1), each module provides meta information that are analyzed during the module preprocessing. Next, Section 6.5.1

describes how the module meta information is represented, before Section 6.5.2 explains the module preprocessing mechanism. Finally, Section 6.4.3 describes the module initialization process.

6.5.1 Module Meta Information

We represent the module meta information in an Extensible Markup Language[44] (XML) file. XML is an open standard for textual data formats that provides a clean structure and is readable for programs as well as humans. Furthermore, the syntax of XML files can be verified using XML schema definitions. Many programming languages, including python, provide programming libraries that allow parsing and modifying of XML files. We chose XML as representation format, because is allows a high flexibility and can be used to elegantly express the desired data, such as module dependencies.

```
<?xml version="1.0" encoding="UTF-8"?>

<module
    name="moduleX"
    version="1.0"
    description="This module provides functionality X"
    developer="John Doe"
    bugaddress="hipl-users@freelists.org"
    webpage="http://infrahip.hiit.fi/">

    <requires>
        <module name="registration" minversion="0.9" maxversion="0.9" />
        <module name="mobility" minversion="1.1" />
    </requires>

    <conflicts>
        <module name="moduleY" maxversion="0.9" />
    </conflicts>

    <application
        name="hipd"
        header_file="modules/moduleX/hipd/moduleX.h"
        init_function="hip_moduleX_init" />
</module>
```

Figure 6.4 Example for module meta information in XML format

Figure 6.4 shows the structure we define for module meta information. All mandatory information (*tags* and *attributes*), are printed in **bold** font. According to the design (see Section 5.3.4.1) each module must provide its name and version. In addition, we define a mandatory application tag that specifies the integration of a module and the respective program core. Thus, the application tag contains the name of the application, a header file that will be included by this application and a module initialization function.

Further optional fields, which have only informational character, may be present in the meta information. These fields are description, name of the developer, address

for bug reports as well as webpage. Modules dependencies may have two types requires or conflicts. If a dependency tag is present, it must provide the name as mandatory attribute. The minimum and maximum version numbers are optional and can be used for refining the dependency. For instance, the module with the meta information in Figure 6.4 depends on three different modules. While the registration module is required in exactly version 0.9, the mobility module must have at least version 1.1. The conflict with moduleY persists up to version 0.9 of moduleY. Through the flexibility of XML, module dependencies can be expressed adequately and elegantly.

6.5.2 Preprocessing Algorithm

We realize the module preprocessing algorithm in the build system and even before compiler invocation. This enables a high optimization potential for the compiler. Optimizations such as inlining and resolving of function pointers are possible, due to the module integration being static. That means there are no changes to the registered function pointers during runtime. Sophisticated compilers, such as the `gcc` (GNU Compiler Collection [19]), can identify and realize these optimization potentials[18].

As defined in the design (see Section 5.3.4), the preprocessing algorithm has to perform three activities.

1. **Parse** module meta information.
2. **Analyze** meta information. This step includes verification of completeness and dependency checks.
3. **Provide** module initialization information to the protocol core.

We implement these features in a python script, which is integrated into the standard Autotools build system of HIPL and executed by the `configure` script. Python is used, because it is available on the compilation tool-chains of targeted platforms and provides convenient features for the desired tasks, such as file operations and parsing of XML files. Furthermore, python comes with sophisticated data types that can be used for elegant information management, such as checks of dependencies. In order to execute the python script, we extend the project configuration mechanism. Every time the `configure` script runs, it triggers the python script. Furthermore, we create a configuration option to disable modules. If this option is used, the user can provide a list of modules that should be disabled. This list is delegated to the python script that will ignore the denoted modules during the parsing phase. Hence, the module will neither be built nor linked to the protocol core.

After the python script has read the meta information of all enabled modules, it checks whether the mandatory data was provided completely. If so, the module dependencies are checked by iterating over all modules and verifying that all required modules are existent and enabled. Furthermore, the preprocessing ensures that the current configuration does not contain conflicts.

The last step of the module preprocessing mechanism prepares the collected information for the usage in the protocol core. Hence, the python script creates a C header file that is included by the modular application. The header file contains a list with all enabled modules, the respective initialization functions and the module dependencies. However, the dependencies were checked during the preprocessing, the application may need to perform a simplified dependency check, because modules were disabled at program startup. Within this check, the application verifies that all required modules are loaded or the dependent modules are disabled as well. Checks of module versions or conflicts dependencies are not necessary, because the preprocessing has already performed them and due to the static linking, incompatibilities cannot occur after building.

Furthermore, the generated header file includes the module header files that are specified in the meta information. Hence, the protocol core can resolve the denoted initialization functions that are located in the module source code.

7

Evaluation

In this chapter we discuss the results of our work by evaluating our design against the design goals stated in Section 5.1. The evaluation consists a quantitative performance analysis (Section 7.1) and a qualitative discussion of the design goals (Section 7.2).

7.1 Performance Analysis

In Section 6.4 we describe the application of our modularization library `libmod` to the HIP for Linux project. In order to evaluate the performance-related consequences of our modularization, we compare the original version of `hipd` (*monolithic* ***`hipd`***) with our implementation integrating `libmod` (*modular* ***`hipd`***). While both versions of `hipd` are single-threaded userspace applications, they differ slightly in the implemented set of extensions. The monolithic `hipd` contains various research extensions that can be disabled using configure options, which cause the precompiler to remove the respective code. In contrast, the modular version of `hipd` includes currently three modules: mobility and multihoming, heartbeat and heartbeat-update. In all test runs the utilized versions of `hipd` were as similar as possible. The precompiler of the monolithic version was configured to remove all unneeded extensions and the module preprocessing of the modular version was configured to enable all existing modules, because we want to analyze impact caused by modules.

For our quantitative evaluation we measure the duration of the `configure` run in the HIPL build system, the `hipd` startup and the protocol packet handling for the basic HIP packet types I1, I2, R1, R2, UPDATE, CLOSE and CLOSE ACK. The measurements were performed on all platforms currently supported by HIPL: Linux, OpenWRT and Maemo.

7.1.1 Test Setup

In order to evaluate all supported platforms, we perform measurements with one representative for each platform. For the Linux related measurements we use two standard personal computers with an AMD Athlon 64 X2 Dual Core Processor 4800+ at 3200 MHz and 4 Gigabyte RAM. The personal computers run Ubuntu 10.04 stocked with Linux kernel 2.6.32-21 x86_64 and were connected to an isolated network using a Gigabit Ethernet switch. As platform for the OpenWRT related measurements, we utilize a Linksys WRT160NL Router with an Atheros CPU at 400 MHz and 32 Megabyte RAM. For measurements on the Maemo platform, the Nokia N900 smartphone with 600 MHz ARM Cortex CPU and 256 Megabyte system memory is used. In order to create comparable results, the Nokia N900 was connected to its power supply during all measurements. Thus, deviations due to variability in the battery state and power management were reduced to a minimum. For each performance value to be evaluated, we took a series of 100 measurements and compute the average as well as the standard deviation, except if noted otherwise.

7.1.2 Module Preprocessing

As described in Section 6.5.2 we realize the module preprocessing in the build system. Even if the duration of the build process does not directly relate with the protocol implementation performance, we analyze the impacts of the modularization on the build system in order to evaluate all aspects of our work. Since the module preprocessing is embedded into the `configure` run, we measure the overall execution time of one configure run with the unmodified software (Monolithic HIPL) and with the modularized HIPL (Modular HIPL). We have performed the measurements using the UNIX time command by executing `time ./configure` on the standard computers described in 7.1.1. monolithic: standard set of features, modular with all modules enabled.

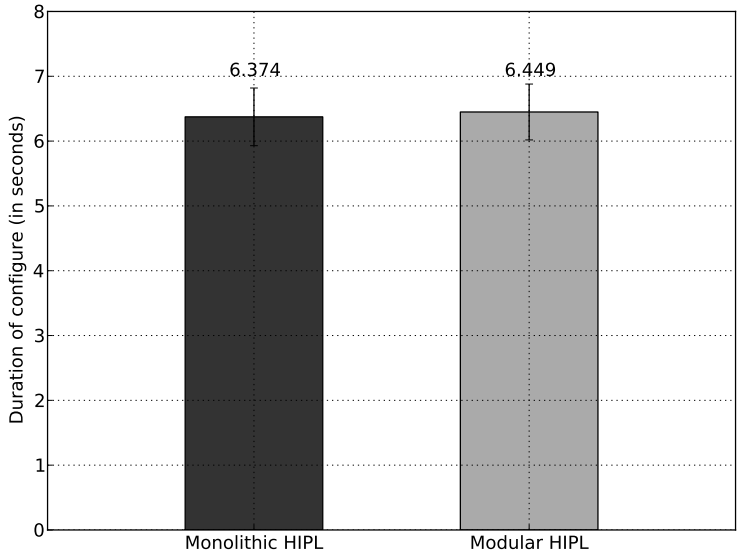

Figure 7.1 Average duration of one `configure` run with standard deviation

Figure 7.1 shows the results of our measurements. With an average configuration time of 6.374 seconds the monolithic version is minimal faster than the modular ver-

sion of HIPL with 6.449 seconds. The slowdown of average 0.075 seconds (+1,18%) results from the additional execution of python, parsing the XML meta information, checking module dependencies and creating the header file. The difference is significantly smaller than the standard deviations of 0.445 seconds for monolithic HIPL and 0.431 seconds for modular HIPL. The high standard deviation can be explained by the not avoidable impact of other processes running on the build computer.

The results show that it is not predictable which `configure` execution will be faster. Thus, the difference between monolithic and modular HIPL is hardly perceptible in practice and absolutely acceptable.

7.1.3 Module Initialization

The module initialization (see Section 6.4.3) for the modular `hipd` is performed at the program startup. During the initialization phase modules register all of their functionality the protocol core must be aware of. For instance, state items, packet handle functions and maintenance functionality is registered during this phase. Thus, the modularization could influence the startup time of `hipd`. However, the impact of the startup time of a daemon is not a hard, because the startup is only performed once.

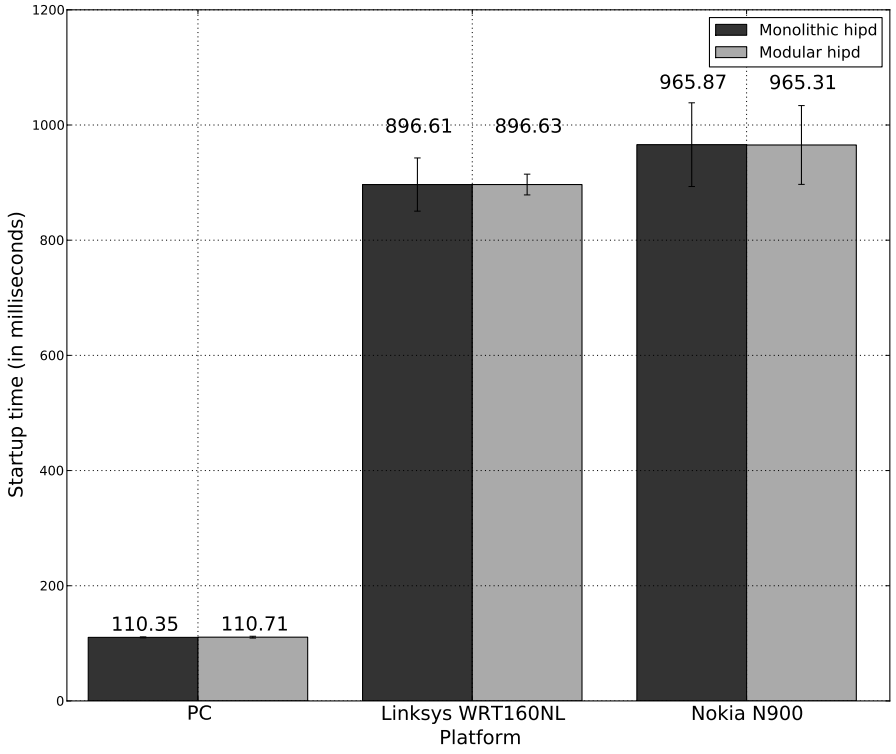

Figure 7.2 Average startup times of the `hipd` with standard deviation

To examine the modularization consequences, we measure the startup time of the monolithic and the modular `hipd` on a PC, a WRT160NL and a Nokia N900. Thereby, startup time denotes the time between invocation of the daemon and the initiation of the main loop; the processing times in the main loop are objective to another measurement (see Section 7.1.4).

The `hipd` initialization phase contains necessary preparations, such as reading of configuration files and opening of network sockets. For the modular version of `hipd`, the initialization phase additionally includes the entire functionality registration of the protocol core and three modules (mobility and multihoming, heartbeat, heartbeat-update), which adds up to 130 registrations.

Figure 7.2 depicts the average startup times for the monolithic and the modular `hipd` on the three target platforms and their standard deviations. The differences between the monolithic and the modular version are nearly nonexistent. On personal computers (PC) the average overhead amounts to 0.36 ms (<1%); on the Linksys WRT160NL the startup of the modular version is slowed down at 0.02 ms (<1%) and on the Nokia N900 the average startup of the modular version is actually faster at 0.56 ms (<1%). All of these differences are significantly smaller than the standard deviation of the respective measured values. For this reason, the we argue that the modularization does not slow down the startup time of `hipd`.

7.1.4 Packet Handling

The processing of received packets is a central aspect of protocol implementations and its execution time is an important indicator for the overall performance of a protocol implementation. In contrast, to the initialization phase, which is performed only once per execution, the packet handling mechanism is executed for every received packet. Hence, a low performance of the packet handling could cause unacceptable delays.

The monolithic version of `hipd` demultiplexes control packets using two nested switch-statements. The outer one parses the packet and executes a hard-coded handle function for the respective packet type. Inside of this handle function, the inner switch statement depends on the current connection state and is used for determining which functionality should be executed.

We have modularized the packet handling mechanism of `hipd` for all processed packet types. Thereby, protocol core and modules register their handle functionality for combinations of packet types and connection states during the initialization and `libmod` executes the designated functionality, when a matched packet arrives in the adequate state.

In order to analyze the described packet handling mechanisms, we made performance measurements with both `hipd` versions on the three targeted platforms. We installed one measuring point per packet type and quantified the entire packet processing mechanism, starting from the packet parsing up to the transmission of the response packet.

The test procedure contains a HIP base exchange, an UPDATE procedure and the connection closure (see Section 2.2.3 for details). While, all operations are triggered from the initiator host, the responder replies accordingly. Due to the heavy use of public-key cryptography in control packets HIP, the major part of the packet processing time is spend for the execution of cryptographic algorithms. The cryptographic functionality was not modified during the modularization and for this reason, we expect little differences for the packet handling between monolithic and modular HIP.

7.1.4.1 Personal Computer

We connected to standard personal computers (see Section 7.1.1) using a dedicated Gigabit Ethernet switch and performed the test procedure described above.

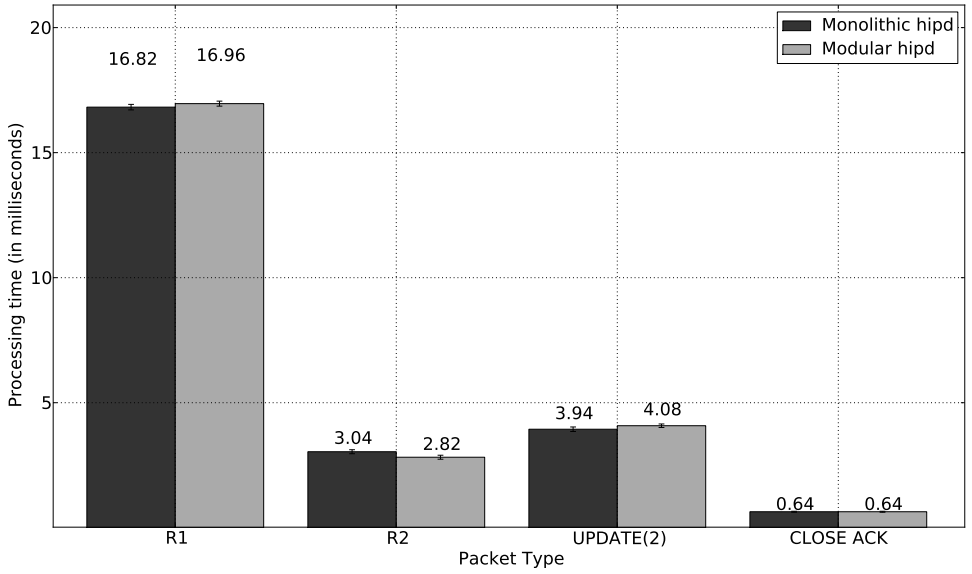

Figure 7.3 Average packet processing time on PC (initiator) with standard deviation

Figure 7.3 depicts the results for the initiator. The R1 packet handling shows only little differences (<1%) that can be explained with external influences. Notably, the R1 packet handling lasts longer than the handling of other packets, because the initiator must solve the responder's puzzle challenge (see Section 2.2.3.4 for details).

The R2 packet handling has a drift of (-7.24%). This can be reasoned with a functional difference between the monolithic and the modular `hipd`. While the modular version encapsulates the heartbeat functionality into a dedicated module, the correspondant functionality is scattered to the entire code base for the monolithic version. The R2 packet handling in monolithic `hipd` contains the creation (including memory allocation) and sending of the first heartbeat packet. This additional effort in the monolithic version explains the higher processing time for R2 packets.

The processing of the second received UPDATE packet takes 0.14 ms (+3.55 %) longer in the modular version of `hipd`. This marginal differnce bases on external influences. CLOSE ACK packets are processed with the same average time in both versions.

The packet processing times for the responder PC are shown in Figure 7.4. There are marginal differences between the average packet processing time for monolithic `hipd` and modular `hipd` for all packet types, except the first UPDATE packet. The mentioned differences caused by external influences and hence negligable. Since the handling of HIP I2 packets causes various cryptographic computations, the clear difference in the processing time of I2 and the other packets is explainable.

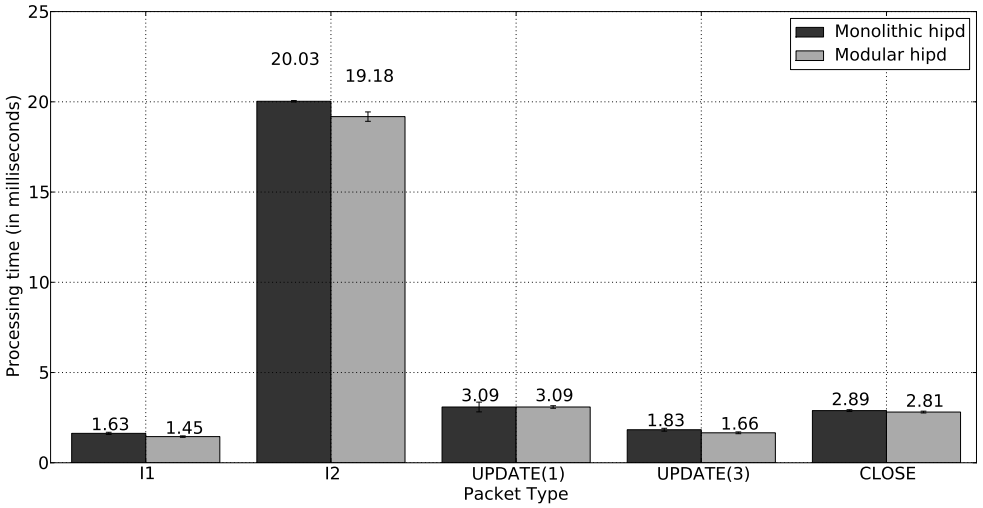

Figure 7.4 Average packet processing time on PC (responder) with standard deviation

7.1.4.2 Nokia N900

For the performance analysis on the Maemo platform we utilize a Nokia N900 smartphone as initiator and a Linksys WRT160NL router running OpenWRT as responder and connect the devices via a 802.11 wireless LAN.

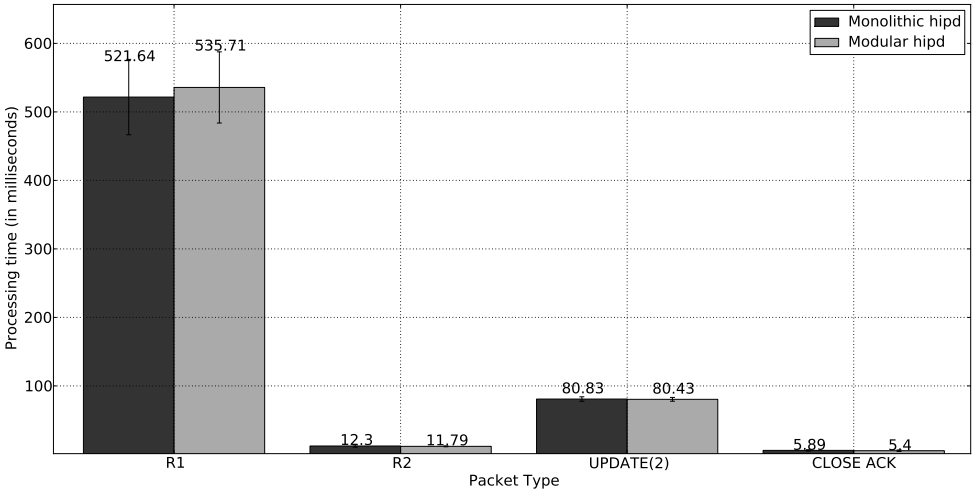

Figure 7.5 Average packet processing time on Nokia N900 (initiator) with standard deviation

For the performance measurements on the Nokia N900 (see Figure 7.5), there are only little differences between the monolithic and the modular `hipd`. According to the results on a standard computer, the R1 processing takes longer with the modular version. Notably, the standard deviation for the R1 packet handling is extensive. The reason for this variability are external influences that affect the performance of the cryptographic computations, which are needed for the handling of R1 packets.

7.1.4.3 Linksys WRT160NL

As mentioned in Section 7.1.4.2, the performance measurements for the WRT160NL Router are taken using the Nokia N900 as initiator and the router as responder.

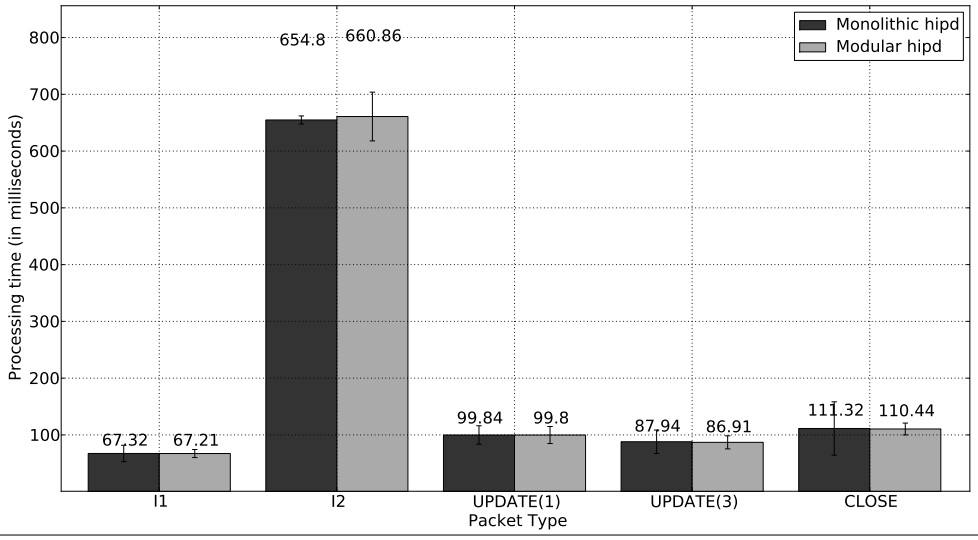

Figure 7.6 Average packet processing time on Linksys WRT160NL (responder) with standard deviation

The average processing times for the packet handling on the Linksys WRT160NL router are almost identical for both versions of `hipd`, as shown in Figure 7.6. Similar to the results on standard PCs, the I2 processing consumes significantly more time than the processing of other packets.

7.2 Discussion of Design Goals

The qualitative analysis of our work discusses the conformance of our solution with the design goals described in Section 5.1. We will refer the reader to the corresponding sections when appropriate.

7.2.1 Maintainability

We argue that modular protocol implementations using `libmod` are better maintainable in comparison to monolithic implementations, because the modular structure improves the three aspects of maintainability; namely extensibility, modifiability and testability.

The presented modular HIP implementation based on `libmod` is highly extensible, because additional functionality can be integrated in the shape of modules without any modification to the protocol core. Modules can extend or modify the current functionality without causing dependencies at the source code level. The ability to add and remove handle functions (see Section 5.3.3.1) makes protocol implementations with `libmod` adaptable to future needs.

In modular HIP and `libmod`, components (that are protocol core and modules) utilize well-defined interfaces for accessing functionality from each other. For this reason, changes inside components and without interface modifications, do not entail further changes in other components. Furthermore, modules can be replaced by other modules providing the same interface. This increases the modifiability of the considered building blocks.

The testability of modular HIP is also improved compared to the monolithic version, because the existing components can be tested independently. For instance, the functionality of the protocol core could be verified without the integration of modules. Furthermore, the functioning of different feature sets can be verified by enabling and disabling the relevant modules. Through clear separation, the modular structure offers intuitive mechanisms to create customized feature sets and cares for the compliance of module dependencies.

In contrast, the "precompiler approach" used in monolithic HIPL is not able to perform semantic checks. Each configuration option is handled isolated with the effect of removing specific code blocks or not. There is no relation between the available configuration options and hence all combinations of features sets are allowed without performing sanity checks.

7.2.2 Reusability

We have achieved reusability in two dimensions: Firstly, within `libmod` and modular HIP, functionalities used frequently are encapsulated into general functions, which can be specified for the actual task. We realized this architecture by the usage of the adapter and command design patterns. For instance, the modular packet handling mechanism 5.3.3.1 is a specialization of the general function registry (see Section 5.3.3) from `libmod`.

Secondly, `libmod` is a general library for modular protocol implementations and also applicable to other protocol implementations than HIPL. Hence the library as a whole is reusable.

7.2.3 Simplicity and Structuredness

The clear modular structure of protocol implementations with `libmod` reduces the implementation complexity due to the encapsulation of functionality into self-contained building blocks. Each building block can be realized independently from other components and thus with less complexity.

Furthermore, the clear separation of functionalities enables the definition of responsibilities. Multiple developers can independently work on different modules, because the module interaction and hence their integration is specified using well-defined interfaces.

7.2.4 Performance

The quantitative performance analysis in Section 7.1 shows that the modularization of `hipd` does not introduce performance overhead in the three examined measuring points build system, daemon startup and packet handling. Thus, we argue that modular protocol implementations are feasible with the concepts used in `libmod`.

8

Conclusion

Contemporary network protocols are adaptable to different usage scenarios and flexible in order to enable forward compatibility. Our problem analysis shows that monolithic implementations of adaptable and flexible protocols lack maintainability and cause considerable implementation complexity, due to the loose architecture in monolithic software. Thus, we argue that monolithic protocol implementations are not suitable nor sustainable for this category of protocols.

Our proposed concept for modular implementations of adaptable and flexible protocols strongly bases the implementational structure on the protocol specifications. While mandatory protocol features are an integral part of the protocol core, optional features are encapsulated in self-contained modules that can be added to the protocol core when necessary. The thus created modular structure allows the liberally extension of functionality. Furthermore, the consequent usage of well-defined interfaces for component interactions increases the modifiability of existing components.

We prove the feasibility of our approach for modular protocol implementations by realizing the protocol modularization library `libmod` and integrating it to the Host Identity Protocol for Linux as an example for flexible protocols. Our quantitative and qualitative analysis shows that the proposed modular concept significantly increases the software quality of protocol implementations without causing performance loss.

The accomplished implementation of the HIP daemon is considered as a proof-of-concept for the approach this thesis presents. For future work we suggest the further application of `libmod` to the remaining, still monolithic, parts of HIPL. Furthermore, our modularization concept can be applied to any other monolithic protocol implementation.

Bibliography

[1] Openwrt - linux-based firmware for embedded devices. `http://openwrt.org/`. [online, last visited: May 14, 2010].

[2] SCons - software construction tool. `http://scons.org/`. [online, last visited: May 14, 2010].

[3] Aalto University. HIPL User Manual. `http://infrahip.hiit.fi/hipl/manual/`. [online, last visited: May 19, 2010].

[4] Aalto University. Infrahip - open source implementation of hip. `http://infrahip.hiit.fi/`, 2010. [online, last visited: May 19, 2010].

[5] Alexander, C., Ishikawa, S., and Silverstein, M. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[6] Allman, M., Paxson, V., and Blanton, E. TCP Congestion Control. RFC 5681 (Draft Standard), IETF, Sept. 2009.

[7] Bovet, D., and Cesati, M. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[8] Braden, R. Requirements for internet hosts - communication layers. RFC 1122 (Standard), IETF, October 1989. Updated by RFCs 1349, 4379.

[9] Cerf, V., Dalal, Y., and Sunshine, C. Specification of Internet Transmission Control Program. RFC 675, IETF, Dec. 1974.

[10] Diffie, W., and Hellman, M. New directions in cryptography. *Information Theory, IEEE Transactions on 22*, 6 (November 1976), 644 – 654.

[11] Dijkstra, E. W. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective* (1982), 60 – 66.

[12] Fenton, N. E., and Pfleeger, S. L. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1998.

[13] Floyd, S., Arcia, A., Ros, D., and Iyengar, J. Adding Acknowledgement Congestion Control to TCP. RFC 5690 (Informational), IETF, Feb. 2010.

[14] Free Software Foundation. GNU make - build tool for the generation of executables and other non-source files. `http://www.gnu.org/software/make/`, 2006. [online, last visited: May 14, 2010].

[15] Free Software Foundation. GNU Libtool - The GNU Portable Library Tool. http://www.gnu.org/software/libtool/, 2008. [online, last visited: May 19, 2010].

[16] Free Software Foundation. Autoconf - GNU Project. http://www.gnu.org/software/autoconf/, 2009. [online, last visited: May 19, 2010].

[17] Free Software Foundation. Automake - GNU Project. http://www.gnu.org/software/automake/, 2010. [online, last visited: May 19, 2010].

[18] Free Software Foundation. GCC - Options that Control Optimization. http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, 2010. [online, last visited: May 17, 2010].

[19] Free Software Foundation. GCC, the GNU Compiler Collection. http://gcc.gnu.org/, 2010. [online, last visited: May 14, 2010].

[20] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns.* Addison-Wesley, 1995.

[21] Hüni, H., Johnson, R., and Engel, R. A framework for network protocol software. *SIGPLAN Not. 30*, 10 (1995), 358–369.

[22] IEEE. Standard glossary of software engineering terminology. *IEEE Std 610.12-1990* (1990), 1.

[23] ISO/IEC. 7498-1: Open systems interconnection - basic reference model, November 1994.

[24] Jokela, P., Moskowitz, R., and Nikander, P. Using the encapsulating security payload (esp) transport format with the host identity protocol (hip). RFC 5202 (Experimental), IETF, April 2008.

[25] Kitware, Inc. Cmake - cross-platform, open-source build system. http://www.cmake.org/. [online, last visited: May 19, 2010].

[26] Laganier, J., and Eggert, L. Host Identity Protocol (HIP) Rendezvous Extension. RFC 5204 (Experimental), IETF, Apr. 2008.

[27] Laganier, J., Koponen, T., and Eggert, L. Host Identity Protocol (HIP) Registration Extension. RFC 5203 (Experimental), IETF, Apr. 2008.

[28] Landsiedel, O. *Mechanisms, Models, and Tools for Flexible Protocol Development and Accurate Network Experimentation.* PhD thesis, RWTH Aachen University, Aachen, Germany, 2010.

[29] Lehtinen, J. C-pluff, a plug-in framework for c. http://www.c-pluff.org/, 2007. [online, last visited: May 19, 2010].

[30] Moskowitz, R., and Nikander, P. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), IETF, May 2006.

[31] Moskowitz, R., Nikander, P., Jokela, P., and Henderson, T. Host Identity Protocol. RFC 5201 (Experimental), IETF, Apr. 2008.

[32] Nagle, J. Congestion Control in IP/TCP Internetworks. RFC 896, IETF, Jan. 1984.

[33] Nikander, P., Henderson, T., Vogt, C., and Arkko, J. End-Host Mobility and Multihoming with the Host Identity Protocol. RFC 5206 (Experimental), IETF, Apr. 2008.

[34] Nokia Corporation. Maemo - software platform for smartphones. `http://maemo.org/`. [online, last visited: May 14, 2010].

[35] Parnas, D. L. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 279–287.

[36] Perlman, R. *Interconnections (2nd ed.): bridges, routers, switches, and internetworking protocols*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[37] Postel, J. Internet Protocol. RFC 791 (Standard), IETF, Sept. 1981. Updated by RFC 1349.

[38] Postel, J. Transmission Control Protocol. RFC 793 (Standard), IETF, Sept. 1981. Updated by RFCs 1122, 3168.

[39] Pressman, R. *Software Engineering: A Practitioner's Approach, 7th International edition*. McGraw-Hill, 2009.

[40] Provos, N. libevent - an event notification library. `http://www.monkey.org/~provos/libevent/`, 2010. [online, last visited: May 19, 2010].

[41] Python Software Foundation. Python Programming Language - Official Website. `http://www.python.org/`, 2010. [online, last visited: May 14, 2010].

[42] The OpenSSL Software Foundation. Openssl: The open source toolkit for ssl/tls. `http://www.openssl.org/`, 2010. [online, last visited: May 19, 2010].

[43] Verma, C. S., and Liu, L. Re-engineering legacy code with design patterns: A case study in mesh generation software.

[44] W3C. Extensible markup language (xml) 1.0 (fifth edition), w3c recommendation. `http://www.w3.org/TR/xml/`, 2008. [online, last visited: May 14, 2010].