CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack

Christoph Borchert[a] Daniel Lohmann[b] Olaf Spinczyk[a]
christoph.borchert@tu-dortmund.de lohmann@cs.fau.de
olaf.spinczyk@tu-dortmund.de
[a]Technische Universität Dortmund
[b]Friedrich-Alexander-Universität Erlangen-Nürnberg

ABSTRACT

Internet protocols are constantly gaining relevance for the domain of mobile and embedded systems. However, building complex network protocol stacks for small resource-constrained devices is more than just porting a reference implementation. Due to the cost pressure in this area especially the memory footprint has to be minimized. Therefore, embedded TCP/IP implementations tend to be statically configurable with respect to the concrete application scenario. This paper describes our software engineering approach for building CiAO/IP – a tailorable TCP/IP stack for small embedded systems, which pushes the limits of static configurability while retaining source code maintainability. Our evaluation results show that CiAO/IP thereby outperforms both *lwIP* and *uIP* in terms of code size (up to 90% less than *uIP*), throughput (up to 20% higher than *lwIP*), energy consumption (at least 40% lower than *uIP*) and, most importantly, tailorability.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*TCP/IP*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and Interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

AOP, Aspect-Oriented Programming, AspectC++, Operating Systems, Embedded Systems, TCP/IP, Internet Protocol, Network Protocol Stacks

1. INTRODUCTION

Communication has become an integral part of today's computer systems. Every personal computer is equipped with a built-in network interface and there is a clear trend towards smart (computerized) devices, which very often have the ability to interact with the Internet, such as modern TVs. Hence, small and, in the future, even smaller devices need to implement the TCP/IP protocol suite, which is the de-facto standard for data communication and by definition used in the Internet in order to achieve *interoperability*.

However, there is still a huge gap between vision and reality: Science and industry are thinking about *Service-Oriented Architectures (SOA) for embedded devices* [3], small devices being connected with the Internet using a lightweight middleware such as *Universal Plug'n'Play (UPnP)*[1] or its successor *Devices Profile for Web Services (DPWS)*[2]. Both are based on TCP/IP. On the other hand, the resource contraints of embedded microcontroller units often forbid to use the software-intensive Internet protocols, which in practice leads to hybrid systems and a "reinvention of the wheel".

Whether a full-featured TCP/IP stack is well suited for deeply embedded systems with tight resource constraints, has already been debated in great detail [11, 13]. The purpose of this paper is *not* to give the final answer. However, we can show *how* the TCP/IP protocol suite could be used in nearly any device, despite of the enormous diversity in hardware platforms and application requirements.

Today, many TCP/IP implementations are available. Most of them were designed as a component of a PC or server operating system – such as BSD, Linux, or Windows – and are quite resource intensive.

Common examples for TCP/IP stacks for resource-constrained embedded systems are *micro-IP (uIP)* and *lightweight-IP (lwIP)* [8]. uIP proved that a full TCP/IP implementation fits even into a small 8-bit microcontroller unit by leaving out all optional features, wheras lwIP provides better performance at the cost of higher memory consumption. It is remarkable that both, uIP and lwIP, stem from the same author and have been presented together in [8]. This underlines that a "one size fits all" TCP/IP implementation is not well suited for embedded systems. However, is a "two sizes fit all" solution so much better?

1.1 Contribution and Outline

In this paper we present the design approach of the CiAO/IP[3] stack and a comparison of CiAO/IP with *uIP* and *lwIP*. Our approach is to construct CiAO/IP as a highly configurable *Software Product Line* [26] that provides for very tailored stack implementations for embedded applications. In order to keep the highly configurable code maintainable, *Aspect-Oriented Software Development* [18] has been exercised not only for the implementation but already during the design phase.

In summary, our contribution is to show that *applying* aspect-oriented design and implementation techniques to a new domain – networked embedded systems – is promising from a code mainte-

MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.

[1] `http://www.upnp.org/`
[2] `http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01/`
[3] The source code is available under the terms of the GNU GPL at: `http://ess.cs.tu-dortmund.de/Software/CiAO-IP`

nance perspective without incurring energy and performance overhead.

Moreover, the fine-grained configurability of CiAO/IP allows developers to tailor the IP stack at compile time to their actual usage scenario, so that memory footprint, performance, and energy consumption are even improved over the state-of-the-art. We regard that knowledge of *how to build* network stacks for small-footprint devices as a useful contribution that deserves to be communicated more broadly.

The outline of this paper is as follows:

- We show that configurability is an important requirement on system software for networked embedded systems, which is not sufficiently fulfilled by other state-of-the-art IP stacks in this domain (Section 2). Despite the tension between source code maintainability and fine-grained configurability, the paper shows that CiAO/IP achieves *both* objectives (Section 6).
- We apply our unique design methodology, which is based on several years of experience with the aspect-oriented operating system CiAO[4] [22], to the domain of the Internet Protocol stack. Starting from requirements with explicit variability we step-by-step derive an aspect-oriented software design. Reusable programming idioms help to turn the design into code (Section 3 and Section 4).
- We show how the methodology led to our CiAO/IP design and implementation. Thereby, we document the structure of CiAO/IP, which is the first aspect-oriented TCP/IP stack we are aware of.
- By comparing CiAO/IP with *uIP* and *lwIP* we prove the implementation's scalability (8 to 64-bit systems) and high configurability (in terms of functional features), which we could achieve without any drawbacks in terms of code size, performance, and energy consumption (Section 5).

2. CONFIGURABILITY OF IP STACKS

The development of an IP stack for small mobile and embedded devices is more than just porting a reference implementation.

2.1 Diversity of Platforms and Requirements

The first challenge is that two embedded systems rarely look the same. The microcontroller market is heterogeneous and offers many entirely different devices. Simple control tasks can be performed by small 8-bit CPUs, which are still dominating the market in terms of shipped units [34]. At the other end of the spectrum, complex signal-processing often requires high-performance 32-bit or even 64-bit devices. The memory sizes are as diverse as their layouts, but share the commonality that they define hard constraints that a software developer must fulfill. When it comes down to networking, the number of possible interfaces seems to be vast. Both wired and wireless technologies are available in many different kinds, such as Ethernet, wireless LAN, GSM, and IEEE 802.15.x.

System software that should support and cope with all this variability has to be exceptionally *flexible* and *portable*, and must be *minimalistic* in order to fit into the tiny memories of even the most miniaturized devices – often only a few KiB of RAM!

The second challenge is that the requirements on system software vary a lot in different embedded systems. Depending on the application scenario, the operating system has to support best-effort scheduling for concurrent tasks, while other scenarios require strict spacial and temporal isolation of each task in order to meet certain dependability properties. The requirements on network stacks are diverse as well: Multimedia applications, such as *Voice over IP*, call for low latencies and guaranteed quality of service. Video streaming, on the other hand, requires high throughput and is not tied to low latency constraints. Furthermore, systems differ in the protocols they use. Information exchange such as E-Mail requires a reliable transport protocol that confirms data reception, whereas best-effort data delivery is sufficient for sensor applications. In short: the functionality required from the network stack strongly depends on the concrete application.

The big challenge from a software engineer's point of view is to overcome this multi-dimensional diversity and to design software that fits into all those situations in an optimal way. Static (compile-time) *configurability* is the key to support a broad variety of hardware platforms and application requirements without sacrificing efficiency.

The CiAO/IP stack aims at a very high degree of configurability to tailor the stack for the chosen hardware platform and application scenario. The goal is to be able to leave everything off that is *not* needed – in order to achieve minimal resource consumption.

2.2 State of the Art

Table 1 gives an overview on the features provided by our implementation of CiAO/IP and those of uIP and lwIP[5] [8].

A configurable feature that can be omitted if not needed, is denoted by ✓. This kind of feature contributes to functional scalability. A fixed feature, which is always present and not removable by the end-user, hinders scalability and is denoted by ∗. Unimplemented features of each IP stack are shown as whitespace.

uIP has by far the most reduced feature set. It supports solely a single networking interface, manages a single buffer for exactly one packet and provides no support for an operating system. The variability of uIP is very limited, too, for instance the protocols ICMP and TCP are fixed[6]. Furthermore, uIP does not implement a sliding window for TCP, so that poor throughput can be expected. Thus, uIP is only suited for application scenarios on deeply embedded devices that require TCP connectivity .

lwIP provides rich functionality in terms of features. As shown in Table 1, lwIP implements almost every feature that we consider for this comparison. Nevertheless, the configurability of networking protocols is coarse-grained. For example, the feature TCP includes client *and* server functionality (often only one side is needed), a sliding window (again, only for *both* Tx and Rx) and a bunch of common optimization algorithms, such as Silly Window Syndrome (SWS) avoidance, Round-Trip Time estimation and congestion control. None of these optional protocol features is actually optional in the lwIP implementation – TCP can only be included and excluded at a whole. Because of this fixed set of rich functionality, which might be welcome on powerful devices, lwIP does not scale to devices with tighter resource constraints. The large number of fixed features sacrifices scalability.

The CiAO/IP implementation differs substantially from the preceding ones: In CiAO/IP *every* feature is optional and, thus, configurable by the application developer. We abstain from fixed features in favor of scalability. Thereby, the developer has full control and does not have to pay for features that are not needed. The overall functionality of CiAO/IP is comparable to lwIP, but configurability is much more fine-grained.

[4] CiAO is Aspect-Oriented

[5] versions 1.0 and 1.32 respectively

[6] Contiki 2.5 (`http://www.contiki-os.org/`) includes an updated version of uIP that provides configurability of TCP and ICMP by `#ifdef` directives.

Feature	uIP	lwIP	CiAO/IP
Multiple Networking Interfaces		*	✓
Checksum Offloading per Interface			✓
Multiple Connections (Concurrency)	*	*	✓
Buffers per Connection (Isolation)			✓
Operating System Support		✓	✓
IPv4	✓	✓	✓
IPv4 Tx	*	*	✓
IPv4 Rx	*	*	✓
IPv4 Fragment Reassembly	✓	✓	✓
IPv4 Fragmentation		✓	
IPv6	(✓)[a]	(✓)[a]	(✓)[b]
ARP	✓	✓	✓
ARP Reply	*	*	✓
ARP Request	*	*	✓
ARP Cache Timeout	*	*	✓
Static ARP Cache Entries			✓
ICMP	*	✓	✓
UDP	✓	✓	✓
UDP Tx	*	*	✓
UDP Rx	*	*	✓
UDP Checksumming	✓	✓	✓
TCP	*	✓	✓
Client (Connect)	✓	*	✓
Server (Listen)	*	*	✓
Sliding Window (Tx)		*	✓
Sliding Window (Rx)		*	✓
Avoid Silly Window Syndrome (Tx)		*	✓
Avoid Silly Window Syndrome (Rx)		*	✓
Round-Trip Time Estimation		*	✓
Congestion Control (Slow-Start)		*	✓
TCP Urgent Data	✓	*	
Limit Excessive Retransmissions	*	*	✓
MSS Option	*	*	✓
Timestamp Option		✓	

[a] provided by a different code base, no dual stack
[b] under development

Table 1: Features provided by uIP, lwIP and CiAO/IP

2.3 A Software Engineering Challenge

Traditionally, configurability in system software has been expressed by the use of the C preprocessor. It supports textual substitution by macro expansion and conditional compilation by means of `#ifdef` directives. The preprocessor transforms the source code before it is parsed by the C compiler. A typical idiom is to represent all configurable features as macros in common header file, which controls the source code adaptation process of the system.

Expressing software variability in that way has been heavily criticized as error-prone, unreadable and unmaintainable [29]. A recent study of Linux kernel code shows that `#ifdef`-based configuration led to hundreds of bugs [33]. There are two main problems:

1. Dependencies between features are represented on the source-code level: This leads to a mix of abstraction levels within the code, because for each particular feature the *high-level* requirements and dependencies become intermixed with *low-level* implementation details.

2. The implementation of features often "crosscuts" various modules (functions, files, etc.): For many configurable features, the implementation is "scattered" over many modules and "tangled" with the implementation of other features. The consequence is that configurable features lead to code that is bloated with lots of macros and `#ifdef` directives. The whole code base becomes hard to understand and maintain.

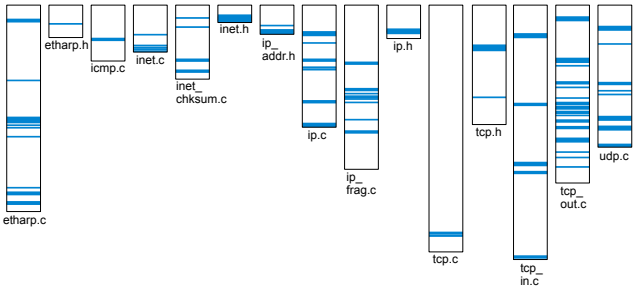


Figure 1: Scattered implementation of byte order conversion in lwIP. Each bar represents a source code file of lwIP; a horizontal line indicates the use of a conversion macro (`htons, ntohs,...`) in the respective source line.

Figure 1 illustrates these problems on the example of byte ordering in lwIP: In lwIP (as in uIP, Linux, and BSD) the conversion of 16-bit and 32-bit words to the network byte order is performed by dedicated preprocessor macros (`htons, ntohs, htonl, ntohl`). Access to network protocol headers is coupled to these macros – their use is compulsory to ensure data correctness. Hence, these macros appear 210 times in 15 files of the analyzed lwIP configuration. This implementation of the byte order conversion is tedious and error-prone, as a single forgotten macro call can already lead to data corruption in terms of swapped bytes.

The conclusion is that traditional preprocessor-based configuration does not scale: Software with lots of configurable features almost automatically leads to a maintenance nightmare [33]. This is a software engineering challenge. The problems can only be solved by using a configurability-oriented design approach combined with suitable programming language abstractions.

3. DESIGN APPROACH

The CiAO/IP stack is a generic platform for IP networking software. It supports the automatic *derivation* of custom IP implementations that fulfill application-specific constraints. An application developer just has to specify the relevant constraints by selecting software features in a graphical user interface. After that, he instantly gets a highly optimized IP implementation that meets his requirements. Even architectural properties, such as simultaneous support for different link layers, are configurable and do not cause any overhead if not needed.

CiAO/IP has been developed as an aspect-oriented software product line. A software product line is "a *family* of systems in a domain, rather than a single system" [15]. Due to systematic *reuse*, the software is better tested and, thus, less error-prone. From an economic perspective, the effort for maintaining a software product-line is much less than for several individual products.

One of the key ideas from the product-line engineering community is to strictly separate the so called *problem space* from the *solution space*. This addresses and solves the first problem identified in Section 2.3, namely the mix of abstraction levels in the source code. Typically a problem-space model of a product line describes the common and variable features and their dependencies in an abstract and problem-oriented manner. The problem-space model is independent from the product line's implementation. The solution space is the set of code artifacts that form the implementation.

Aspect-Oriented Programming is a paradigm that provides pro-

gramming language means to modularize the implementation of crosscutting concerns [18]. This allows us to avoid the second problem from Section 2.3. As a result the highly configurable code of CiAO/IP is completely free of `#ifdef` directives.

In the following, we will first explain feature modeling, which is the methodology that we used to model the problem space of the CiAO/IP product line. Then we briefly introduce the concept of Aspect-Oriented Programming.

3.1 Feature Modeling

Feature modeling [15] is a mature technique for formalizing variability. The idea is to identify and document requirements in terms of *features*. A feature is a requirement of a specification or an informal demand of a stakeholder. Each feature can be either mandatory or optional – the latter case is where variability comes into play. Furthermore, features may have restrictions and dependencies among each other. A feature model encompasses all requirements, both mandatory and optional, that are imposed for the product being developed. The success of a fine-grained configurable software product-line relies mainly on high-quality feature models: It is crucial to have detailed information about the expected variability in advance in order to guide the subsequent design and implementation process.

We believe that the methodology of feature modeling is well suited for analyzing the complex requirements that IP networking software must fulfill. As a starting point for a feature model of the Internet TCP/IP protocol suite, we use the official specification published by the Internet Engineering Task Force (IETF). There are several RFC documents which define requirements for each protocol. In RFC1122 [6], the most important requirements for Internet hosts are summed up and categorized into three different kinds: *MUST*, *SHOULD*, and *MAY*. These capitalized verbs determine the significance of each particular requirement; in terms of a software product line *MUST* requirements map to mandatory features, whereas *SHOULD* and *MAY* requirements describe optional features.

Figure 2 shows a simplified feature diagram of the Internet Protocol. A feature diagram graphically represents the variability of a feature model by using a tree structure. Each node represents a feature that depends on all of its ancestors. A filled circle at the lower end of an edge indicates a mandatory feature (c.f. RFC *MUST*), whereas a nonfilled circle describes an optional one (c.f. RFC *SHOULD* or *MAY*). Cumulative features, of which a least one must be present, are grouped together by a filled arch at the upper ends of their edges. As shown in Figure 2, the Internet Protocol can be implemented as version 4 (IPv4) as well as 6 (IPv6). For each version, there is a distinction between sending and receiving, and each particular version requires at least one of them. Additionally, the Internet Protocol is extensible via *IPv4/IPv6 Options*, which are in fact optional features.

The development of feature models for each protocol of the TCP/IP protocol suite has been necessary for the sequel of this paper. A more complex example is given in Figure 3, which contains a still heavily simplified feature diagram of TCP, the most complex protocol we dealt with.

Compared to the RFC, the feature diagram is far more fine-grained. For instance, the feature of a *Sliding Window* is considered optional, because the minimal size of that window is not specified, which makes it effectively optional. A window size of exactly one segment is equivalent to a nonexisting Sliding Window.

Hence, feature modeling is a challenging discipline, which relies on the knowledge and farsightedness of experts in the analyzed domain in order to grasp all hidden features, which are not explicitly stated in the specification, but relevant to stakeholders. On the other hand, no expertise in programming languages or specific paradigms is needed. This part can be left to solution space experts.

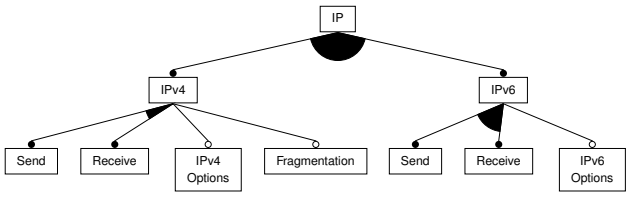


Figure 2: Feature Diagram of the Internet Protocol

3.2 Aspect-Oriented Programming

Traditionally, the developer of an Internet Protocol stack decides upfront which features to implement, so that a particular system consists of a fixed subset of the RFC1122 requirements. Our idea is to postpone this decision as far as possible to preserve this freedom for the actual user of the protocol stack.

The design of a software architecture that adheres to the variability of feature models is a nontrivial task. We propose *Aspect-Oriented Programming (AOP)* [18] as a solution for this problem in order to avoid the already discussed pitfalls of the C preprocessor. AOP supports the decomposition of a system into orthogonal modules by featuring *implicit invocation*. In AOP, an *aspect* contains one or more pieces of *advice*, that intercept the control flow and extend the underlying types. A piece of advice targets several *join points*, which are either locations in the dynamic control flow or part of the static program structure, where arbitrary code can be invoked. Join points are described declaratively via *pointcut expressions* in a textual form.

AspectC++ [30], which has been developed by our group over the last 10 years, extends the C++ programming language by AOP mechanisms. It consists of an *aspect weaver* that processes ordinary C++ code and aspect code, and weaves the latter type of code into the C++ files at the relevant locations (i.e., join points). Thus, the advice code is *inlined* into existing C++ code and produces no overhead compared to an implementation by hand (see [23]). In constrast to C preprocessor directives, the AspectC++ language elements are fully integrated into the syntax and semantics of C++.

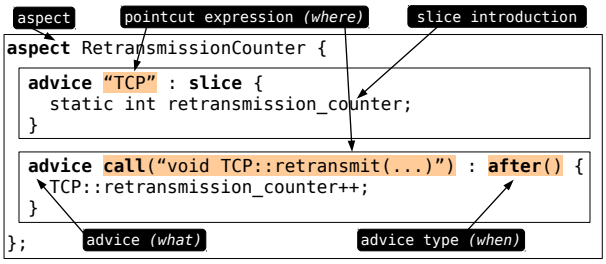


Figure 4: Syntax of AspectC++

Figure 4 outlines the essential syntactic elements of AspectC++ by taking the example of a retransmission counter for TCP. The given aspect implements the counter in a modular way by two pieces of advice. The first piece of advice extends the class `TCP` by an `int` member to store the counter's value. This way, the static program structure is modified by introducing additional class members, which is called a *slice introduction*. The second piece of advice targets the function `TCP::retransmit(...)` and implicitly

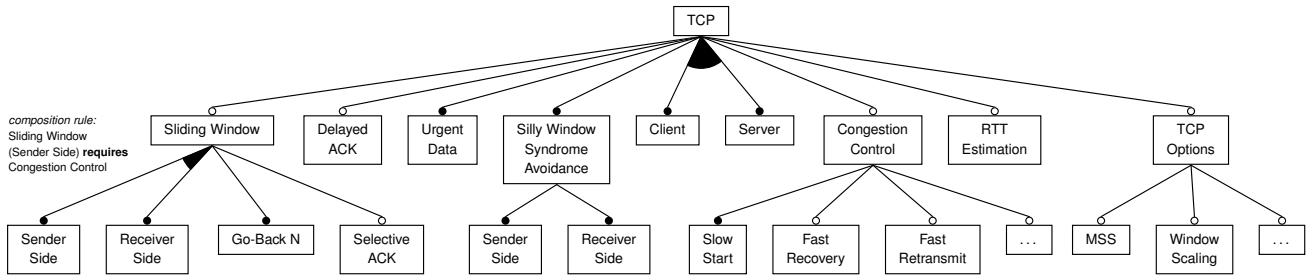


Figure 3: Simplified Feature Diagram of TCP

increases the counter *after* each *call* to that function. The ellipsis of the function's arguments in the pointcut expression ensures that in case of an overloaded function all overloads are matched regardless their arguments.

Furthermore, multiple pointcut expressions can be combined using the algebra of sets. Besides *after* advice, *before* and *around* (the latter replaces the original behavior of the join point) advice can be used to specify when advice code shall be invoked.

In the following section, we show that AOP provides a rich toolkit for the design and implementation of a highly modular networking stack. Even crosscutting features, which are conceptually scattered over many different software modules, can be developed as separate modules by using AOP. We provide a design method for deriving a mapping of features to aspects and ordinary C++ classes and answer the emerging question: Which features should be turned into aspects?

4. INSIDE THE IP STACK

The previous section outlined basic methods we use for the design and implementation of fine-grained configurable networking software. Feature modeling reveals all features that have to be considered during the following design process. Before the actual design of the software system, the interactions of all features and their impact on the system must be analyzed. The system designer needs to gather this information in order to design the fundamental software architecture. Section 4.1 covers the necessary analysis that results in a reference software architecture, which is elaborated in Section 4.4. In between, general design principles and idioms are introduced that guide the design process.

4.1 Concern Impact Analysis

Concern Impact Analysis, which originates from the design of automotive operating systems [21], is a method for the investigation of interrelation between features. Both, concerns that are part of the system's specification (e.g., RFCs) and internal concerns serve as input for the analysis. Internal concerns are inherent properties of the system that are not documented in the specification. For example, the network byte order is not discussed in RFC1122 although it is crucial for interoperability. The starting point for the following analysis is a set of both explicit concerns (i.e., features) and internal concerns. Since it is hardly possible to be aware of all internal concerns in advance, missing ones can be added iteratively. Networking software in general consists of

1. an *API*, that provides accessible system services to applications,

2. *connection state* to distinguish between different connections and their actual usage,

	Protocols				Options			Internal	
	ARP	IP	UDP	TCP	UDP Checksum	TCP Sliding Window	TCP MSS Option	Checksum Algorithm	Byte Order
`send()`		⊕	⊛	⊛	◐	●	◐		◐
`receive()`		⊕	⊛	⊛	◑	●	◑		◑
`connect()`				⊕		◐	◐		
`listen()`				⊕		◑	◑		
`close()`				⊕					
`bind()`			⊕						
`unbind()`			⊕						
`set_ipv4_addr()`		⊕							◐
`set_dst_port()`			⊕	⊕					◐
`set_src_port()`			⊕	⊕					◐
Packet Buffers	⊛	⊛	⊛	⊛		⊛			
ARP Cache	⊕								⊛
IP Addresses		⊕							⊛
Port Numbers			⊕	⊕					⊛
TCP States				⊕		⊛	⊛		
Receive IRQ	◑	◑	◑	◑					
Timeout	◑			◑		◑			
TCP/UDP ⇄ IP					◑			◑	
IP ⇄ Data Link	◐							◐	
Initialization		◑							

Table 2: Impact of configurable concerns on configurable IP networking software. Impact of concerns (columns) on the API, internal states, and events (rows).

3. and external events or rather internal *state transitions* that alter the system's behavior.

These three ingredients are of special interest because they emphasize places where various configurable concerns can interfere. An arrangement of all identified configurable concerns over these three ingredients leads to a comprehensive *crosscut table matrix* that reveals critical join points.

Table 2 shows an excerpt of the crosscut table matrix that we prepared for IP networking software. The columns list a few of the configurable concerns, which are themselves grouped into sets of networking *protocols*, *options* that extend protocols, and *internal* concerns. The aforementioned three categories (API, state, and transitions) form the rows of Table 2. Each category is separated by a horizontal line and covers a few relevant examples.

After the columns and rows are identified, the crosscut table matrix has to be populated. The influence of each concern on each item (row) is analyzed and classified into mainly three different types of impact:

1. **Introduction** of system services or states (static structure). This kind of impact implies the introduction of basic functionality that is added to the system and typically becomes visible

in the system's API. This fundamental type of influence provides join points for other concerns, denoted by a ⊕ sign in the crosscut table matrix.

2. **Extension** of existing system services or states (static structure). Some concerns modify already existing API services or existing system states. This less invasive type of impact, compared to the preceding one, is visually denoted by a ⊛.

3. **Modification** of the run-time behavior (dynamic structure). Concerns may have impact on the execution of API functions, the occurrence of external events or internal state transitions. The actual modification can take place before / after / around the event, as denoted by ◐ / ◑ / ●.

The crosscut table matrix provides a comprehensive overview of the concerns present in the system and their (subtle) relationships. It enables the developer to make informed decisions in the subsequent architecture design process – in which the concerns ultimately have to be mapped to classes and aspects.

4.2 Mapping to Classes and Aspects

As a driver for this process, the crosscut table matrix can be read in two directions:

A **horizontal analysis** focuses on a technical element, that is a state or service provided by the API for which it yields the contributing and dependent concerns. Once a state or service is introduced by a concern, each modifying concern (denoted by a ⊛) explicitly "*uses*" the introducing one. For example, the concerns *UDP* and *TCP* directly "use" *IP* – they built on the respective `send()` and `receive()` primitives, as observable in the first two rows of Table 2. With this information, the system designer is able to develop a *concern hierarchy* [21] by arranging the concerns according to their "uses" relationships. Figure 5 summarizes the concern hierarchy that encompasses the subset of concerns used in the preceding crosscut table matrix[7].

A **vertical analysis** provides more detailed information about a conceptual concern. Basic concerns can be identified by at least one introduction (⊕), for instance *ARP* in the first column. A new state, namely the *ARP Cache*, is introduced by this concern. Crosscutting concerns can easily be spotted by encountering multiple modifications (◐ / ◑ / ●) for a given concern, such as the *Byte Order* in the last column, which crosscuts several API functions and states.

The information from both analyses can then be used to achieve an initial mapping of concerns to classes and aspects by some simple rules of thumb:

- A *basic concern*, such as *ARP*, *IP*, *UDP*, and *TCP*, is mapped to a class.
- If a basic concern "uses" another basic concern, it becomes a derived class. For example, *UDP* and *TCP* both will be derived from *IP* (see Figure 5).
- A crosscutting concern, such as *Byte Order* or *Checksum Algorithm*, becomes an aspect.
- A configuration option, such as *TCP Sliding Window* or *UDP Checksum*, is also often crosscutting and thus becomes an aspect.

Of course, these rules of thumb only provide an initial software structure, that has to be refined in the subsequent design process, in which usually additional aspects and classes are employed to implement the binding and configurability of the identified concerns. In order to ease the following fine-grained design process, we propose design principles that aid the system designer to make "right" design decisions.

[7]Once again, all concerns for IP networking software do not fit into a single comprehensive diagram.

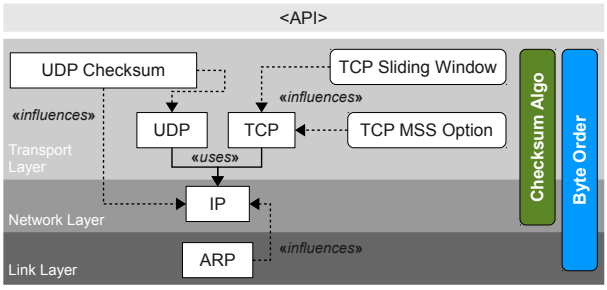


Figure 5: Layered concern hierarchy. Rectangles denote basic concerns, rounded boxes indicate crosscutting concerns. Concerns have "*uses*" and "*influence*" relationships, the latter denoted by a dashed arrow.

4.3 Design Principles

Based on our experience with the automotive operating system CiAO [22, 32], the following design process is guided by four reusable principles that lead to an aspect-aware system design. The corresponding AspectC++ idioms are beneficial for a clean system design and fine-grained configurability that does not induce run-time overhead.

4.3.1 Loose Coupling

The principle of *loose coupling* describes the relation between two software modules. Consider a networking device driver and the IP networking protocol implementation. Both modules should be exchangeable – the device driver should work with different protocol stacks and a protocol stack should be able to use various device drivers. Using AOP, loose coupling between these modules can be established easily:

The device driver exposes an *explicit join point*, which is an empty function without arguments named `ready()`, that is called after the device driver is initialized. Aspects can hook into this function and bind the device driver module to the IP stack on the initialization event (see last row of Table 2). The *binding aspect* is a recurring aspect role whenever several software modules, possibly developed by independent working groups, need to be integrated in a noninvasive manner.

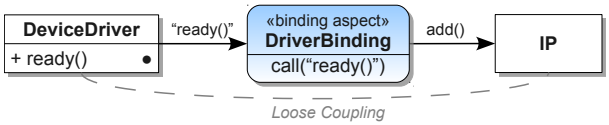


Figure 6: The principle of *loose coupling*. A *binding aspect* establishes a relation between two otherwise unrelated modules. The invocation of `ready()` is translated to an `add()` call, which registers the `DeviceDriver` at the `IP` stack.

4.3.2 Minimal Extensions

The principle of *minimal extensions* aims at incremental system design. Software modules are designed to provide only the minimal functionality that is required for a working system – no more, no less. Additional features are only introduced by *extension aspects*.

For instance, the TCP module itself does neither support a sliding window for transmitting nor for receiving. In Section 4.2, these features were already identified as crosscutting concerns and mapped to aspects. Thus, the sliding window for transmitting and for receiving are both designed as minimal extensions to the TCP module. As shown in Figure 7, the sliding window crosscuts the module TCP as well as both receive and transmit buffering.

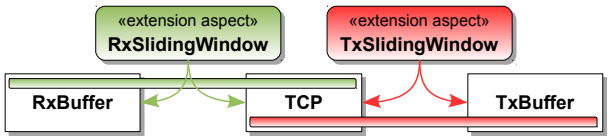


Figure 7: The principle of *minimal extensions*. *Extension aspects* introduce optional concerns by means of *slices*. Existing software modules are extended by additional member variables and functions in order to fulfill their duty.

4.3.3 Visible Transitions

The aspect-aware design enables the substitution of internal system policies. Thus, all relevant transitions need to be accessible by aspects. The system designer has to ensure that control-flow transitions are visible as unambiguous join points, that are preferably distinguishable at compile time. A fine-grained class hierarchy enclosed in expressive C++ namespaces provides a rich set of join points in the sense of *visible transitions*.

Often, a highly crosscutting concern represents a system policy. For example, the byte order conversion, which is examined in the last column of Table 2, is an exchangeable policy. On machines that internally operate at the network byte order, no conversion has to be performed. Otherwise, a conversion of every multi-byte entry of each protocol header takes place. We designed all protocol header structures as C++ classes, whose attributes are exclusively accessed throughout `get()` and `set()` functions. Therefore, aspects can alter the result and arguments of these functions and perform the byte order conversion directly in place. Figure 8 shows such an aspect for `get()` functions of 16-bit words. The source code for 32-bit words and `set()` functions is similar. Other software modules do not have to be aware of network byte ordering at all, because this concern is entirely encapsulated by a single *policy aspect*.

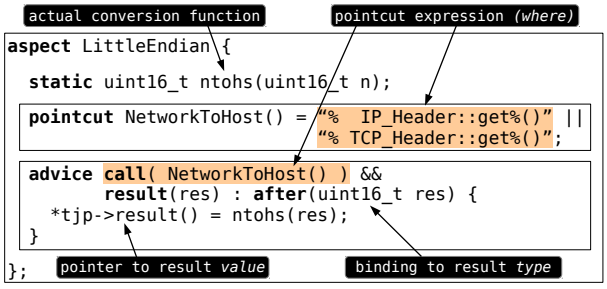


Figure 8: Byte order conversion by *policy aspect*. Conversion logic is encapsulated in the *static* function `ntohs`, which is implicitly invoked by the advice. The pointcut declaration describes *where* the advice shall take effect, that is all `get()` functions exposed by protocol header structures (*visible transitions*). `tjp` (this join point) provides access to context information, and `tjp->result()` yields a typed pointer to the return value.

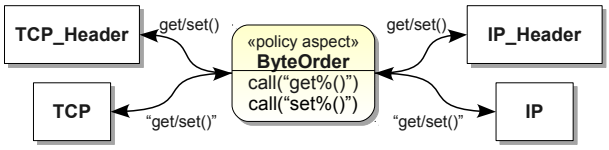


Figure 9: The principle of *visible transistions*. A fine-grained call structure makes system internal transitions explicit. Thus, system policies are designed to be exchangeable *policy aspects*, that alter the control-flow at relevant locations.

4.3.4 Upcall Dispatcher Hierarchy

A layered system defines a "uses" relation among its layers – typically top-down (see Figure 5). Each module is aware of all subordinate layers and directly depends on the ones it uses. The other way around, a module that is used by superordinate modules must not have knowledge of them. Otherwise, circular dependencies occur and configurability is lost.

Upcalls, which are invocations in the opposite direction of the "uses" relation, cause difficulties since they manifest in a tight coupling between layers due to circular dependencies. Consider an IP module that receives an IP packet and propagates it to superordinate UDP and TCP modules, which are in turn configurable and thus not always present. There may be even more subscribers for the content of an IP packet, such as a configurable ICMP module. The problematic nature of upcalls becomes even worse if multiple layers are involved. For example, an Ethernet packet, that is dispatched from the bottom layer up to the top layer, forces circular dependencies between all traversed layers.

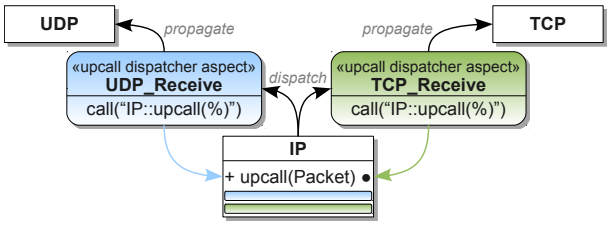


Figure 10: The principle of *upcall dispatcher hierarchy*. Bottom-up data propagation in a layered system is performed by independent *upcall dispatcher aspects*. Each aspect evaluates inherent dispatch conditions and eventually lifts data upwards.

The *upcall dispatcher hierarchy* circumvents this issue by strictly sticking to the inherent "uses" relation. Figure 10 illustrates this design principle based on the aforementioned example. The bottom-up propagation of data is implemented by explicit join points, which are empty functions (see Section 4.3.1), whose arguments contain the data to be propagated. In Figure 10, the module `IP` offers the public function `upcall(Packet)`, that is called by `IP` in order to dispatch a packet to superordinate modules. The actual dispatch is performed by independent *upcall dispatcher aspects*, which hook into the `upcall` function and evaluate whether the packet is to be accepted. Dispatch criteria are usually a check of the protocol's header identification field and a verification of the checksum. The dispatch process itself is distributed to several aspects, which provide configurability because they can be omitted easily.

A key benefit of upcall dispatcher aspects is that they are stackable. Consider a module `Ethernet`, that also provides an explicit join point `upcall(Ethernet_Frame)`. The upcall dispatching to superordinate modules, for instance `ARP` and `IP`, can be performed

in the same way. Thus, circular dependencies between layers are avoided and the already developed "uses" relation is enforced by this design principle.

4.4 Architectural Design Decisions

The efficiency of a system is vitally determined by crucial design decisions during the development process. Several concerns are not covered in the system's specification (i.e., RFCs), although they define fundamental properties of the software architecture. These *hidden concerns* have to be identified, evaluated and carefully designed in order to narrow the design space. Thus, the unnecessary variability is eliminated that is known in advance to bear no convincing benefits. This is the major software engineering challenge at the very end of the design phase, which is involved with the overall performance of the system.

4.4.1 Memory Management

The most important concern with regard to performance and scalability is memory management [7]. Each network packet has to be buffered in data memory, and access to it must be very fast. On typical embedded systems, memory and especially data memory (RAM) is a scarce resource, so that an economical utilization is necessary. Furthermore, on such systems, a dynamic memory management (heap) is often not economical and thus not available.

In general, there are two possible designs for the memory management of an IP stack:

1. **Linear buffers**, large enough to hold a contiguous packet.
2. **Linked lists** of small buffers, each buffer containing a fragment of the packet.

Both designs are reasonable and used in popular networking systems – Linux applies linear buffers whereas BSD favors linked lists. The advantage of linear buffers is speed, because a contiguous packet often fits into a single cache line. Extra effort has to be paid to packet construction in the space of linear buffers. The size of all applied protocol headers has to be known in advance and an appropriate amount of memory has to be reserved in the front of each buffer. Linked lists simplify packet construction, because arbitrary protocol headers can be linked in front of the list. This flexibility comes at the price of poor cache locality due to the scattering of list elements across the data memory.

In order to gain maximal efficiency, we chose to use linear buffers with two configurable sizes. This design decision takes into account that the distribution of packet sizes tends to be bimodal [25]. Large packets deliver the payload while small packets acknowledge successful reception. Because the size of the payload heavily depends on the application, we leave the configuration of the buffer sizes to the application programmer. We designed the memory management to be part of a parametrized API, that is implemented as generic C++ templates.

Thus, the application programmer can tune the memory utilization for each particular connection by instantiating a template class of the API. For example, the memory management parameters for high-throughput connections should satisfy the *bandwidth-delay product*. In general, the optimal amount of memory is approximately $throughput_{desired} \times delay_{end-to-end}$ for a sliding window.

4.4.2 Concurrency

An implicit requirement for IP networking software is the concurrent handling of multiple connections. Full concurrency is achieved by threads that are scheduled by an operating system. Thus, IP networking software could be run inside the context of threads or inside the operating system itself.

Our design decision is to reuse the runtime context of threads in order to implement concurrency. This approach has several advantages: A *prioritization* of threads is enforced, because the networking routines of high-priority threads are preferred over low-priority ones. This is essential for real-time systems, where hard timing deadlines have to be met. As a consequence, the feature of prioritized connection comes for free if thread priorities are supported by the underlying operating system.

The second important advantage of this design is that *isolation* between different connections can easily be established. Each connection can constitute separate pools of buffers in the context of the actual thread, so that no interference between concurrent connections can occur. The isolation of memory regions is especially interesting for safety-critical applications. On the other hand, shared buffers use the memory resources more efficiently. Therefore, CiAO/IP implements buffer sharing as a configurable feature.

4.4.3 Synchronization

A structured coordination between multiple threads running IP networking software and the underlying operating system is crucial for the performance of the system. The actual synchronization primitives depend highly on the services provided by the operating system, and are even nonexistent if no operating system is used. Therefore, synchronization is an exchangeable system policy.

According to the design principles (see Section 4.3), we make synchronization explicit by providing unambiguous join points for visible transitions. Thus, policy aspects implement synchronization by using operating system services. For example, we integrated our IP stack into the automotive operating system CiAO. CiAO itself provides *events* and *alarms* for synchronization, both specified by the AUTOSAR [2] standard. Policy aspects invoke these services implicitly and allow a noninvasive integration, so that neither the operating system nor the IP stack have to be modified. Therefore, the integration of this IP Stack into other operating systems is straightforward.

5. EVALUATION

To evaluate the CiAO/IP stack, we have implemented most features specified by the relevant RFC documents on a wide range of platforms:

8-bit: A BTnode[8] sensor network platform, which uses an Atmel ATmega128L *AVR* microcontroller with 128KiB ROM and 4KiB RAM combined with a sub 1 GHz CC1000 radio.

16-bit: A TI EZ430-Chronos[9] development system, which comes in a sports watch. It combines a *MSP430* microcontroller with 32KiB ROM, 4KiB RAM and a sub 1 GHz CC1101 radio device.

32-bit: An *IA-32* PC with Intel Core 2 Quad 6600 CPU, 4GiB RAM and an Intel 82566DM Gbit Ethernet adapter.

64-bit: Same as above, but running in *AMD64* mode.

Table 1 in Section 2 has already shown that all features of CiAO/IP that we identified as "optional" during the domain analysis became configurable in the implementation. This section will prove that this high degree of variability does not come a cost. CiAO/IP clearly outperforms uIP and lwIP in terms of memory consumption (Section 5.1), performance (Section 5.2), and energy efficiency (Section 5.3).

[8] `http://www.btnode.ethz.ch/`

[9] `http://processors.wiki.ti.com/index.php/EZ430-Chronos`

5.1 Memory Consumption

Memory is a scarce resource on embedded devices. This is true for both program memory (ROM) as well as data memory (RAM). For instance, the aforementioned 16-bit MSP430 platform contains only 32KiB ROM and 4KiB RAM, which software must not exceed. Especially system software that offers services to the actual application and is itself not self-contained, has to be exceptionally small and specifically designed for the embedded systems domain. A counterexample is the Linux[10] kernel, whose basic feature *TCP/IP Networking* increases the uncompressed kernel size by about 234KiB on IA-32. Hence, we focus our evaluation on uIP, lwIP and CiAO/IP.

The requirements on data memory are dominated by buffering. Network packets are inherently dynamic content that has to be stored in the RAM. Besides buffers, networking software stores connection state in the RAM, which requires an infinitesimal amount of memory compared to a buffer for a single IP packet. The buffer sizes of uIP, lwIP and CiAO/IP are statically configurable and, thus, their RAM consumption is comparable.

However, CiAO/IP is more flexible as it offers to configure whether buffers for a particular connection should be allocated on the data memory section, on the dynamic heap (if present), or even on the runtime stack of the current application. This feature is especially useful for temporary connections that are active only for a short period of time (for instance, during a software update), and, hence, consume data memory only within this period. Buffers, that are globally allocated on the data memory or the heap at system startup, constantly take RAM.

Besides data memory, an efficient usage of program memory is crucial. The requirements to ROM directly depend on the functionality of the software: More features lead to higher ROM consumption. In the sequel, we discuss the issue of program memory in detail. All following values are obtained by using the *gcc*[11] with optimization for size (-0s).

5.1.1 The Cost of TCP

TCP is certainly the most complex part of the Internet protocol suite. As a consequence, an implementation of TCP requires more program memory than other protocols. Nevertheless, TCP is essential for interoperability since many applications rely on it, for example web services. A realistic scenario for TCP also includes IP, and on IA-32 and AMD64 additionally Ethernet and ARP, since almost every personal computer comes with an Ethernet adapter.

Figure 11(a) shows the memory consumption of a TCP client in the described scenario. We use the minimal possible feature selections of uIP, lwIP and CiAO/IP to evaluate the minimal costs for a TCP client. The stacked bar diagram outlines the costs for the IP stacks themselves (lower section) and the cost for the application code that has to be added for a working TCP client (upper hatched section). This is relevant, as the required application code differs substantially between the IP stacks: uIP, for instance, does not perform retransmissions of lost TCP segments itself – it is up to the application logic to detect packet loss and to retransmit missing packets. For lwIP, we use its memory-efficient event-driven, nonsequential API that, however, also requires more effort on the application side than the UNIX-like sockets of CiAO/IP. For all IP stacks, the application code also handles synchronization with network devices and timer events. The necessary synchronization primitives are provided by the underlying CiAO operating system.

Figure 11(b) shows the memory consumption of a TCP server in the same scenario. For both client and server, the cost of TCP varies among the different architectures. It is notable that uIP has been heavily optimized for 8-bit architectures by using 8-bit data types and arithmetic whenever possible [9]. This specialization saves about 50% of the TCP code size on AVR and MSP430 compared to CiAO/IP, whereas it leads to an increased code size on IA-32 and AMD64.

5.1.2 The Cost of UDP

UDP is the second essential transport protocol of the Internet protocol suite. Compared to TCP, UDP is almost a null protocol, which is reflected by its low memory consumption. Consider a node of a sensor network that periodically transmits sensor measurements. For such a scenario, UDP is sufficient.

Figure 11(c) summarizes the cost of transmitting UDP packets without optional checksumming[12]. Since TCP is a mandatory feature of uIP, there is a high overhead if used for UDP only. lwIP consumes even more program memory due to architectural properties, such as flexible packet buffers and generic driver interfaces. CiAO/IP is by far the most efficient implementation with regard to UDP. On the average over all architectures, CiAO/IP requires only twelve percent of the memory that uIP respectively lwIP take.

For receiving UDP datagrams the results are similar. Figure 11(d) outlines our results. The application code (hatched section of top the bars) for receiving is larger than for transmitting, because synchronization with receive interrupts is involved.

5.1.3 Feature Scalability

We have measured the code size induced by each configurable feature: Starting with the minimal possible feature selection of the particular IP stack we have incremented the number of active features one by one. The results are depicted in Figure 12.

The minimal code size of uIP (see Figure 12(a)) is rather high due to the mandatory TCP feature, ranging from 4253 byte on MSP430 to 6481 byte on AMD64. When all features of uIP are active, the code size almost doubles.

lwIP scales from a minimum of 6836 byte on IA-32 up to a maximum of 38.7kB on AVR. This spread indicates scalability to some extend, but the absolute values are quite high. For example, the sum of all considered lwIP features on MSP430 occupies 30KiB of its 32KiB internal ROM (94%).

In contrast, CiAO/IP scales from a minimum of 789 byte on AMD64 up to a maximim of 20.8kB on AVR. In between, each configurable feature increases the memory consumption by a very small amount. The sharp spike in this diagram is caused by the TCP feature, which adds an exceptional large portion of code due to its complexity.

5.2 Performance

The aforementioned 8-bit and 16-bit hardware platforms are ill-suited for performance benchmarking, since their gross networking throughput is limited to 76.8 kbit/s respectively 500 kbit/s by the radio hardware. Moreover, the net throughput of payload (*goodput*) is primarily determined by the link quality[13] and timing of TCP retransmissions. To analyze the influence of the different IP stacks on high-speed networking performance, we therefore focus on the Gbit Ethernet adapter of the IA-32 platform:

We have connected two PCs (Intel Core 2 Quad 6600 CPU, 4GiB RAM, Intel 82566DM Gbit Ethernet) via a crossover cable to create

[10] version 2.6.33

[11] version 4.4 (MSP430/IA-32/AMD64), version 4.3 (AVR)

[12] The values for IA-32 and AMD64 also encompass the cost for Ethernet and ARP.

[13] We measured a goodput of 1161 bit/s for CiAO/IP and 680 bit/s for uIP on the average in the wireless setup described in Section 5.3. Both systems were configured with exactly equivalent features.

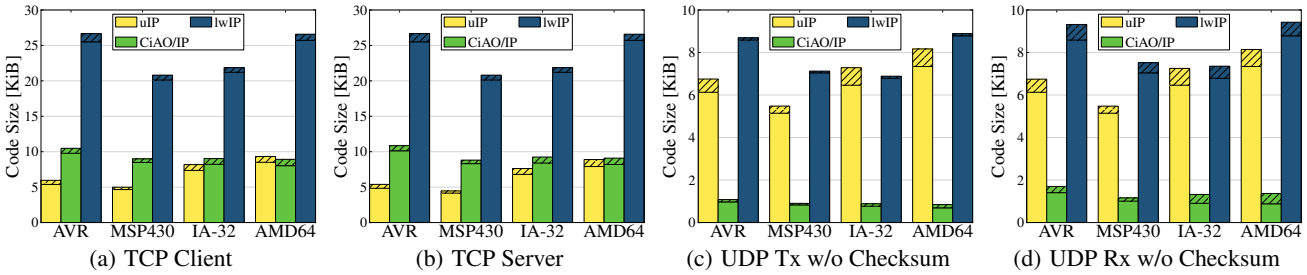
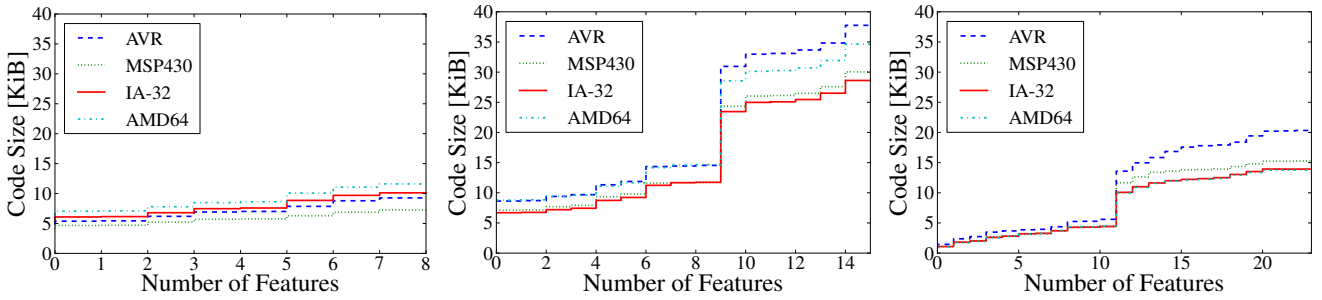


Figure 11: Code size (ROM) for common use cases. Each use case includes IPv4. On IA-32 and AMD64, Ethernet and ARP is also included to get a working system. Figures (a) - (d) show the minimal memory consumption of each particular IP stack for the desired functionality. The lower sections of the stacked bars constitute the code size of the IP stacks themselves. The upper hatched sections indicate the cost for the application code that has to be added for full operation. Hardware platforms range from 8-bit (AVR) to 64-bit (AMD64).



(a) **Feature increment of uIP:** TCP Server (IPv4) and ICMP, TCP Client, TCP Urgent Data, IPv4 Fragment Reassembly, UDP, UDP Checksumming, ARP, IPv4 Forwarding, NDP

(b) **Feature increment of lwIP:** UDP (IPv4), UDP Tx Checksumming, UDP Rx Checksumming, UDP Lite, IPv4 Forwarding, IPv4 Fragment Reassembly, IPv4 Fragmentation, ARP, ARP Queuing, VLAN, TCP, TCP Queuing, TCP Backlog, TCP Timestamps, ICMP, IGMP

(c) **Feature increment of CiAO/IP:** UDP Tx (IPv4), UDP Tx Cksum., UDP Rx (IPv4), UDP Rx Cksum., IPv4 Frag. Reass., Ethernet, Tx Cksum Offload., Rx Cksum Offload., ARP Reply, ARP Request, Static ARP Cache, ARP Timeout, TCP Client, TCP Server, Reset Invalid TCP Packets, Tx Slid. Window, Rx Slid. Window, SWS Tx, SWS Rx, MSS Opt., RTT Est., Cong. Control, Generic Buffer, Generic Rx Queue

Figure 12: Code size (ROM) vs. features. For each TCP/IP stack, the first measuring point shows the code size of the initial minimal feature selection. Each subsequent measuring point constitutes the addition of one more configurable feature. Thus, figures (a) - (c) outline an incremental feature selection and visually evince scalability.

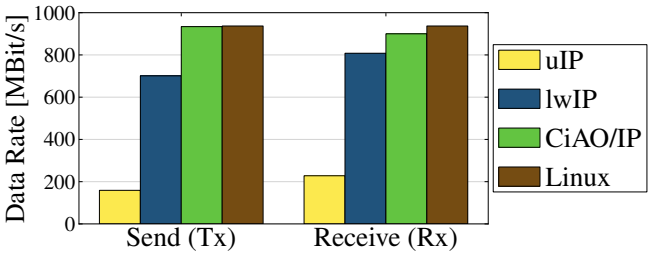


Figure 13: Goodput of a TCP connection. Depicted is the goodput between two local IA-32 machines, that are connected via gigabit Ethernet (crossover). One peer is always a Linux 2.6 box, whereas the machine under test runs the different IP stacks.

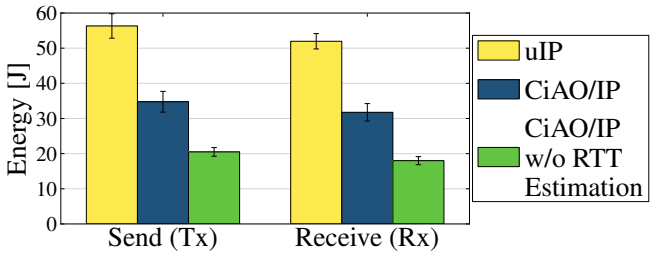


Figure 14: Energy consumption. Shown is the mean energy consumption and the sample standard deviation of unidirectional TCP data transfers (32 kB) between two wireless sensor networking nodes. *CiAO/IP w/o RTT Estimation* uses a fixed retransmission timeout (200 ms) for lost packets.

an actual loss-free environment. One box constantly runs Linux 2.6, whereas the other box runs the IP stack under test. We have established a single TCP connection between the two machines and measured the achieved goodput. Each IP stack is equipped with 100 KiB RAM for buffering and exactly equivalent feature configuration, except for uIP which solely supports a single buffer (1514 byte) due to the missing sliding window. For fairness, uIP is extended by *uip_split* [9] in order to reach maximal throughput. This uip_split extension avoids the default 250 ms delay between packet transmits. As uIP implements a stop-and-wait procedure, it otherwise would suffer from delayed acknowledgments of the Linux peer. Figure 13 shows the results of our measurements. We classify the results into sending and receiving, which describe the operational mode of the IP stack under test.

With Linux on both ends a maximal goodput of 937 Mbit/s is achieved, which here can be understood as the upper bound.

CiAO/IP, however, achieves almost the same transfer rates (934 Mbit/s for Tx, 900 Mbit/s for Rx).

lwIP clearly lacks behind. This is mostly caused by lwIP's linked packet buffer management (*pbufs*), which have to be copied into a contiguous memory region before delivery to the device driver. We intentionally do not use vectored I/O (scatter/gather) hardware accelerators that might overcome this design decision, since embedded devices generally do not provide such hardware support.

uIP is not competitive due to the missing sliding window, which results in a significant performance degradation.

5.3 Energy Consumption

Energy is probably the most scarce resource for battery-powered devices. We have chosen the aforementioned BTnode[14] sensor-net platform for determining the impact of IP stacks on energy consumption. A typical duty for sensor nodes is a firmware update: a unidirectional, reliable data transfer of a single file. Thus, our setup for energy measurements[15] consists of a TCP client, that transmits 32 kB of data to a listening TCP server.

On the link layer, we use the *B-MAC* [27] protocol for collision avoidance and preamble sampling with an interval of 25 ms (*Low Power Listening*). The packet size is limited to 255 byte, and each IP stack is equipped with 300 byte for buffering. Due to a quite noisy radio channel, which leads to a high percentage of corrupt packets (approximately 50%), we disabled the exponential backoff algorithm of every TCP implementation, which otherwise would slow down the retransmission of lost packets unnecessarily. Unfortunately, we did not get lwIP working under these harsh conditions, so that we had to omit lwIP in the following measurements.

Figure 14 presents the mean energy consumption and the sample standard deviation of uIP and CiAO/IP. For fairness, we did not activate the sliding window feature of CiAO/IP, because uIP does not support it and would otherwise be disadvantaged. To sum up, both IP stacks were configured with exactly equivalent features. Consequently, no congestion control was performed, because the slow-start algorithm can only be applied to a sliding window.

However, even without this beneficial feature CiAO/IP consumes significantly less energy: A uIP sender consumes a mean of 56.3 Joule, whereas CiAO/IP takes a mean of 34.8 Joule for the same task. This is caused by the coarse granularity of uIP's round-trip time estimation, which can only be multiple of 500 ms. In contrast, CiAO/IP performs this estimation much more accurate in units of 1 ms. Thus, lost packets are detected earlier (up to 499 ms per lost packet). To prove this claim, we deactivated the round-trip time estimation of CiAO/IP (an optional feature in CiAO/IP) and fixed the retransmission timeout to 200 ms. This way, the energy consumption of the sender could be further reduced to 20.5 Joule. These observations are also valid for the receiver, whose energy consumption is determined by the time the transfer takes to complete.

[14] AVR microcontroller (8-bit) running at 7.37 MHz, CC1000 transceiver operating at 868 MHz and 19.2 kbit/s

[15] *Hitex PowerScale* with *ACM probes* used for energy measurements

5.4 Summary of Results

CiAO/IP outperforms uIP and lwIP in almost all scenarios. The only exception is the TCP client/server, where uIP requires on the AVR/MSP430 platform only half the code size of CiAO/IP – which is exactly the scenario uIP has been optimized for [8]. However, even in this configuration uIP consumes significantly more energy (1.6x) and delivers a much smaller throughput (0.59x). In all other configurations, our aspect-oriented CiAO/IP stack outperforms the C implementations of uIP and lwIP notably with respect to code size, throughput, energy, and feature-wise scalability.

6. DISCUSSION

The main goal of this work is to achieve feature-wise scalability of the layered TCP/IP protocols by a software engineering approach for static configurability. As shown in the previous section, CiAO/IP offers fine-grained configurability and thereby achieves excellent results with respect to code size, throughput and energy. In the following, we discuss the pros and cons of our approach with respect to more "soft" properties, including maintainability, configurability, portability, and optimization.

6.1 Maintainability

The complexity and maintainability of some piece of software is generally hard to quantify. However, source-code metrics, such as lines of code (LOC) can serve as an indicator: The minimalistic uIP consists of 2796 LOC[16] whereas CiAO/IP consists of 4943 LOC and a comparable subset of lwIP[17] encompasses 11987 LOC. Clearly, more features correspond to more LOC, but in general aspect-oriented software consists of less LOC than its C counterparts, because crosscutting concerns are modularized into aspects and, thus, avoid scattered fragments of similar code.

A good example for this is the macro-based implementation of the byte order concern illustrated in Figure 1. As pointed out in Section 2.3, every header access has to be surrounded by one of the byte ordering macros (`htons, ntohs, ...`). This is a tedious and error-prone design rule. Our aspect-oriented CiAO/IP stack circumvents this issue by a much better separation of concerns: Byte ordering is a policy, hence we implement the byte order conversion as a modular policy aspect (see Section 4.3.3) that implicitly affects the `get()` and `set()` functions of all classes that describe network protocol headers. The respective aspect consists of a single source code file with less than 50 lines of code. This reduces complexity, as the byte order in CiAO/IP is correct by construction – it is not possible to create an invalid packet by a forgotten call of a conversion macro. Therefore, AOP contributes to maintainability of source code by modularization and separation of concerns.

On the other hand, AOP may impair the independent development of modules of the system [19]. Since aspects do not expose an explicit interface, the interaction of the system's modules can become complicated, because conceptually, each aspect can potentially affect each module [31]. To obviate this drawback, our aspect-oriented

[16] effective lines of code (excluding empty lines and comments), obtained with *cloc*: `http://cloc.sourceforge.net/`

[17] version 1.32, without IPv6, DNS and DHCP

design principles (see Section 4.3) are also applied as a means to make such interactions explicit.

6.2 Configurability

The excellent configurability we achieved in CiAO/IP is primarily the result of a software product-line development process that considers configurability as a fundamental design goal from the very beginning. However, eventually the individual features have to be implemented in a way that makes it possible to "leave them off" if not needed. The advantage of AOP here is *loose coupling* by the advice concept: Essentially, advice inverts the direction in which control-flow relationships are specified. This facilitates the self-integration of optional features into the control flows of the base system. Furthermore, advice-based binding is inherently loose – if the addressed join point is not present, the binding is silently dropped. This property is useful for the implementation of *interacting* optional features, which are difficult to tackle with other decomposition approaches [17]. In CiAO/IP we thereby could keep a textbook-like strict layering in the design and implementation of the IP stack without scarifying efficiency.

6.3 Portability

Loose coupling is also beneficial with respect to portability to other platforms. The aspect-based integration of our IP stack into the operating system makes it easy to use CiAO/IP with other systems or to use it without any operating system. We are currently working on implementations for eCos [24] and FreeRTOS [4].

From the perspective of application developers, switching to CiAO/IP is relatively easy as CiAO/IP offers the well-understood BSD socket concept and does not, like uIP and lwIP, require the application engineer to implement and maintain stack-specific state machines on the application level.

6.4 Optimization

The main benefit of the high configurability offered by CiAO/IP is optimization for *nonfunctional properties*, such as memory footprint, throughput, and energy consumption. This is done by first activating only those network protocols that meet the *functional* requirements depending on the actual field of application, for example TCP and IPv4. Other protocols should be disabled. This way, the design space of valid feature selections is significantly narrowed. As outlined in Table 1, several optional features do not provide basic networking protocols by themselves, and, thus still remain *open* for this particular feature selection. The assignment of these open features, for instance the TCP sliding window and round-trip time estimation to either *on* or *off*, leads to a *design space exploration*.

We developed the *feedback approach* [28] for statistical reasoning on nonfunctional properties and assignment of open features. Information about previously configured feature selections have to be captured by run-time tests and the results feed into a database. Thus, nonfunctional properties for further feature selections that are not yet evaluated can be estimated, and a testing set for validation is generated automatically. This is a semi-automated process that relies on testing of specific feature selections until the database is saturated.

7. RELATED WORK

We originally developed our aspect-aware design method as well CiAO/IP for our CiAO operating system family for deeply embedded devices [22, 21]. CiAO implements the automotive AUTOSAR OS standard [2], a rapidly growing domain with respect to “mobile Internet” applications. Other operating system projects that aim to bridge deeply embedded systems and the Internet world are mostly from the domain of sensor networks: Recent versions of *TinyOS* [20] provide the BLIP IPv6 stack, which also provides configurability of UDP and TCP and is implemented with the component model offered by the NesC language [12]. BLIP has the distinction of operating solely at IPv6. All other systems we are aware of are based on either uIP or lwIP, both of which clearly dominate the domain: *Contiki* [10] uses uIP; *eCos* [24] supports lwIP and, alternatively, a BSD-derived stack; *FreeRTOS* [4] allows developers to choose between uIP (called FreeTCPIP) and, again, lwIP.

AOP as a modularization approach for configurable system software has also been applied in the domain of middleware [35, 14] and embedded databases [16]. Other modularization approaches in the domain are too numerous to be discussed here. However, several researchers favor *feature-oriented programming* [5] – and even the C preprocessor is experiencing a renaissance by the provisioning of better tool support towards a “virtual separation of concerns” [1].

8. CONCLUSIONS

Network protocol stacks for the domain of resource-constrained embedded systems have to fullfill a broad variety of functional requirements, while at the same being thrifty with respect to hardware resources, especially memory and energy. This calls for static configuration approaches to tailor the provided functionality to the actual application’s needs.

We presented the CiAO/IP stack and its underlying design approach for embedded system software, which pushes the limits of static configurability to a new level. In our design approach, the domain (mainly given by RFCs) is analyzed with *software product line* methodology; the resulting features are then structured by a *concern impact analysis* and implemented with *aspect-oriented programming* as fine-grained and loosly coupled implementation artefacts.

By following this design approach in the development of CiAO/IP, we did not only achieve a clear and complete separation of concerns in the code (thus, maintainability and portability), but also excellent configurability and scalability of the resulting protocol stack: CiAO/IP outperforms *uIP* and *lwIP* in terms of code size (up to 84% / 88% less than uIP / lwIP for a UDP sender on an AVR), throughput (up to 587% / 33% higher than uIP / lwIP for a TCP sender on IA-32 with Gbit Ethernet) and energy consumption (up to 63% lower than uIP on AVR for a TCP sender).

We hope that our results encourage other developers of system software to follow the guidelines in this paper.

9. ACKNOWLEDGEMENTS

We wish to thank the anonymous reviewers for their helpful and encouraging comments. Special thanks go to Prabal Dutta, whose shepherding helped us to clarify the content of this paper.

This work was partly supported by the German Research Council (DFG) under grant no. SP 968/4-1, SP 968/5-1, and SFB 876 projects A1 and A4.

10. REFERENCES

[1] Sven Apel and Christian Kästner. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[2] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[3] D. Barisic, M. Krogmann, G. Stromberg, and P. Schramm. Making embedded software development more efficient with SOA. In *21st International Conference on Advanced*

Information Networking and Applications Workshops, 2007 (AINAW '07), volume 1, pages 941–946, May 2007.

[4] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd, 2010.

[5] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *26th Int. Conf. on Software Engineering (ICSE '04)*, pages 702–703. IEEE, 2004.

[6] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989.

[7] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27:23–29, 1989.

[8] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98. ACM, 2003.

[9] Adam Dunkels. *The uIP Embedded TCP/IP Stack. The uIP 1.0 Reference Manual*. Swedish Institute of Computer Science, 2006.

[10] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki — a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[11] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 263–270, New York, NY, USA, 1999. ACM.

[12] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '03)*, pages 1–11, San Diego, CA, USA, 2003. ACM.

[13] Jonathan W. Hui and David E. Culler. IP is dead, long live IP for wireless sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 15–28, New York, NY, USA, 2008. ACM.

[14] Frank Hunleth and Ron Cytron. Footprint and feature management using aspect-oriented programming techniques. In *2002 Joint LCTES & SCOPES Conferences (LCTES/SCOPES '02)*, pages 38–45, Berlin, Germany, June 2002. ACM.

[15] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[16] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *11th Software Product Line Conf. (SPLC '07)*, pages 223–232. IEEE, 2007.

[17] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: Analysis and case studies. In Dirk Muthig and John D. McGregor, editors, *13th Software Product Line Conf. (SPLC '09)*, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.

[19] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th Int. Conf. on Software Engineering (ICSE '05)*, pages 49–58, New York, NY, USA, 2005. ACM.

[20] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. *TinyOS: An Operating System for Wireless Sensor Networks*. Ambient Intelligence. Springer, Heidelberg, Germany, 2005.

[21] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Aspect-aware operating-system development. In Shigeru Chiba, editor, *10th Int. Conf. on Aspect-Oriented Software Development (AOSD '11)*, pages 69–80, New York, NY, USA, 2011. ACM.

[22] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX ATC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[23] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In Yolande Berbers and Willy Zwaenepoel, editors, *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM.

[24] Anthony Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[25] Bosko Milekic. Network buffer allocation in the FreeBSD operating system. In *Proceedings of BSDCan*, Ottawa, ON, Canada, May 2004.

[26] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. AW, 2001.

[27] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 95–107, New York, NY, USA, 2004. ACM.

[28] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Approaching Non-Functional Properties of Software Product Lines: Learning from Products. In IEEE Computer Society Press, editor, *Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC 2010)*, pages 147–155, Los Alamitos, CA, USA, 2010.

[29] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *1992 USENIX ATC*, Berkeley, CA, USA, June 1992. USENIX.

[30] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[31] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *21st ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '06)*, pages 481–497, New York, NY, USA, 2006. ACM.

[32] Jochen Streicher, Christoph Borchert, and Olaf Spinczyk. Upcall dispatcher aspects: Combining modularity with efficiency in the CiAO IP stack. In *1st AOSD W'shop on*

Modularity in Systems Software (AOSD-MISS '11), pages 23–27. ACM, March 2011.

[33] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In Christoph M. Kirsch and Gernot Heiser, editors, *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2011 (EuroSys '11)*, pages 47–60, New York, NY, USA, April 2011. ACM.

[34] Jim Turley. The two percent solution. *embedded.com*, December 2002. `http://www.embedded.com/story/OEG20021217S0039`, visited 2011-04-08.

[35] Charles Zhang and Hans-Arno Jacobsen. Quantifying aspects in middleware platforms. In *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 130–139, New York, NY, USA, 2003. ACM.