THÈSE DE DOCTORAT

de

L'UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS

Spécialité

SYSTÈMES INFORMATIQUES

présentée par

Matthias Jung

pour obtenir le grade de DOCTEUR de l'UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS

Sujet de la thèse:

Software Engineering Techniques for Support of Communication Protocol Implementation

Soutenu le 3 Octobre 2000 devant le jury composé de:

Président	Dr. Walid Dabbous	Directeur de Recherche, INRIA
Rapporteurs	Dr. Wolfgang Effelsberg	Professeur, Université de Mannheim
	Dr. Béatrice Paillassa	Maître de Conférence, ENSEEIHT
Membre Invité	Dr. Vincent Roca	Chargé de Recherche, INRIA
Directeur de Thèse	Dr. Ernst W. Biersack	Professeur, Institut Eurécom

Acknowledgements

Thanks to my supervisor Ernst Biersack for his trust and his honesty, his encouragement, and the many things I've learned during the last three years.

Thanks to Stefan Böcking and the people from Siemens Research (ZT IK2) for their financial support, which gave me the opportunity to follow my PhD studies at Eurécom.

Thanks to the members of my jury – Walid Dabbous, Wolfgang Effelsberg, Béatrice Paillassa, and Vincent Roca – for the time they spent to read and comment my work.

Thanks to Didier Loisel and David Trémouillac for their permanent cooperation and competent help concerning hardware and software administration.

Thanks to Evelyne Biersack, Morsy Cheikhrouhou, Pierre Conti, and Yves Roudier for reading and correcting the french part of my dissertation.

Thanks to the btroup members and to all people at Eurécom who contributed to this unique atmosphere of amity and inspiration.

Thanks to Sergio for his open ear and for never running out of cookies.

Thanks to Audrey for her love, patience, and endless encouragement.

Thanks to my parents for making all this possible.

Contents

List of Figures

List of Tables

Acronyms

ALF: Application Level Framing
ATP: Application-Tailored Protocol
CASE: Computer-Aided Software Engineering
CBD: Component-Based Development
DOS: Distributed Object System
FSM: Finite State Machine
GPP: General Purpose Protocol
ILP: Integrated Layer Processing
IP: Internet Protocol
LLC: Logical Link Control
MAC: Media Access Control
OMG: Object Management Group
OO: Object-Oriented
OS: Operating System
PE: Protocol Editor
PA: Protocol Animator
PITOU: Protocol Implementation, Tailoring, and Organization Utilities
QoS: Quality of Service
ORB: Object Request Broker
PC: Personal Computer
RTT: Round-Trip Time
SE: Software Engineering
TCP: Transmission Control Protocol
UDP: User Datagram Protocol
UML: Unified Modeling Language
VAJ: Visual-Age for Java [70]
XTP: eXpress Transport Protocol

Chapter 1

Introduction

1.1 Motivation

The pervasiveness of the Internet fosters the development of new distributed applications with increasingly diverse and multi-fold communication requirements like multicast or real-time data streaming. While the applications' communications requirements diverge, the Internet protocol suite still offers only two standard transport protocols: TCP [109] and UDP [110]. TCP provides a connection oriented, fully reliable and ordered, point-to-point transmission service, whereas UDP offers only a simple datagram service. The choice between TCP and UDP can be seen as a choice between *all* or *nothing*, which may create a gap between what an application requires and what it gets from its transport protocol.

Hence, developers of distributed applications are often obliged to close this gap by implementing their own transport services. That means, the development of distributed applications often involves the implementation of rather sophisticated communication protocols that deal with both application and transport semantics. Since protocol implementation is known to be difficult and tedious, the cost in terms of time and money for implementing, deploying, and maintaining new distributed applications may become unacceptably high.

The goal of our thesis is the development of techniques and tools to minimize the cost for implementing protocols in end-systems and thus the cost of deploying new distributed applications. We consider modern software engineering techniques and examine how they can be applied to protocol implementation. Our work intents to bridge the two research domains of protocol implementation and software engineering. We exploit well-known techniques – object-oriented frameworks, design-patterns, and component-based development – to maximize code reuse and thus to leverage the domain knowledge of communication protocol experts.

1.2 Protocols

1.2.1 What is a Protocol?

Throughout the thesis, we will refer to a **protocol** as *a set of rules and formats that govern the interaction between communicating peers* [152]. A **protocol specification** refers to the documentation and description of a protocol. A **protocol implementation** is the coding of a protocol using a specific programming language.

Examples for typical protocol functions on transport level are error-control, flow-control, or congestion control. Examples for typical protocol functions on application level are data representation, encryption, and session management.

1.2.2 General Purpose vs. Application-Tailored Protocols

Characteristics

The protocols we are concerned with in our thesis are protocols built to accommodate the needs of a specific application in contrary to protocols that are assumed to serve a wide number of different applications. We therefore refer to the former family of protocols as **application-tailored protocols** (ATP) and the latter as **general purpose protocols** (GPP).

In contrary to what is called **application protocols** and represented by the Application Layer of the ISO/OSI model, application-*tailored* protocols integrate protocol functions across all end-to-end layers (ISO/OSI Layers 4-7) and even the Network Layer[1]. Application-tailored protocols deal with application semantics as well as with transport semantics and typically make only minimal assumptions about the underlying communications services. They can be considered part of the application that uses it. The RTP [128] implementation in the Free-Phone [56] tool is an example for an application-tailored protocol. FTP [16], however, is an application protocol, since it deals only with application semantics and relies on a reliable and ordered transport service rather than providing it.

Application-tailored protocols should not be confused with **configurable transport protocols**. For example, TCP provides some configuration facilities like disabling Nagle's data blocking algorithm or the use of options. XTP [35] allows even to change fundamental service characteristics to adapt to different applications. Nevertheless, XTP does not deal with application semantics and remains a – though flexible and configurable – general purpose protocol.

We briefly resume the main characteristics of application-tailored protocols. They

[1]The Active Networking paradigm [140] can be considered as an approach to apply application-tailoring for network layer protocols.

- deal with application semantics
- are implemented and executed as part of the application program
- integrate protocol functions from different layers
- make only minimal assumptions about the underlying network and transport services and rather implement these services themselves
- address typically the needs of a single application

Discussion

General purpose protocols (GPP) typically reside in the operating system kernel. This assures on the one hand better performance, but on the other hand makes deployment of new protocol implementations and later modifications of existing code difficult and tedious. Installing protocol software in the operating system kernel is therefore a serious obstacle to innovation.

GPP typically include a relatively high number of services and features to be general enough to meet the requirements of different applications. The fact to accommodate many applications also creates the permanent need of being back-ward compatible. Integrating a new feature in a GPP requires testing for many different application scenarios. The delay between specification and deployment of new features is even aggravated when the protocol is standardized.

On the other hand, step-wise and careful refinements of GPP implementations also lead to robust and optimized code. The life-cycle of general purpose protocols is very long as the example TCP shows, which has been around for more than two decades.

In many respects, application-tailored protocols (ATP) are the opposite to general purpose protocols. Since they are implemented within user space, implementation, de-bugging, and maintenance is less cumbersome and expensive. While user space implementations can hardly achieve the performance (in terms of execution speed) of kernel implementations, this handicap can be compensated by the fact that application-tailored protocol implementations are rather lean and better adapted to the application requirements. Research [7], [57], [94], [36] shows that well adapted communication services can outperform general purpose protocols in terms of performance experienced by the application.

Since ATP are built from scratch without the constraint of being backward-compatible, they can easily be deployed, modified and extended. As the example RealPlayer [3] shows, ATP software may even be down-loaded and deployed together with its application over the Internet.

From the perspective of an application developer, the overall implementation and deployment cost and time of a new distributed application is of great importance. If the gap between services required and provided is rather small, the application developer could rely on an existing general purpose transport protocol and accept minor mismatches with the desired services. That way, he avoids implementing transport protocol functions and only needs to implement the application protocol, which normally is a less complex task to do.

A prominent example of this approach is the deployment of HTTP [54] over TCP. Although the stream-oriented TCP service does not match very well the request/response character of Web-Traffic, and a number of problems has been experienced [98], [142], the use of TCP for web-servers has allowed for immediate deployment, fits well in the existing infrastructure, and has so become a great success.

The cost of building distributed applications that incorporate application-tailored protocols largely depends on the size and reusability of existing protocol libraries and modules. How to achieve reusability for protocol software is one major concern of our work. Our vision is to implement protocols by simply putting existing pieces together without the need to write any additional line of code.

Table 1.1 summarizes the characteristics of general purpose and application-tailored protocols.

	General Purpose	**Application-Tailored**
Protocol Implementation Cost	very high	medium
Time of Deployment	long	short
Life cycle	long	short
Element of Reuse	whole protocol	protocol modules
Service Match	depends	high
Execution performance	very high	low
Overall Performance	depends	high

Table 1.1: General Purpose vs. Application-Tailored Protocols

1.3 Software Engineering

Software Engineering is a disciplined approach to software development to manage the complexity of large software systems. Its high-level goals are to improve **productivity** during the process of software development (specification, design, implementation and maintenance) and to improve the **quality** of the code produced. Productivity is measured in time, cost, and personnel resource requirements. Most cited qualities of ”good” software are

- correctness: does the software meet its requirements and specifications?
- maintainability: can the code be understood and maintained?

- modifiability: is the software open for modifications and extensions?
- reusability: can parts of the software be reused for other applications?
- efficiency: does the software make efficient use of computer resources?
- robustness: does the software work even under unforeseen conditions?

There are of course a number of other criteria – like portability, ease-of-use, compatibility, integrity – and business-oriented qualities that may play an important role during software development.

1.3.1 Object-Oriented Software Engineering Concepts

In order to achieve the goals listed above, a collection of principles are applied during the whole process of the software development process. **Object-oriented** (OO) software engineering refers to a SE paradigm that gives explicit support to fundamental principles such as abstraction, information hiding, decomposition, or hierarchy. Object-oriented programming languages provide language features (like classes, encapsulation, inheritance, and polymorphism) to implement these principles.

Abstraction is a fundamental principle to deal with complexity. *The essence of abstraction is to extract essential properties while omitting inessential details* [117]. Objects and classes are much better abstractions to model real-world problems than e.g. functions. The principle of abstraction goes hand in hand with the principle of **information hiding**, that is *hiding all internal details of an entity that do not contribute to its essential characteristics* [25]. In the context of object-orientation, the term **encapsulation** is often used as a synonym to information hiding.

Decomposition refers to the process of *dividing a system into smaller parts*. Booch [25] states the following advantages of object-oriented decomposition compared to algorithmic decomposition: *OO decomposition leads to software systems that can be incrementally extended, are more resilient to change, and improve reuse of existing parts*.

Hierarchy refers to *ranking or ordering of abstractions*. Examples for hierarchical relations in the object model are *aggregation* ("has-a"-relations) and *generalization* ("is-a"-relations). OO languages provide the mechanism of **inheritance** to support generalization.

Object-Oriented Methodology

The traditional methodology in software engineering divides development into four phases: analysis, design, implementation, and maintenance.

The main objective during the **analysis phase** of the software development process is to obtain a consistent and powerful model of the problem domain (in our context communications protocols). Activities include a domain analysis, the investigation of domain specific problems, and the definition of requirements.

During the **design phase**, the focus shifts away from domain specific issues towards issues and problems related to the implementation environment. The goal of design is to provide a basis for implementation. Typical design activities are therefore the development of a software architecture, breaking a system into sub-systems (layers, partitions), the choice of computer and programming platform, and the solution to concrete problems like concurrency control, user-interfaces, or data storage.

During the **implementation phase**, the results of analysis and design are transformed into programming language specific code. The **maintenance phase** comprises all activities during installation, test, and use of the implemented software including code modifications due to user feedback.

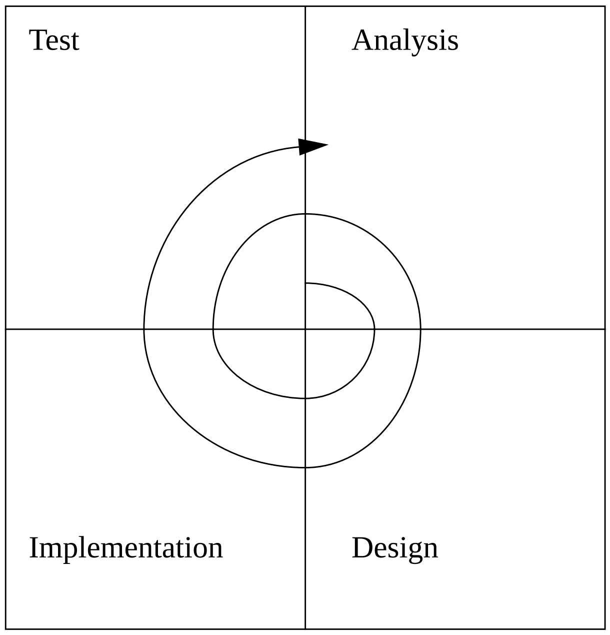


Figure 1.1: Spiral model (Boehm, 1986)

Normally, these phases experience several iterations. The spiral-model of Boehm [23] illustrates this need for permanent refinement and enhancement (Figure 1.1). However, in traditional software engineering, the phases described are clearly separated and supported by specific tools. These disruptions between phases may lead to information loss and misunderstandings. One indisputable benefit of an OO methodology is the possibility of using the same concepts, tools, and notations during the whole software development process. Analysis and design activities have no clear boundaries any more, but rather exist on a continuum (see Figure 1.2) [89]. This allows modeling, design, and implementation to be incrementally refined and done in parallel. Furthermore, maintenance is largely simplified since the model of the problem domain is still visible in the software.

Rumbaugh [118] denotes the activities during object-oriented analysis (OOA) as follows:

- **object modeling**, e.g. identification of objects, classes and their attributes and associations, development of a data dictionary, identification of association among objects, grouping of classes
- **dynamic modeling**, e.g. preparing scenarios and use-cases, build state diagrams, identify events
- **functional modeling**, e.g. building data flow diagrams, describe functions, specify optimization criteria

Typical activities during object-oriented design (OOD) are the extension and refinement of the analysis model by helper classes and associations, optimizations, the design of algorithms, or the packaging of classes and associations into modules.

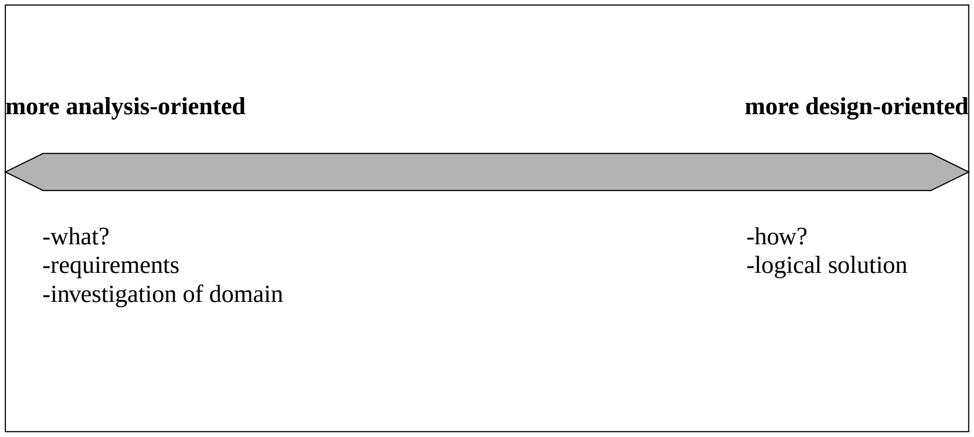


Figure 1.2: Analysis and design activities exist on a continuum (Larman, 1998)

1.3.2 UML – Unified Modeling Language

The Unified Modeling Language (UML) provides a standardized notation that fuses best-practice concepts (see Booch [25] and Rumbaugh [118]) to support object-oriented software engineering during the whole development cycle. UML is neither a programming language, nor a development tool. UML comprises a collection of diagrams that are used to specify, visualize and document the artifacts of software systems independently from the programming language. Each diagram captures a different aspect and provides a specific view of the system.

In the thesis, we make use of UML notation to describe and visualize our ideas for protocol modeling and protocol framework design. We therefore give a short overview of the most important UML diagrams used in the following chapters. For a more detailed description of UML, we recommend the book [26] or the web-site (www.rational.com) of the original developers of UML.

Use-case diagrams visualize the system's use cases, i.e. a description of a set of sequences of actions, including variants, that a system typically performs. A use-case analysis of end-to-end protocol systems can be found at the beginning of Chapter 2. However, we preferred to document possible use-cases in a textual style and do not apply use-case diagrams.

Class diagrams represent the static structure of a system by depicting the system's classes and class relationships. UML considers a class as a description of a set of objects that share the same attributes, operations, relationships, and semantics. Objects are the concrete manifestations of an abstraction and instances of a class. Classes can be abstract (represented in class diagrams by using *italic* style for the class name), i.e. not able to be directly instantiated.

The **visibility** of a class attribute or method indicates from which classes this attribute or method can be accessed. `Public` attributes or methods are accessible for any other class, `protected` only for sub-classes, and `private` only for the class itself. In Figure 1.3, you find the symbols for the different visibility levels.

The relationships between classes can be divided into the following categories.

- An **association** between classes represents a connection among their instances. An association can be parameterized with a **multiplicity**, i.e. a specification of the range of allowed cardinalities (see Figure 1.3 for examples), and a **role**, i.e. a description of the kind of behavior of the participating instances.

- An **aggregation** is a special form of association that specifies a whole-part relationship between the aggregate (the whole) and a part. Aggregation relationships are often refered to as ”has-a”-relationship.

- A **generalization** relationship between classes represents a ”kind-of” hierarchy between a generalized class and a specialized class. Instances of the generalized class may thus be substituted by instances of the specialized class. Generalization relationships are normally realized by the inheritance mechanism of OO languages.

- A **dependency** is a semantic relationship between two classes in which a change to the one may affect the semantics of the other (the dependent class). The creation of new instances is one example of a dependency relationship.

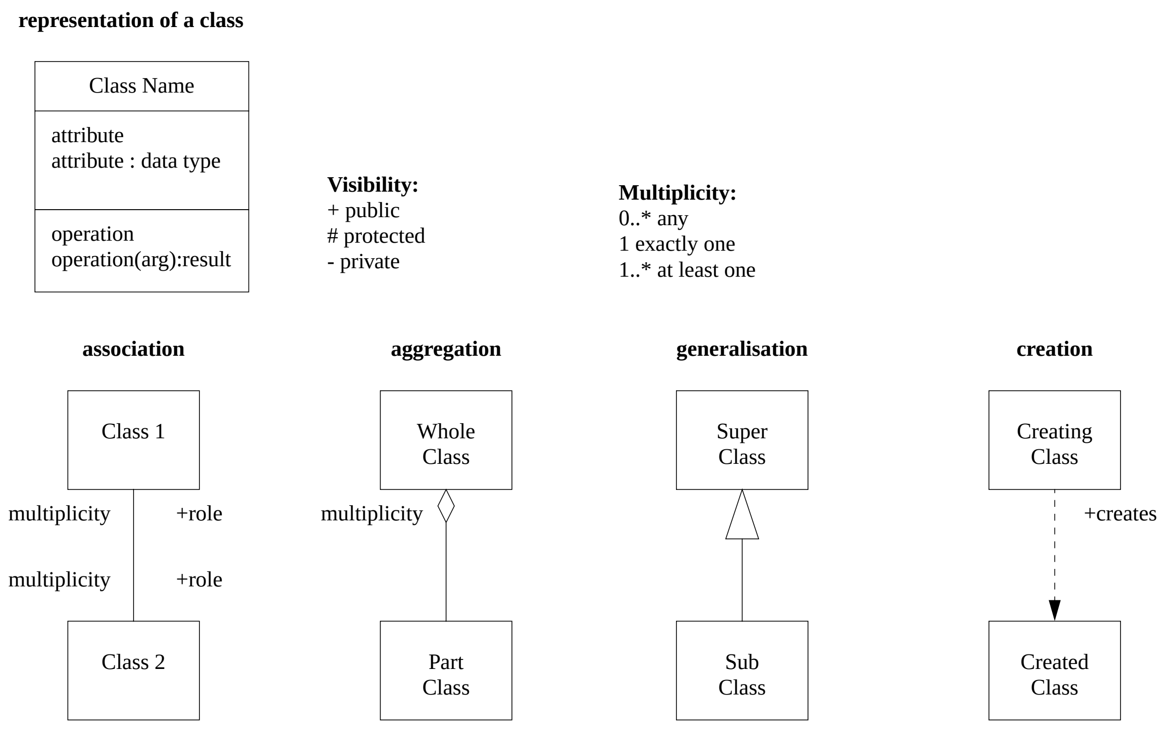
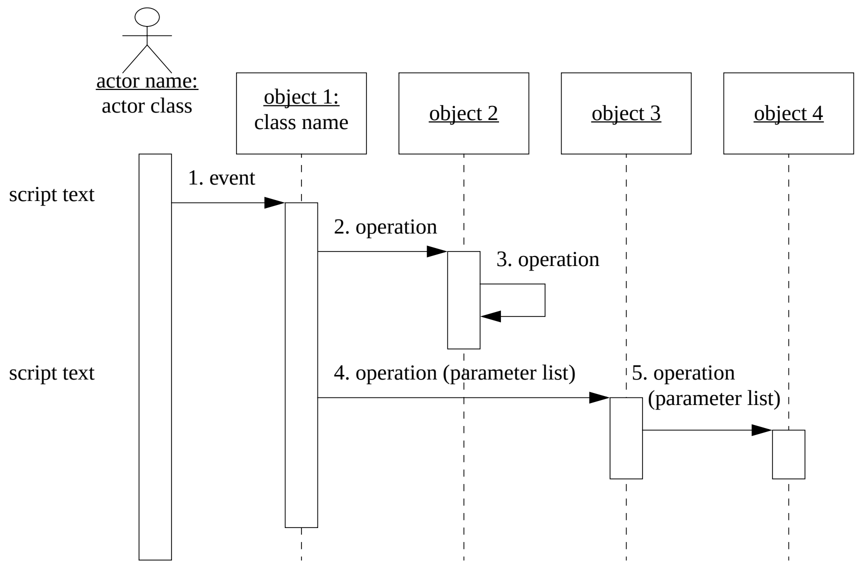


Figure 1.3: UML Class Diagrams

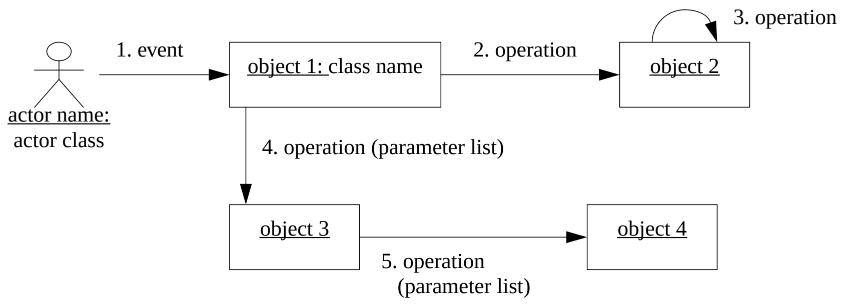
Besides the structural view of a system, it is important to describe the system's dynamic behavior, i.e. the interactions between instances of the system for a certain use-case. UML provides two kind of diagrams to describe the dynamics of a running system. While **sequence diagrams** rather emphasize the time ordering of interactions, **collaboration diagrams** emphasize the structural organization of the instances involved. Figure 1.4 depicts a sequence diagram and a collaboration diagram both representing the same use-case. In order to distinguish instances from classes, the names of instances are underlined. The actor of a use-case represents either the user himself or the role a user plays with respect to the system.

1.3.3 Object-Oriented Frameworks

We refer to an **object-oriented framework** as a *collection of interacting classes that are used to develop new applications with related requirements*. By encapsulating the changing part of an implementation behind well-defined and stable interfaces and decoupling it from the generic part of the implementation, frameworks allow software developers to concentrate on the specifics of the application instead of spending time to find solutions to recurring problems and design challenges. OO frameworks thus allow for rapid-prototyping and cost effec-



(a) Sequence Diagram



(b) Collaboration Diagram

Figure 1.4: UML Interaction Diagrams

tive implementation as well as enhanced modularity and easier maintainability of the software. Application developers just need to understand how to integrate application specifics in the framework at explicit plug-in points (so called **hot spots** [111]), and do not need to worry about architectural design issues.

What is the difference between a framework and a class library? First, frameworks are more domain specific than class libraries (typical class libraries implement lists, strings, or numerical functions). Second, frameworks provide classes that control the application, while class libraries are only passive (also known as the principle of **inversion of control** [126]). In practice, frameworks make heavily use of class libraries.

Schmidt [52] distinguishes **foundation frameworks** from **application frameworks**. While foundation frameworks provide rather loosely coupled, general classes that are useful for a large number of applications and thus rather close to class libraries, application frameworks impose stronger structural constraints for application developers and are meant to support only a specific class of applications. Foundation frameworks are successfully used for the development of multimedia applications (like JMF [1], MET++ [4]), graphical user interfaces (like MFC [2]), or middleware integration (like RMI [138] or Voyager [105]). Application frameworks have been developed for computer integrated manufacturing (CIM) [65], financial engineering [21], remote learning [67], and even e-business applications [69].

While the use of frameworks significantly simplifies the development of new applications, the design and implementation of a framework itself is a complex and challenging task and requires careful design. Johnson [77] compares framework design even with the design of a programming language (whereby the syntax of a framework is determined by the rules to implement applications), and considers frameworks as compilers for high-level, domain-specific languages.

Object-oriented frameworks can be classified into **black box** and **white box** frameworks. In black box frameworks, the application developer does not have insight in classes of the framework he uses. In white box frameworks, the application developer has access to internal information of framework classes and needs to understand the class hierarchies of the framework. Black box frameworks are extended by inserting classes with predefined interfaces at the provided hot spots. White box frameworks are extended by deriving new classes via inheritance and re-link them with the framework. Black box are easier to use for the application developer, but harder to design. They also offer higher flexibility by allowing runtime extensions. White box frameworks require less design efforts, but are harder to use.

1.3.4 Component-Based Development

D'Souza et al. [46] define component-based development (CBD) as *an approach to software development in which all artifacts – from executable code to interface specifications, architectures, and business models; and scaling from complete applications and systems down to small parts – can be built by assembling, adapting, and wiring together existing components into a variety of configurations*.

A component is usually characterized by the following attributes:

- Components are largely decoupled; they can be independently developed and delivered
- Components have explicit and well-specified interfaces for the services they provide
- Components have explicit and well-specified interfaces for services they expect from other components

- Components can be customized and composed with other Components without modification of code

CBD is often confused with object-oriented development. This stems from the fact that OO development can be seen as the technology best suited to CBD. In contrary to an object, a component provides a richer range of intercommunication mechanisms, a higher degree of reuse and adaptability, and usually a larger granularity. A component may comprise one or more objects.

Java-Beans [75],[50] is the component model provided by Sun. Java Beans is written in the Java programming language [136] and hence profits from Java's machine-independence and portability.

A **Java Bean** (or shortly bean) is the Java representation of a component, i.e. a Java class or object characterized by three elements:

- A **property** is a named attribute that may affect the behaviour or appearance of a bean (e.g. maximum packet size, time out value).
- An **event** stands for possibly asynchronous data that is generated by a component (e.g. window size changed, error appeared, etc.) It is fired by an *event source* and delivered to an *event listener*.
- The **behaviour** of a bean comprises all its methods accessible for any other component (public methods).

Any tool that is used to configure and wire the components together can identify properties, events, and behaviour of a bean by analyzing its Java code. A Java class that intends to be Java Beans compliant has to follow certain naming conventions, e.g. properties are identified by method names that start with `set` or `get`. For more details see the Java Beans specification [75].

Components can be configured and wired by either using mark-up languages (like XML [41]) or – the more popular approach – programming environments that allow visual specification, representation, and configuration of components. Such a tool is commonly referred to as **visual builder tool**. The visual builder tool we are using for visual implementation is *Visual-Age for Java* [70] by IBM.

1.3.5 Design Patterns

The idea of **patterns** is to capture solutions to recurring problems. The architect Christoph Alexander [5] first described patterns in the context of creating buildings, neighborhoods, and cities. He states:

Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice.

Gamma et al. [59] apply the same idea to the domain of (OO) software design[2]. **Design patterns** are a form of documenting recurring problems and solutions to software design. It is important to note that a solution to a software design problem is considered a pattern only when it is known to be successfully implemented in at least three different contexts (also called the **Rule-of-Three** [143]). Patterns are even considered to have an ”aggressive disregard of originality” (Brian Foote).

Patterns are documented by using a well-defined format. Each pattern gets a name, provides a description of the problem, the context of the problems, the forces that need to be balanced, and a description of the solution capturing static and dynamic behavior and interactions of the design elements [114].

Patterns provide guidance during software development and allow to reuse software solutions that a number of experts found useful. Patterns provide a common language among software developers and thus largely improve the communication among different groups of software engineers. The use of patterns can thus significantly improve the productivity during software development and enhance the quality of the software. However, patterns don't provide implementation and thus do not directly lead to code reuse. They neither disburden a programmer from understanding the problem first before he can reason about possible solutions.

Design patterns have been proved to be useful to document frameworks and to detect the hot-spots, i.e. the application specific plug-in points, during framework design [120]. Johnson [77] views design patterns as *the micro-architectural elements of frameworks*. Examples for the use of design patterns in the domain of communication software can be found in [121] and [9].

In addition to the rather low-level design patterns, Fowler [58] presents a number of **analysis patterns**, which are more oriented towards domain modeling than design patterns and apply during the analysis phase of the software development process. On an even higher level are the **architectural patterns** presented by Buschmann [31], which are meant to support architectural issues in building complex systems. Layering is an example for an architectural pattern.

1.3.6 Reuse

Reuse is an important concept during the whole software development process. The object-oriented paradigm allows to reuse domain-specific abstractions during the whole development process. Component-based development and frameworks promise the reuse even of large parts of the implemented software. Applying design patterns allows the reuse of design solutions and

[2]The expressiveness of OO abstraction gave rise to the idea pattern of software design patterns.

expertise.

The main goal of reusing *code* is saving cost in terms of time and money. Code reuse fosters rapid prototyping and deployment of new applications. A side-effect of code reuse is that the robustness and efficiency of the software can be enhanced due to frequent testing and deployment in different contexts.

A number of qualities increase the reusability of software. First of all, software must be conceived to provide a useful service that meets well-defined requirements. Ease-of-use is also important for reuse and comprises qualities like understandability, careful interface design, and good documentation.

During our thesis, we will focus on two other key requirements to make software reusable: software pieces must be flexible and decoupled. We refer to **flexibility** as *degree and facility a piece of software can be adapted to different requirements and application contexts*. Software may be highly efficient, robust, and easy to use – all these qualities are worthless when the software can not be easily adapted to the application context to be used in. Parameterization, configuration, and tailoring are mechanisms to make software more flexible. Parameterization of functions is the simplest form of introducing flexibility. Templates allow to parameterize classes and objects. Component-based development improves code reusability by providing configuration of software components. Frameworks foster the reuse of application architectures and infrastructures by allowing to tailor them.

The **granularity** of a piece of software is a very important parameter that influences the flexibility and mainly determines the gain of reuse. The finer the granularity of a piece of software, the higher its flexibility, but the smaller the gain of reuse. The coarser the granularity, the higher the gain of reuse, but the less the flexibility.

Coupling refers to *the degree of interdependence between software pieces*. Object-oriented libraries never delivered the promised level of code reuse [84], because objects are rather fine-grained abstractions that need to cooperate with many other objects to provide an expected service. These interdependencies lead to code that is difficult to understand and use. Hiding interdependencies behind stable interfaces or making them explicit in the definition of the interface are the most common mechanisms in good SE practice. Function-oriented programming reduces coupling by hiding internal data needed to perform a function. Object-oriented programming reduces coupling by bundling related functions and hiding information that would else-wise be spread over different parts of an application. Component-based development hides the interdependencies of classes, which would otherwise be difficult to understand and use, behind a well-defined interface. Frameworks hide the explicit interdependencies of software components. In all these concepts, we discover the appliance of the fundamental principles of abstraction and information hiding.

1.4 Dissertation Outline

This dissertation is organized as follows. Chapter 2 revises relevant work in the area of protocol structuring and modeling, demonstrates the problems of using fine-grained layering, and makes propositions on how protocol software should be structured to promote reusability and flexibility. Chapter 3 provides a detailed description of our Java protocol framework that implements the concepts from Chapter 2 to allow for rapid-prototyping, testing, and tailoring of protocol software. Chapter 4 describes how our structuring approach contributes to the implementation of generic tools for simulating, editing, and animating protocol software. Chapter 5 contains a number of case-studies to illustrate and discuss our concepts. Chapter 6 offers an evaluation of our work and the software engineering techniques that have been used, and summarizes the contributions of the dissertation.

Chapter 2

Modeling and Structuring Protocol Software

In this chapter, we first revise the state of the art for abstractions to structure protocol software. We show why the traditional layering approach is badly adapted to application-tailoring. We then present and discuss a set of principles that should be followed to make protocol software tailorable, extensible, and parts of it reusable across different protocols. These principles are presented in a style that is close to current formats used to describe software patterns: we first describe the main problem to be addressed and discuss the issues to be taken into account. We then come up with the description of an *idealized* solution (by leaving out certain design issues and low level details), its essential components, and a discussion of advantages and drawbacks. We conclude with a review of work in related or different domains that apply a similar solution. The idea behind this ”pattern-like” form of presentation is to make our ideas and concepts reusable independently of a concrete implementation or architecture and provide a more comprehensive form of documentation.

2.1 Protocol Modeling

2.1.1 Tasks of Protocol Software

In order to determine the tasks of application tailored protocol software, we make a use-case analysis, i.e. we examine prototypical behavior of protocol software based on simple examples.

The following use-case is typical for protocol processing in **output direction** (i.e. from the application to the network):

1. An application passes data to a protocol instance (e.g. a string)

2. The data is transformed from the application representation into a message in form of an array of bytes.
3. The message is manipulated (e.g. encoded).
4. Internal state information is updated (e.g. message is buffered)
5. Control information is added to the message (e.g. a sequence number)
6. The message is sent (e.g. written to a UDP socket)

The following use-case is typical for protocol processing in **input direction** (i.e. from the network to the application):

1. A message arrives at the peer host
2. The message is de-multiplexed to the right protocol instance
3. The message is parsed to determine control and payload information
4. The control information is used to update internal state (e.g. the next expected sequence number)
5. The payload is manipulated (e.g. decoded)
6. A new message may be created (e.g. to acknowledge reception of the last message)
7. The payload is transformed from a byte array into application representation and delivered to the application

The use-case above seem to be simple, but capture the main tasks of end-to-end protocol implementations, such as:

- Provision of interfaces for the application
- Processing of messages in output direction – message manipulation (adding control information to the header or transforming parts of the message) and updating state
- Provision of interfaces to the network – sending and receiving raw data
- De-multiplexing of incoming messages
- Processing of messages in input direction – parsing, removing header information, updating state
- Creation of new messages

2.1.2 Modeling Protocol Software – State of the Art

Most concepts and mechanisms used in end-to-end communication are well-explored and have been applied many times. Most end-to-end protocol software performs roughly the same kind of tasks. However, there are almost no solutions that allow to encapsulate the reused concepts of communication protocols in reusable software modules.

We think that there are several explanations. First, for a long time there has been no need for modular and reusable protocol code. The number of applications was small – ftp, e-mail, telnet – and their requirements could all be met by the same general purpose transport protocol, that is TCP. The implementation of application level protocols has been supported by middleware (like CORBA [130]). Second, modularity had this reputation of causing overhead – doing research in an area with a strong focus on efficiency and performance is not very motivating, when a key concept is considered inherently inefficient. The third reason lies in the complexity of protocol software and its difficulty to capture its different aspects – finite state machines, data and control flow, distributed algorithms – in appropriate abstractions. Additionally, not only correctness of the *code* is important, but also correctness of the messages exchanged with the communication peers. The constraints of a fixed message format significantly limit the possibility of extensions and modifications of protocol software.

We now examine well known abstractions for protocols and protocol software.

Layers

The oldest and still most popular abstraction of a protocol is the **layer** abstraction. Each protocol layer comprises a set of related protocol functions. Protocol layers are hierarchically ordered: each layer is built upon its lower layer (besides the lowest). A hierarchy of protocol layers is called a **protocol stack**. Each layer provides certain **services** to its higher layer (besides the highest), which are accessible via well-defined primitives. Each layer defines its own message formats. One characteristic of layered architectures is that layers communicate exclusively with their neighbour layers. The standard interface between protocol layers comprise a **down-call** function – called by the higher layer to pass a message to the lower layer in output direction – and a **up-call** function – called by the lower layer to pass a message to the higher layer in input direction. In output direction, each layer adds specific information to the message header. In input direction, this information is stripped by the corresponding layer of the peer entity and used for state update.

Layering protocol software provides a couple of benefits. It allows to cope with the complexity of a protocol system by decomposing it into more manageable pieces. It decouples the layer interface from its implementation and thus allows for local changes without affecting code of other parts of the system. Well-defined service interfaces of layers also facilitate standardization, compatibility, and reuse of protocol code.

The idea of layering complex systems into smaller, manageable pieces is older than the author of this thesis [45]. For almost two decades, layering has been accepted as *the* modeling and design principle for communication systems. In the Internet, work about packet switching [33] and the transition from the network control protocol (NCP) to TCP and IP [108] mark the milestones of the success of layering. In parallel, the ISO standardized the 7-layer model [152], [71].

However, there are a number of problems associated with layering of protocol systems. When layers are independently designed and implemented, processing and header overhead due to inter-layer communication [42] and replication of protocol functions [139] may be the consequence. Crowcroft [43] observed unexpected side-effects and performance variations for the use of an RPC layer on top of TCP. From the perspective of building application-tailored protocols, the lack of flexibility [144] is the most important liability of layered protocol software. The lack of flexibility is on the one hand due to the strict hierarchy of layering [83] – layers can access services only from neighbour layers –, and on the other hand due to the typically coarse granularity of layers[31].

Micro-Protocols

To introduce higher flexibility into protocol implementations, O'Malley and Peterson [106] replaced the layer by a fine-grained abstraction called **micro-protocol**.

O'Malley and Peterson consequently apply decomposition of a given protocol stack to factor out atomic protocol functions and encapsulate each of them in micro-protocols. Micro-protocols are small modules that perform an atomic protocol function. So called **virtual-protocols** are used to connect micro-protocols and decide which way a message goes. The resulting organization is supposed to allow for easier modifications compared to the original layered organization. Decomposition also allows to detect and eliminate unneeded or redundant protocol functions. In traditional layered architectures, the protocol graph is simple, the functionality of a layer is complex, and the whole topology static. On the other hand, a micro-protocol architecture promotes complex protocol graphs, a single function per layer, and flexible composition to adapt to application requirements.

While the work on micro-protocols still leaves many questions open – especially concerning the reusability of micro-protocols, their degree of decoupling, their flexibility, their interface design – it demonstrated the benefits of a fine-grained protocol organization and showed that highly modular protocol software does not necessarily suffer from performance penalties. To the question ”What is the right number of layers?”, the micro-protocol architecture gives a simple answer: ”It depends on the message”.

State Machine

A common approach to model protocol layers is to consider them as finite state machines (FSM). The big advantage of making use of such a well-understood and powerful tool as FSM is that both protocol specification and implementation process are supported. FSM provides a formal model and mathematical notation that allows to express protocols as a set of states and their transitions, and allows to verify their correctness. The use of FSM has been particularly useful to model TCP, which comprises 13 different states.

However, modeling protocol layers as finite state machines often covers only one aspect of a protocol: the flow of control information. Data flow processing is not captured by FSM and must be implemented either ”by hand” or using other abstractions or tools, which requires the protocol developers to be familiar with yet another language and yet another tool. Hence, protocol implementation using FSM based languages normally results in semi-automatic compiling, manual integration and optimization.

2.2 Why not just applying finer granularity to layering?

Since layering has proved to be a powerful abstraction, and fine-granularity has proved to increase flexibility, why not just simply combining these two concepts for the purpose of implementing application tailored protocols? Based on some simple examples, we will illustrate in the following why fine-grained layering is not very well suited to implement application-tailored protocols.

2.2.1 Coupling of Streams

One common protocol function in layered protocol systems is **logical de-multiplexing**. Feldmeier [53] defines logical multiplexing as *mapping of multiple streams of layer n into a single stream that is passed to layer $n-1$* (in contrary to *physical resource de-multiplexing* used to share physical resources such as communication links). Logical multiplexing has the disadvantage that lower layers can not distinguish different higher layer streams (see Figure 2.1), which makes it very difficult to apply different quality of service (QoS) to different streams without violating the layer principle. This problem applies to streams from different applications as well as to different streams of the same application.

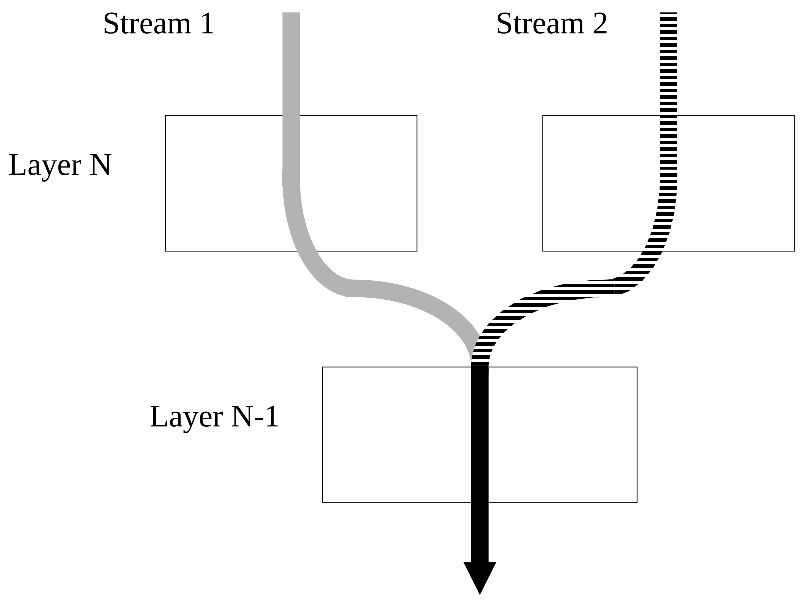


Figure 2.1: Coupling of Streams

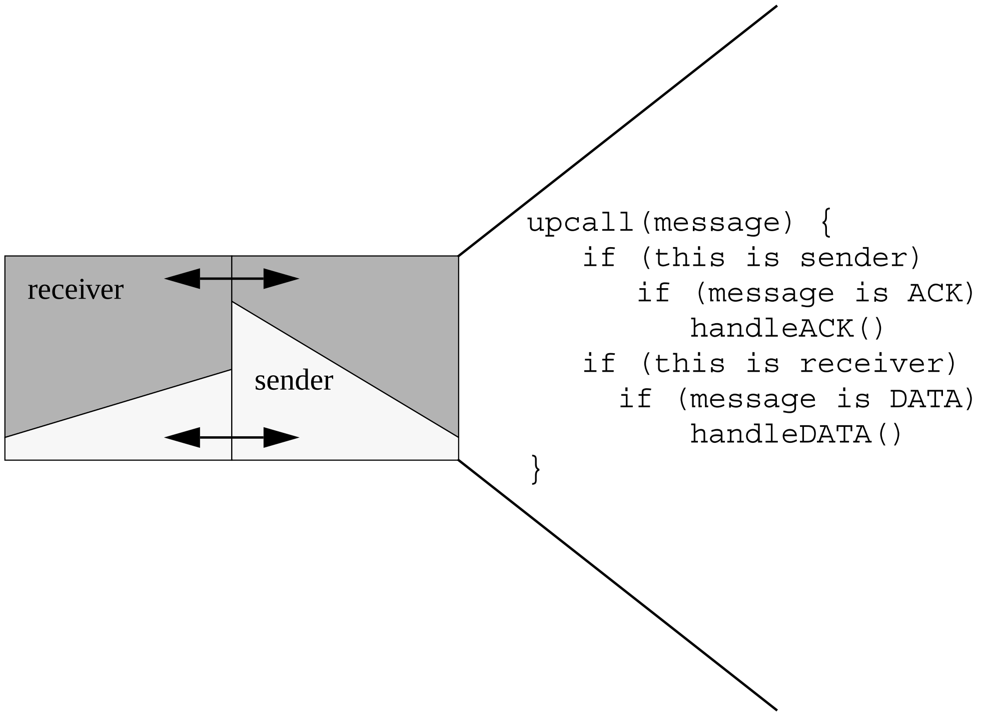
2.2.2 Coupling of Sender- and Receiver Functions

To illustrate another problem of layering, we consider the following stop-and-wait protocol: the sender gives each data chunk a sequence number, copies, buffers, and sends it. The sender also maintains a timer, which is set each time a chunk is sent (the timer-interval is pre-defined and fixed). The receiver acknowledges each incoming packet based on its sequence-number. When the sender receives an acknowledgement for the currently buffered chunk, it frees its buffer, stops the timer, and proceeds with the next data chunk. When the timer expires before an acknowledgment arrives, a loss is assumed and the currently buffered data chunk will be resent.

This simple protocol would typically be implemented within a single layer. When the layer is used on the sender side, application data is handled by down-calls. If no data chunk is outstanding (unacknowledged), the data chunk is sequenced, buffered, and given to the lower neighbour layer (in this case sent). If the last data chunk sent is not yet acknowledged, the chunk is written to a queue. The up-call method implements either handling of acknowledgements (sender functionality) or handling of data chunks (receiver functionality). During an up-call, the layer must be aware if it acts in the role of sender or receiver, and then check if the corresponding header fields of the message are correct. It then either implements acknowledgement handling (remove outstanding data chunk from the buffer and process the next data chunk in the waiting queue) or user data handling (strip header information and give the message to the higher layer).

We see here another problem of fine-grained layering: a layer is by default symmetrical, i.e. it implements both sender and receiver functionality. In our example, the sender's flow/error-control algorithm (stop-and-wait) is coupled with the receivers acknowledgement strategy (positive ack for each data chunk). However, the acknowledgement strategy is fairly independent from the stop-and-wait protocol and could be used together with more sophisticated flow- or error-control protocols, too.

The layer implemented is also hardly reusable for uni-directional communication. Modifications would be necessary to suppress functionality at each side. Additionally, many simple sender functions in protocols (such as logging or gaining statistics) do not even have a peer function at the receiver. The down-call interface is not implemented in these cases. Hence, for fine-grained structuring, layering is a too inflexible abstraction (see also Figure 2.2).



```
upcall(message) {
   if (this is sender)
      if (message is ACK)
         handleACK()
   if (this is receiver)
     if (message is DATA)
         handleDATA()
}
```

Figure 2.2: Layering leads to mixing receiver and sender functions

2.2.3 Coupling of Distinct Protocol Functions

A problem of the example protocol above is that its timer-interval is static, i.e. it can not be adapted dynamically to the actual RTT. This may result in too many timeouts and retransmissions when the timer interval is much shorter than the actual RTT, or in late detection of packet loss when the timer interval is much longer than the actual RTT. In order to determine a good timer-interval, the sender must estimate the round-trip-time to the receiver by measuring and averaging the times between the emission of the data chunks and the reception of the corresponding acknowledgements. How would we extend our protocol above to allow for dynamic timer-intervals? Should we implement a new layer or modify and extend the old one?

Implementing the whole protocol in a single layer definitely trades off a lot of flexibility. Error-control and RTT-estimation would be tightly coupled and can only be reused together. However, RTT-estimation is useful in other contexts, too, e.g. with window-based flow control schemes.

Implementing stop-and-wait and RTT-estimation in two distinct layers, however, raises other problems. It would be extremely inefficient, if both layers would implement their algorithms completely independently from the other, e.g. when the RTT-estimation layer would make measurements based on own data requests sent from time to time, which are answered by the RTT-estimation peer-layer. Strict boundaries would thus cause serious processing overhead and waste of network resources. The only way to deal with this overhead, is violating the layer principle and allow layers to share header data and exchange control messages. However, sharing information largely compromises the big advantage of layering: its simplicity due to uniform interfaces and minimized interaction.

2.2.4 Coupling of Header Information and Protocol Code

The example above points to another problem. Imagine that the sequence-number space in the message header is not sufficient and we need to reserve now 4 instead of 2 bytes in the message header. Changing the header a little bit may seem to be a marginal aspect. However, both protocol layers are concerned by this change and require modifications possibly at different places in the protocol code. This renders tailoring of protocols fairly error-prone.

Additionally, the question arises, which of the two layers that share header data is responsible to add and strip the respective information from the message to make the message readable for the adjacent layers. Layers operating on the same header fields are implicitly tangled and can hardly be reused without modifications. Figure 2.3 illustrates this problem. The two higher layers work as expected, the lowest layer however uses data already appended by its higher layer. It therefore does not append header data, but assumes that the higher layer already has done this. In input direction, the lower layer parses the message, but does not strip its data (will be done by the middle layer). Hence, the lowest layer can not be reused without modifications in other layered systems (or only together with the middle-layer).

2.2.5 Coupling of Protocol Processing and Thread-Model

The thread-model used in protocol software is an important design issue, which concerns both performance aspects and ease of implementation. In the **thread-per-layer** model, each layer implements its own thread, which pushes processed data to the queue of a neighbour layer. The

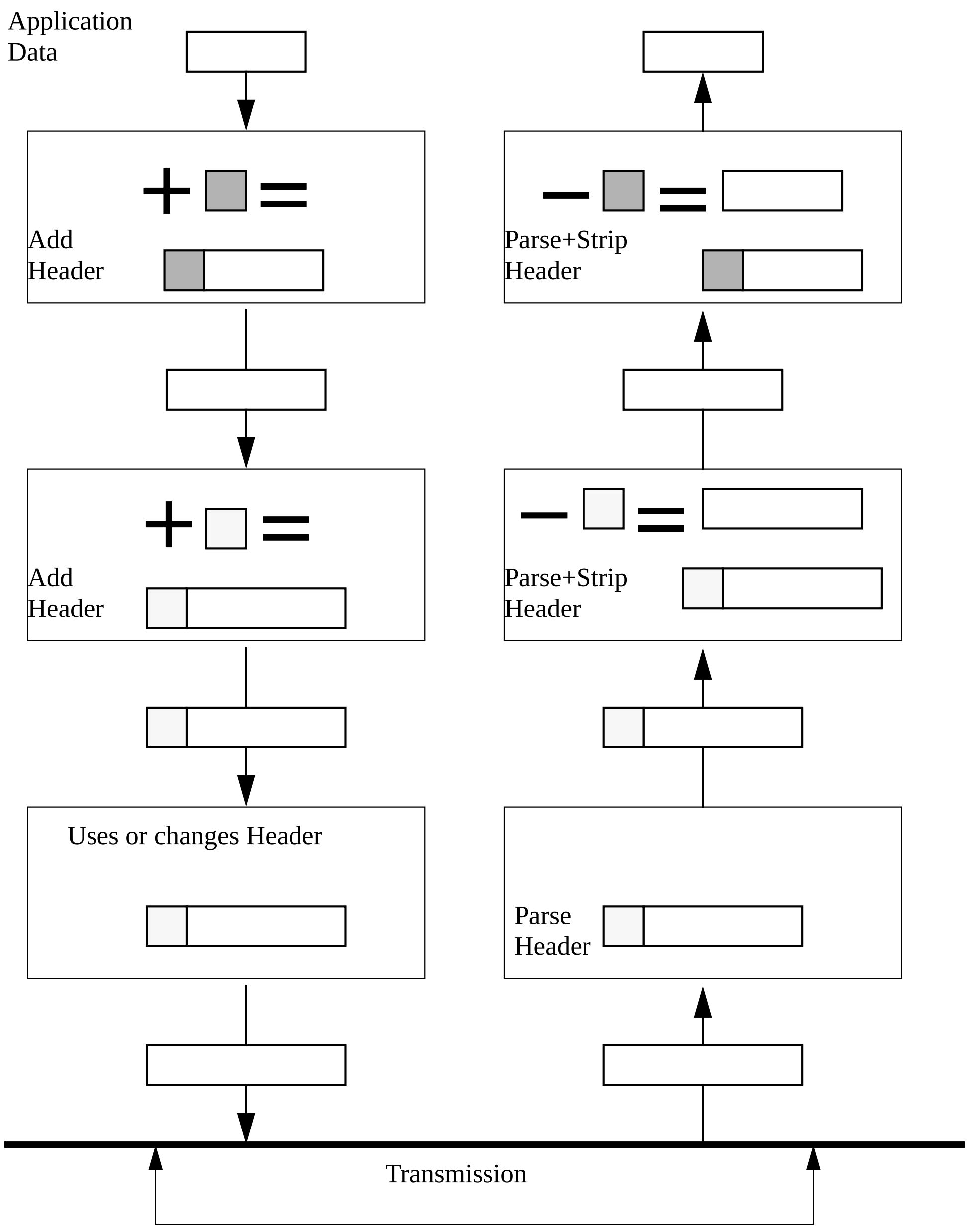
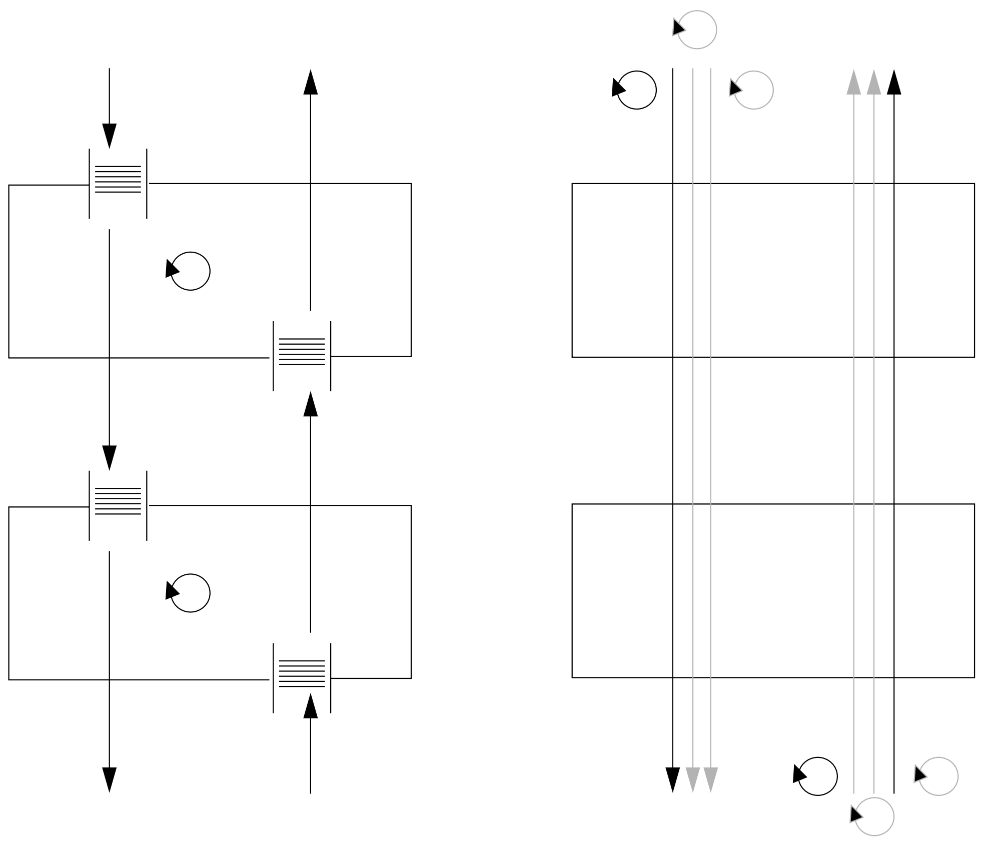


Figure 2.3: Sharing Header Fields couples Layers

advantage of this model is its simplicity, but it can become rather inefficient when a protocol stack is built out of many fine-grained layers. A lot of time is spent for communication between threads by writing to and reading from the message queues.

A more promising approach is the **thread-per-message** model, where one thread guides a message through the whole protocol stack in the same context. While thread-per-message is an appealing approach, it also holds some pitfalls. Debugging is more difficult since it is not obvious when a new thread is started or an existing one is stopped. Additionally, the programmer has to take care that messages are not re-ordered due to concurrent threads. The two models are depicted in Figure 2.4.



(a) Thread-per-Layer (b) Thread-per-Message

Figure 2.4: Different Thread-Models

Since layers may throw away messages or create new messages, they need to have access to an programming interface to control the threads that transport messages. (i.e. basically to stop a thread or create new thread). Unfortunately, giving each layer access to the thread-pool tangles the protocol code with code for thread handling and thus creates dependencies with the sub-system.

In both cases, thread-per-layer and thread-per-message, the thread-model is tightly coupled with protocol processing. In other words: implementing a layer depends on the thread-model used. A posterior change of the thread-model implies error prone code modifications of all layers.

2.3 Modeling Protocols: A System of Structuring Principles

2.3.1 Introduction

This section identifies a set of structuring principles for protocol software with the objective to support application-tailoring of protocol implementations. We describe how these structuring principles enhance decoupling, flexibility, and reuse of different protocol functions. Figure 2.5 depicts the relationships between the structuring principles we describe. The figure also includes the use of *Layers* to illustrate how layering is related to our principles. Our approach should be considered complementary to *coarse-grained* layering, i.e. our structuring principles may be useful to structure layers itself. They are nevertheless intended to replace naive *fine-grained* layering.

We briefly discuss the problems and forces our propositions need to deal with.

- Different applications have diversified communication service requirements and thus need different protocols, on the other hand, they must share system and network resources. The first problem to solve is therefore how applications can be decoupled from each other while efficiently making use of the common resources. This aspect is covered by the principle of *Outsourced De-Multiplexing*.

- The applications' requirements evolve over time. Protocol software should therefore allow to add or remove services. It is furthermore important that each service meets individual QoS requirements. These aspects are covered by the principle of *Data Path Reification*.

- Reuse, configurability, and granularity of protocol components are important issues to allow for rapid development and composition of new protocols. How fine-grained structuring is applied to protocol software to assure easy modification, extension, and autonomy of all protocol components is covered by the principle of *Data Path Partitioning*.

- From the perspective of an distributed application developer, the programming interface to communication protocol services is of great importance. The protocol software should make (almost) no assumptions about the application and the underlying network to be not affected by any changes of those. These aspects are covered by the principle of *Data Path Classification*.

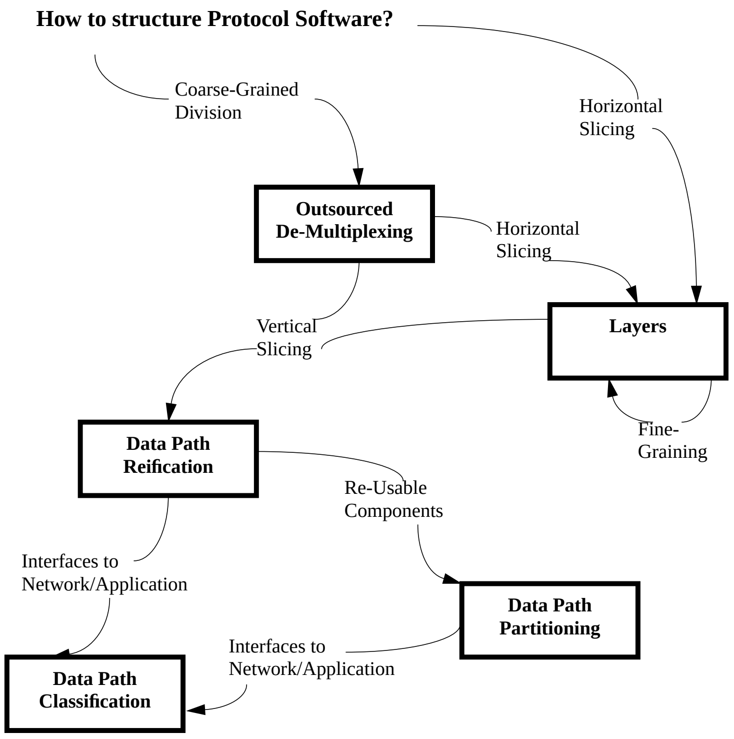


Figure 2.5: A System of Structuring Principles

2.3.2 Principle 1: Outsourced De-Multiplexing

Problem

De-multiplexing is a necessary protocol function when different applications share network resources. Network resources comprise *physical* resources like communication links or processors, and *logical* resources like protocol instances and their state (such buffers, timeouts, windows). In layered systems, logical multiplexing refers to ”the mapping of multiple streams of layer n into a single stream to be passed to layer $n-1$”[53]. While (de)-multiplexing is useful to share logical and physical resources, it also makes it difficult to distinguish different streams with different service requirements and thus causes ”crosstalk” between different streams. In order to assure the different service requirements of different applications – which may result in a variety of different application-tailored protocols – one need to introduce points in the protocol system to perform scheduling decisions and quality of service control. Unfortunately, introducing control and scheduling points significantly complicates protocol implementation and even violates the principle of information hiding, since it requires to spread information over the whole protocol code.

Solution

The solution to the problem of how to share resources and assign QoS to streams without coupling parts of the protocol code is to *integrate a protocol session into a single entity and delegate all de-multiplexing to a lower layer*. In order to emphasize that de-multiplexing is performed outside the actual protocol session and decoupled from message processing, we call the concept **Outsourced De-Multiplexing**. Figure 2.6 illustrates two organizations of protocol software. On the left, you see an implementation with de-multiplexing points within the protocol graph, one the right you see an organization that applies *Outsourced De-Multiplexing*.

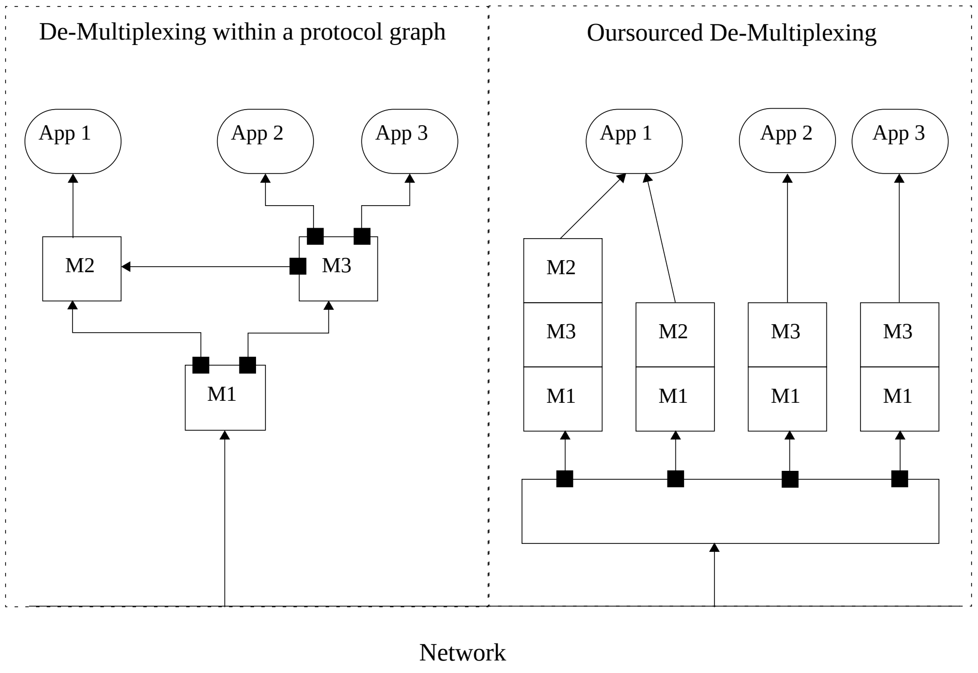

De-Multiplexing within a protocol graph
Oursourced De-Multiplexing
App 1
App 2
App 3
M2
M3
M1
App 1
App 2
App 3
M2
M3
M1
M2
M1
M3
M1
M3
M1
Network

Figure 2.6: Outsourced De-Multiplexing vs. Traditional Organization

The Figures 2.7 and 2.8 depict the prototypical entities of *Outsourced De-Multiplexing* and their interactions in UML notation. As **environment** we refer to a protocol session that provides a number of services and integrates a number of protocol functions that may encompass several layers. Environments may be structured as layers, state-machines, or whatever. The most important characteristic of an environment is that it provides de-multiplexing information in a well-defined, generic format without implementing de-multiplexing itself. Instead, an **anchor** entity is responsible for de-multiplexing. An anchor can be considered a packet filter or a separate protocol layer, the only task of which is de-multiplexing. Anchors are placed directly on top of the lowest de-multiplexing point (e.g. possibly even the device drivers) from the perspective of the application.

There are basically two ways to apply *Outsourced De-Multiplexing*. First, the anchor-layer

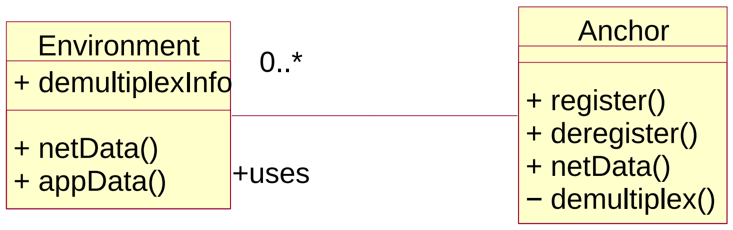


Figure 2.7: Outsourced De-Multiplexing: Class Diagram

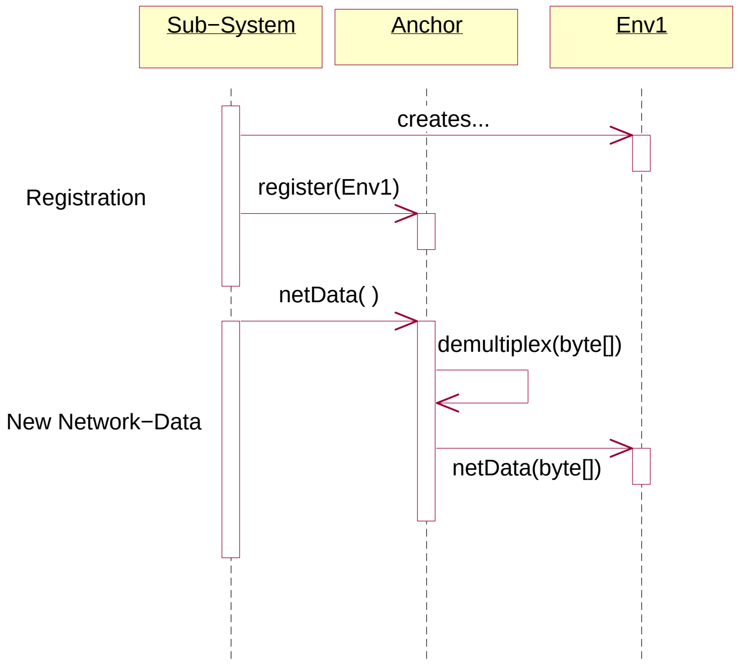


Figure 2.8: Outsourced De-Multiplexing: Sequence Diagram

can be run in user space and maintains references of all active environments. In that case, each active environment must be registered with its de-multiplex information. The anchor-layer obtains data from a transport service access point (like a UDP socket), looks in each packet and decides to which environment entity the packet is forwarded. This variant is necessary when the underlying transport protocol does not perform de-multiplexing or when several environments would share one *transport* service access point.

The second variant is based upon kernel de-multiplexing (e.g. TCP, UDP, IP). In this case, the complete kernel protocol stack acts as anchor that de-multiplexes incoming data to the specified socket based on the port number. Each environment is associated with its own transport socket, from where it gets its data directly. This variant is not only simpler since it avoids implementing the anchor layer, but also promises better latency since kernel processing is prioritized and highly optimized.

Scenarios

We present the typical scenarios of *Outsourced De-Multiplexing* when an anchor is used as explicit entity and one does not rely on kernel de-multiplexing (see Figure 2.8).

The first scenario describes the registration of new environments:

1. an environment is created
2. the application registers the environment at the anchor via `Anchor.register()`
3. the anchor stores the de-multiplex information together with a reference to the environment

The second scenario describes the arrival of data from the network:

1. the anchor is notified about the arrival of a packet via `Anchor.netData()`
2. the anchor applies the de-multiplex information of all registered environments to the packet within the internal function `Anchor.demultiplex()`
3. when the packet matches with the de-multiplex information of a certain environment, the anchor notifies this environment about new network data via `Environment.netData()`

Discussion

Outsourced De-Multiplexing allows to minimize sharing logical resources between applications. It trades off the possibility of sharing and thus reducing the use of logical resources (i.e. session state, buffers, queues) against the simplicity of protocol implementation and the reduction of cross-talk between flows of different applications.

We consider *Outsourced De-Multiplexing* useful especially for the implementation of application-tailored protocols, since application-tailored protocol software is executed as part of independent applications with individual service requirements. Since de-multiplexing of network data is already provided by UDP, all de-multiplexing can be delegated into the kernel, where it is done efficiently. That way, *Outsourced De-Multiplexing* promotes reuse of existing kernel de-multiplexing facilities.

Known Uses

From the perspective of a software engineer, *Outsourced De-Multiplexing* lies the foundation to simplify the implementation of protocol software and is a pre-requisite for introducing a vertical structure. *Outsourced De-Multiplexing* has already been successfully applied in various systems, and thus shown its usefulness. It therefore deserves to be called a *pattern*. Roca [116] modified a TCP/IP stack within the STREAMS [145] protocol environment in the BSD Unix Kernel to concentrate de-multiplexing just on top of the network drivers. He demonstrated a performance speed-up, more flexibility and simplicity compared with performing de-multiplexing in each layer. Rütsche [18] ported a kernel TCP/IP implementation into user-space that delegates de-multiplexing to the underlying ATM adapter to allow better control over protocol processing to guarantee the QoS of networked data. Braun [30] implemented a user space version of TCP, which lets de-multiplexing reside in the kernel to increase flexibility without compromising performance.

2.3.3 Principle 2: Data Path Reification

Problem

While *Outsourced De-Multiplexing* reduces cross-talk between flows of different protocol sessions and allows for application-specific quality of service, it does not tackle the problem how the protocol itself should be structured to be flexible, extensible, and configurable enough to be tailored to the applications needs.

Solutions have to take into account that applications may expect different services (e.g. reliable, real-time, secure services) at the same time from a protocol. Since the applications' requirements evolve over time, it must be easy to add new services or remove old ones without affecting the existing ones. Additionally, the protocol code should be as independent as possible from the runtime thread-model applied, that is changing the thread-model or optimizing the implementation of the chosen model must not affect the protocol code to avoid compromising its reusability.

Solution

We propose to cut a protocol into vertical slices, each of which reifies a possible *data path* through a protocol stack, i.e. all operations and data manipulations necessary to process a certain piece of data. Each data path is pre-defined in dependence of events like message arrivals, application access, or time-outs. Instead of considering data paths as something dynamic, which are implicitly constructed at runtime, we use data paths as a structuring element and explicit abstraction that can already be used during the specification of a protocol. We call this principle **Data Path Reification**. Figure 2.9 visualizes the difference between an implicit and an explicit data path.

Figures 2.10 and 2.11 depict the prototypical entities and dynamics of *Data Path Reification*. The entity **order-type** represents one data path through a protocol and produces runtime repre-

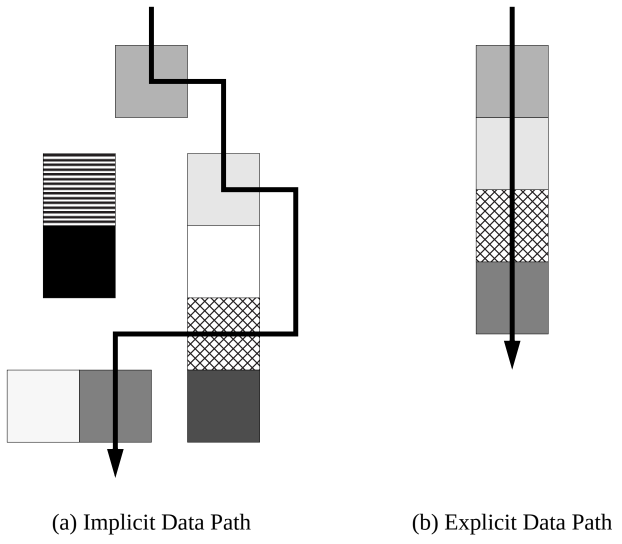


Figure 2.9: Finding a data-path

sentations of the data path they define (called order). The entity **order** represents the runtime representation of an order-type. While order-types define all operations, parameters, resources and data formats, orders are responsible to execute the defined operations with concrete data. The relation between order-type and order is similar to the relation between a piece of code and a program executing this code.

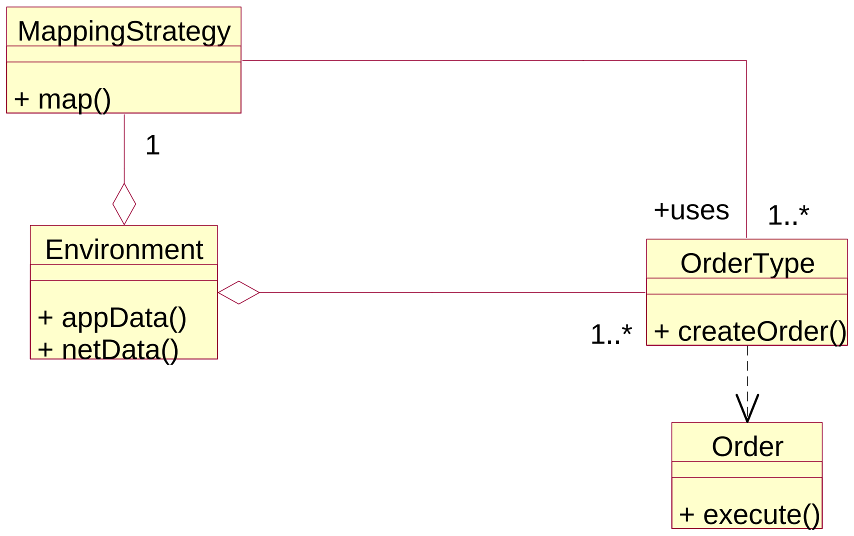


Figure 2.10: Data Path Reification: Class Diagram

In the context of *Data Path Reification*, the **environment** abstraction is considered as a container of order-types and responsible for identifying the right order for every piece of data coming from the application or the network. Once an order-type is identified, an order entity can be created,

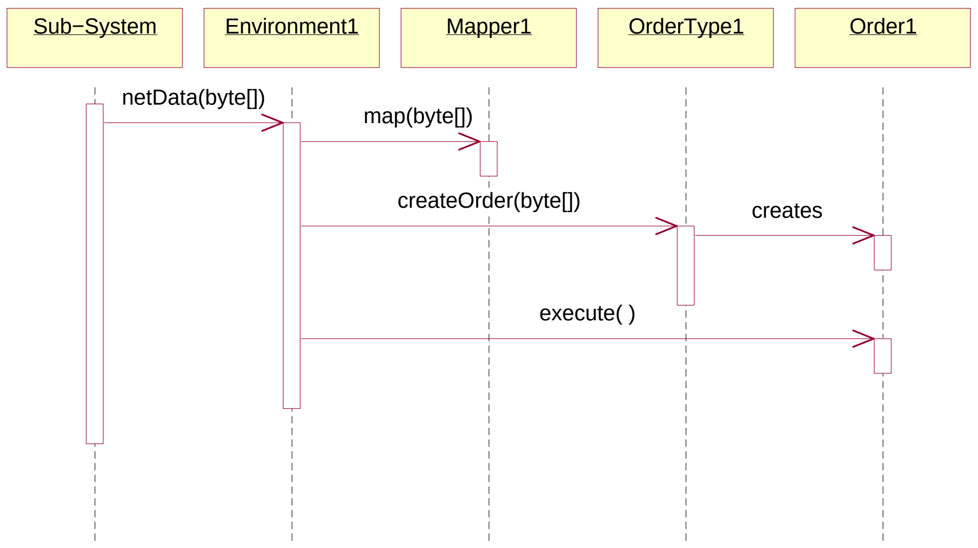


Figure 2.11: Data Path Reification: Sequence Diagram

initialized, and executed.

While for application data finding the right order-type can be done with information from the application (see *Data Path Classification* below), *network* data must be mapped to an order-type entity based on information carried by the message. The environment delegates this task to a mapping-strategy entity.

Mapping strategy entities encapsulate an algorithm that allows to map a piece of data to one order-type entity or at least determine that the data is not destined for any order-type. This may, for example, be done by using an unique identifier for every order-type, which is written to each packet sent after order execution. If a protocol consists of only one order type, the mapping strategy is extremely simple.

Scenarios

The following scenario describes the arrival of data from the network. The environment

1. is notified about the arrival of data from the network via `Environment.netData()`,
2. determines the right order-type entity from its mapping strategy via `MappingStrategy.map()`,
3. uses the obtained order-type entity to create a new order entity via `OrderType.createOrder()`, and
4. triggers processing via `Order.execute()`

For application data, the application can directly determine the order-type for its data via `Environment.appData()`. The indirection via the mapping-strategy is thus not necessary.

Discussion

The main benefit of *Data Path Reification* with respect to flexibility and reusability is that new services can be easily added and old ones removed, while the impact on existing services is minimal.

Testing and debugging is also simplified using a vertical structure. Instead of reconstructing possible data-paths a message may take through a protocol system to localize errors, *Data Path Reification* defines all possible data-paths in advance and allows to associate different messages directly to different data paths.

Moreover, *Data Path Reification* makes the dynamics of protocol software visible and allows to easily gather statistics and measurements about each order to identify possible bottlenecks and room for optimization. It is straight-forward to map orders to threads, to associate priorities, and to optimize scheduling. Each order type could implement its own thread or share threads with other order types, could use thread-pools, or delegate processing to a single system thread. Thus different process-models are supported as an independent aspect without involving the core protocol code.

Data Path Reification is to some regard a refinement of *Outsourced De-Multiplexing*. While the latter promotes the isolation of data streams from different applications, the former assures isolation of data streams of one application.

Known Uses

Data Path Reification emphasizes the data-path abstraction as central element to impose a vertical structure to protocol software. A data path is a very fundamental and natural abstraction used in many different contexts. Business organizations are structured along work-flows, plans are structured along related activities, the TV program is structured in channels with different emissions. In the context of computer systems, the idea of grouping related tasks that share resources comes in various shapes. The concept of concurrency [11] [13], which is applied in all modern operating systems (e.g. UNIX [90]), abstracts the execution of applications as *processes*. The *thread* concept [113], [34], [20] allows to structure application processes along related tasks. Modern languages like Java [136] give explicit language support for threads and concurrency issues to enhance the productivity of development. Birman [19] points out the importance of structuring distributed systems into threads to simplify implementation.

In the context of communication software, a number of optimization techniques are build around the abstraction of a path. Examples are *up-calls* [39], [10], Integrated Layer Processing (ILP) [38][29], or fast-path optimization techniques like TCP header prediction [73] or packet classifiers [95]. The communication-oriented operating system SCOUT [100] exploits a vertical structure to make execution predictable and controllable, and to exploit mechanisms to allocate resources based on specified quality of service requirements of a particular path. The notion of a data-path is thus very common in system engineering, especially for parallel and distributed systems. *Data Path Reification* has been successfully applied in different contexts and can therefore considered to be a pattern. However, it has never been used before to structure protocol specifications or protocol software.

2.3.4 Principle 3: Data Path Partitioning

Problem

While *Data Path Reification* vertically structures protocols and makes protocols extensible for new services, whole orders are still hardly reusable and changing them would still require expensive modifications. Due to the inherent complexity of distributed systems, the deployment of protocols often involves a lot of testing and debugging with various modifications and parameter configurations. Modifications may concern the operations performed during order execution, but also the associated *header information* of the defined messages. Protocol functions should be decoupled from each other, but also be independent from the message syntax.

Solution

We propose to partition the data paths identified into a data representation part and a functional part. The data representation part encompasses properties of message header information, the functional part the operations to be executed per path. The data representation part is further divided into entities, each of which is responsible to represent a header field. The functional part is further divided into entities, each of which encapsulates an atomic protocol function. We call this structuring principle **Data Path Partitioning**.

Figures 2.12 and 2.13 depict the resulting structure after applying *Data Path Partitioning*. The entity **entry-type** encapsulates types of message header fields (e.g. a sequence number, a checksum-value, or a time stamp). Entry-type entities contain information such as the number of bytes representing a header field, the range of a sequence number, or if big-endian [40] representation is used. An **entry** entity represents a concrete message header field information. Entry entities are exclusively instantiated by entry-type entities. Each entry entity can take a message (i.e. an array of bytes) as input, parses it, and transform its information extracted into a typed value (e.g. an integer). Each entry entity also provides functions to transform a typed

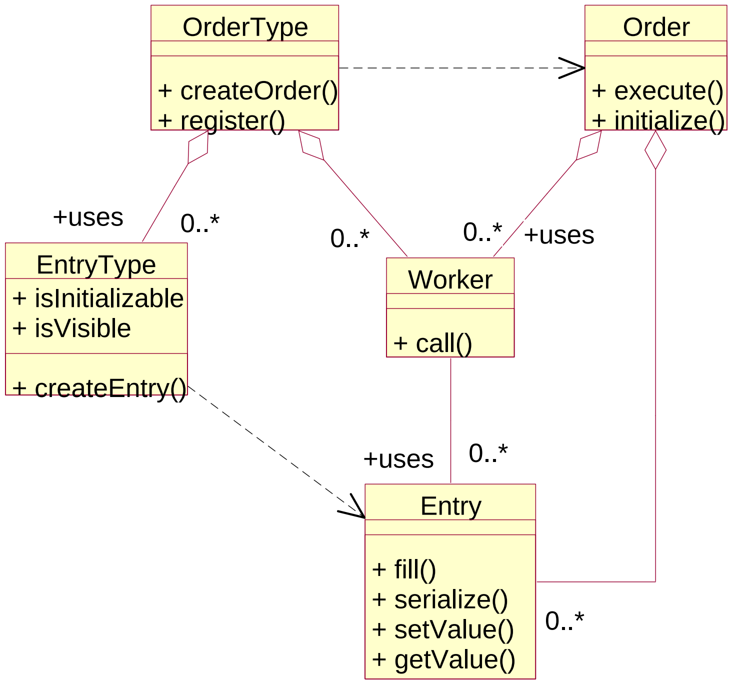


Figure 2.12: Data Path Partitioning: Class Diagram

value back into an array of bytes. Any entry-type entity can be configured **initializable** and/or **visible**. All entry entities created by entry-type entities configured *initializable* are fed with external data, i.e. data from either application or network. All entry entities created by entry-type entities configured *visible* make their data accessible outside the protocol entity, i.e. give the data as a typed value to the application or send it via the network after processing.

Figure 2.14 illustrates the meaning of *initializable* and *visible* based on a simple example scenario (execution of an order in output and an order in input direction, each consisting of two entries whereas the first entry represents a sequence number and the second entry represents a string). The figure depicts the data flow (arrows), the events during the execution of an order (numerated text fields), and the state of the entries (represented by rectangles) that are affected by the events. The string entry is initializable (represented by bold surroundings) and visible (represented by a colored rectangle) at both sender and receiver side since it represents application as well as network data. The sequence number, however, represents network data and is unknown to the application: the sequence number is thus visible only at the sender side (in output direction) and initializable only at the receiver side (in input direction).

When the application inserts data into the protocol instance (step 1), only the initializable string entry is concerned. The sequence number entry remains empty until a worker assigns a value to it (step 2). In step 3, the data of all visible entries (sequence number and string) are transformed into a byte array and sent. When the data arrives at the receiver (step 4), all initializable entries

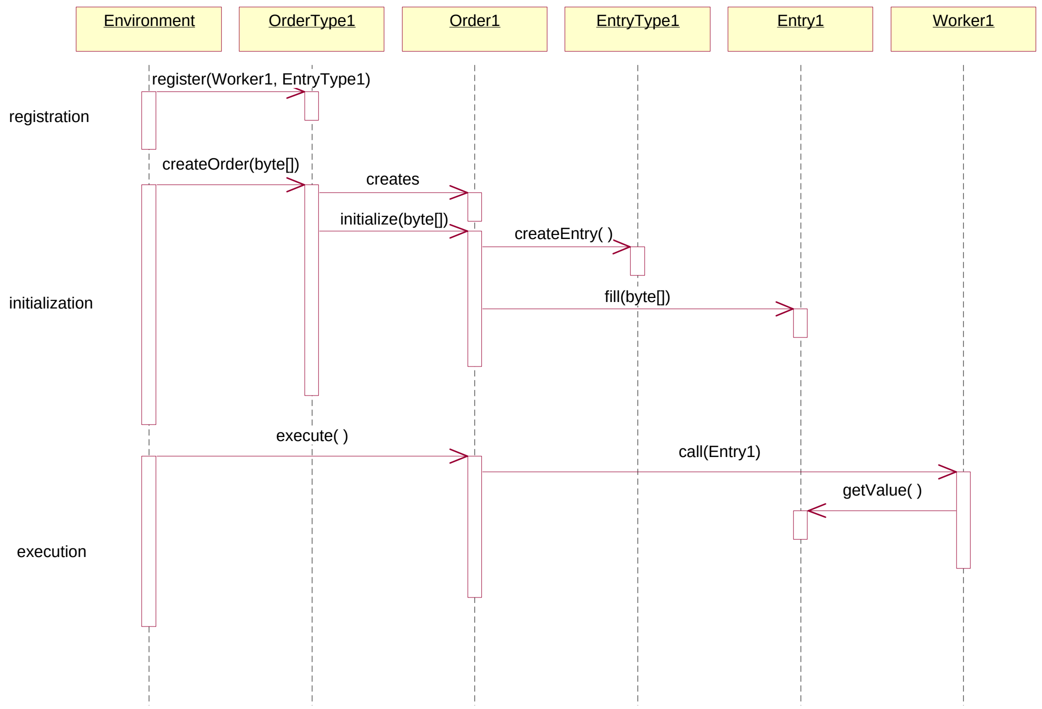


Figure 2.13: Data Path Partitioning: Sequence Diagram

(sequence number and string) are filled with data. Only the value of the visible string entry is delivered to the application (step 5).

A **worker** entity encapsulates a single protocol operation (e.g. sequencing, checksumming, round-trip time measuring). In contrary to a layer, a worker encapsulates either a function in output or in input direction. Another difference with a layer is that a worker does not operate on complete messages, but only on entry objects. Hence, it does not need to have any knowledge about the message format and does not need to perform parsing or message header manipulations.

In the context of *Data Path Partitioning*, order-type entities are defined by a set of entry-type entities and worker entities, and the relations between entry-types and workers.

Scenarios

The following scenario describes the composition of a protocol environment consisting of one order-type entity called *hello*. It consists of two entry-types called *seqnr* and *blob* and one worker called *count*. The worker uses the entry-type *seqnr* as parameter to increment and assign a sequence-number each time it is called.

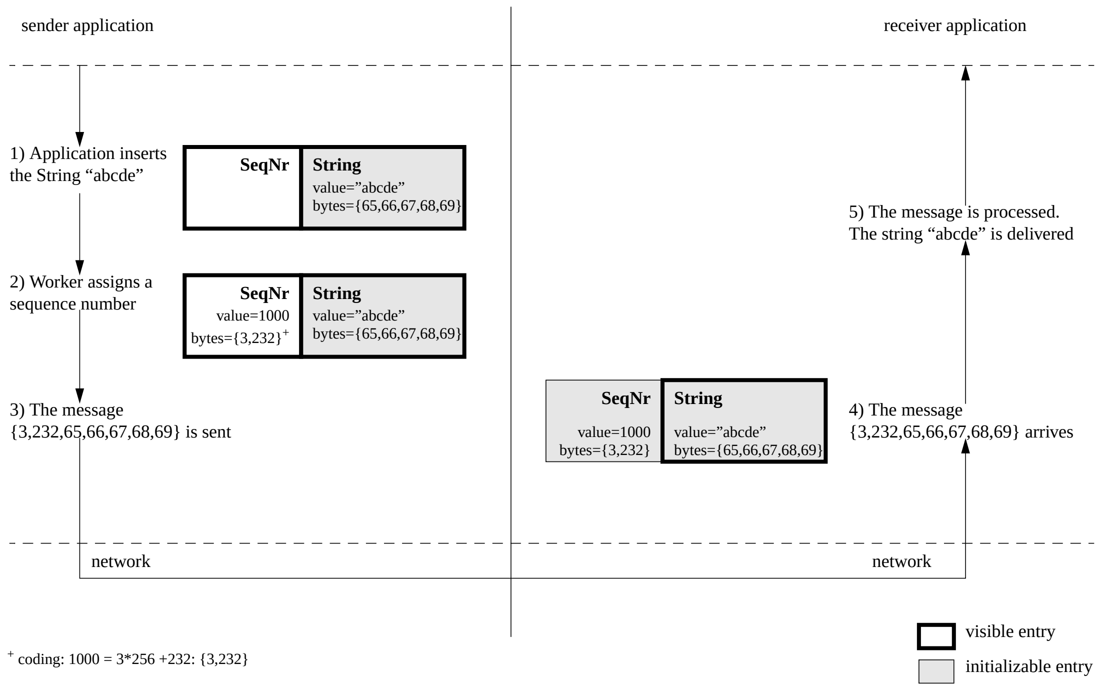

sender application
receiver application
1) Application inserts the String “abcde”
SeqNr
String
value=”abcde”
bytes={65,66,67,68,69}
2) Worker assigns a sequence number
value=1000
bytes={3,232}+
3) The message {3,232,65,66,67,68,69} is sent
4) The message {3,232,65,66,67,68,69} arrives
5) The message is processed. The string “abcde” is delivered
network
network
+ coding: 1000 = 3*256 +232: {3,232}
visible entry
initializable entry

Figure 2.14: Visibility and Initializability of entries

- all workers are registered with their parameter entry-types at the respective order-type, e.g. the worker *count* with its parameter entry-type *seqnr* is registered at the order-type *hello* via `OrderType.register()`
- workers and entry-types can also be registered alone, e.g. the entry-type *blob* is registered without being part of a parameter relation
- all order-types are registered at the environment via `Environment.register()`, e.g. the order-type *hello*

The following scenario describes how a new order is initialized with data. We assume that the entry-type *seqnr* is configured *visible* and the entry-type *blob* is configured *visible* and *initializable*.

1. the application gives data to the environment and specifies the respective order-type via `Environment.appData()`, e.g. for {1,2,3,4,5} as data `appData({1,2,3,4,5}, hello)`
2. the environment creates a new order using the order-type *hello* via `OrderType.createOrder()`
3. the order-type gives the obtained data, e.g. {1,2,3,4,5}, to the created order via `Order.initialize()`

4. during initialization, the order creates new entry objects via `EntryType.create()` (applied to *seqnr* and *blob*), and delegates initialization to all entry entities that are *initializable* (e.g. to the entry created by *blob*) via `Entry.fill()`

The following scenario describes the execution of the order created by the order-type *hello*.

- the protocol environment executes the order created and initialized by order-type *hello* via `Order.execute()`
- the order calls all its workers with their corresponding entries via `Worker.call()`, e.g. the worker *count* with the entry created by entry-type *seqnr*
- the workers obtain and manipulate the entry objects via `Entry.getValue()` and `Entry.setValue()`
- the order serializes all *visible* entries via `Entry.serialize()`, e.g. *seqnr* and *blob*, and sends the obtained byte array

To illustrate parsing, we depict the collaboration diagram of a more sophisticated example in Figure 2.15. Here, we have three initializable entries (Integer, Sequence Number, String), whereby the string entry performs parsing in dependence of the value of the integer entry. The diagram leaves out creation of order and entries and starts with the invocation of the `initialize()` method of order $O1$ where parsing starts. $O1$ gives the parameter array $\{4, 18, 2, 65, 66, 67, 68, 69\}$ to the integer entry *E1*, which takes the first byte (since $size = 1$) and represents it as an integer ($value = 4$). $O1$ now gives the byte array to the sequence number entry $E2$ and indicates that the first byte has already be taken. $E2$ thus starts with byte number 2, takes the next two successive bytes (since $size = 2$), and represents them ($value = 624$). $O1$ now gives the byte array to $E3$ and indicates that three bytes have already be taken. $E3$, which is configured to extract a number of bytes that correspond to the value represented by entry $E1$ ($size = E1.value = 4$), takes the next 4 bytes out of the byte array and represents them as a string ($value = "abcd"$).

Discussion

In contrary to the principles explained before, *Data Path Partitioning* trades off performance against good structure. Fine-grain modularity introduces an overhead due to indirections for calling functions. However, it also provides a lot of benefits with regard to flexibility and reusability. It decouples distinct protocol functions, input- from output-operations, and message parsing from processing and state maintenance. It also inherently provides support to share header fields (as entries) among different protocol functions (workers) without compromising information hiding.

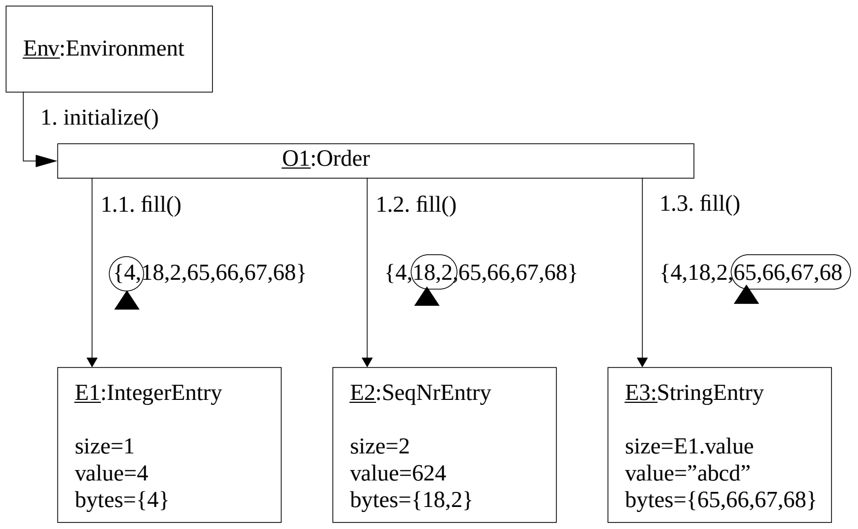


Figure 2.15: Parsing

One might argue that *Data Path Partitioning* is just a form of fine-grained layering. However, layers structure a system only in a horizontal manner, while *Data Path Partitioning* is based on a vertical structure. In contrary to layers, workers don’t deal with message formats or parse a message, but operate simply on arguments that may represent header fields. That way, global aspects of a protocol system like the message formats and the thread-model are separated from reusable algorithms and functions.

Known Uses

Data Path Partitioning as a refinement of *Data Path Reification* is also a well-known and widely applied concept. In the context of a business processes (e.g. a client orders an article by telephone), a worker corresponds to a certain task (e.g. talking with the client, fetching the article from the store, pack the article, preparing the bill, sending the article) and entries correspond to information needed during execution of the tasks (e.g. client number, name, address, article number, etc.). In the context of operating systems, a worker entity corresponds to a task, an entry corresponds to a resource. The relation between worker and entry-type also corresponds to the relation between the fundamental abstractions of function oriented languages *function* and *parameter type*. Both use parameterization to provide reuse and to separate state from operation. In the context of protocol implementation, Renesse [147] proposed a way to let different layers share header information that resembles our approach to sharing entries between workers.

How to model the control flow between workers?

The principles presented structure the *data* flow of protocol software by organizing the code in data-paths each of which contains a set of linear functional units (workers). However, they don't model the exchange of *control* information between workers, which is an important aspect of protocol software. The separation into input- and output functionality yet amplifies this need of interaction between workers.

While a uniform interface like the `Worker.call()` function is sufficient to model data flow, the exchange of control information requires much more flexibility. Ideally, a worker should be able to access any method of any other worker. However, to avoid the coupling of workers, a worker should not make any assumptions about the specific interface or even the concrete type of another worker. When one worker would depend on another worker to effect its functions, both flexibility and reusability of the worker would be seriously compromised. Our problem is therefore how entities communicate without having any knowledge about each other and the context they are used in.

A very common technique to model interactions between entities is the use of events. There is a design pattern called *Publisher/Subscriber* that allows a *subscriber* entity to register for events fired by a *publisher* entity. The interaction protocol between both is the type of the event. This model reduces the coupling of two entities to the common knowledge of a certain event type. The publisher entity does not know anything about the subscriber entity besides the fact that it is interested to be notified about a certain event. Coupling is shifted to an abstracter level. However, both entities are still – though more loosely – coupled.

A simple way to circumvent this coupling is the construction of so called *event adapters*. Event adapters know both, the event type expected by the subscriber entity and the event type provided by the publisher entity. Instead of registering a subscriber entity at the publisher, the subscriber is registered at an event adapter entity and the event adapter is registered at the publisher. When the publisher fires an event, the event adapter is notified, maps the event received to an event of the type known by the subscriber, and finally notifies the subscriber. While this model introduces a level of indirection, it provides full flexibility in the sense that arbitrary entities that follow the publisher/subscriber model can interact without loosing their independence. Event adapters can be generated automatically – this is what visual builder tools for component-based development are doing. The difference between *Event Adapters* and *Publish/Subscribe* is illustrated in Figure 2.16.

In the next chapter, we will address the design problem of connecting arbitrary components in more detail and propose a concrete solution.

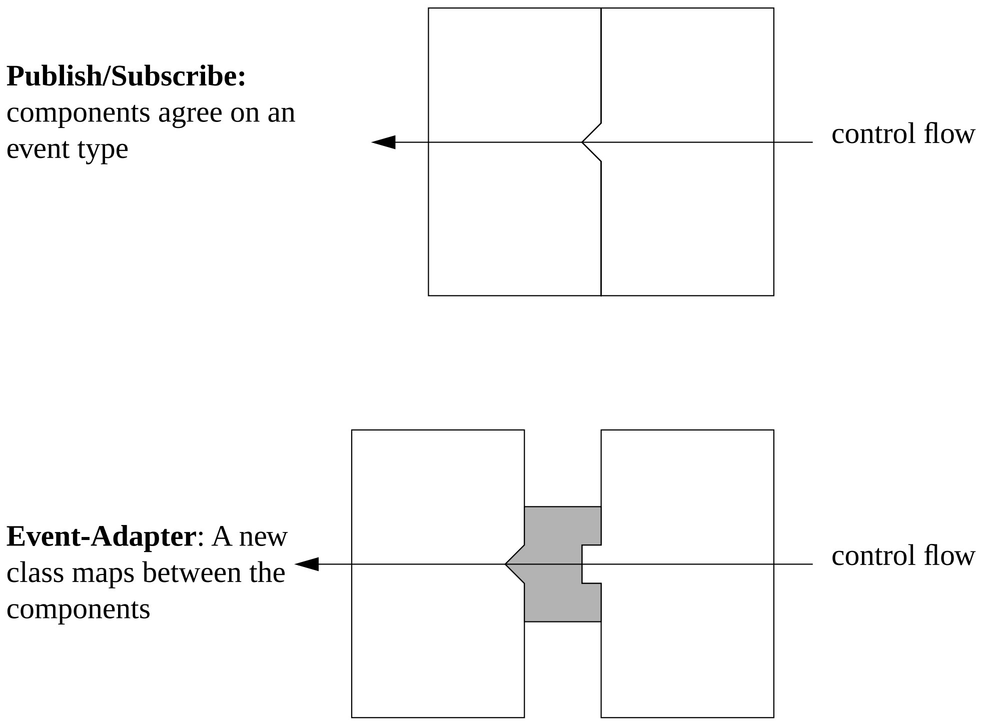

Publish/Subscribe: components agree on an event type
control flow
Event-Adapter: A new class maps between the components
control flow

Figure 2.16: Publish/Subscribe vs. Event Adapter

2.3.5 Principle 4: Data Path Classification

Problem

Our solutions presented above deal with the extensibility, reusability, and flexibility of protocol entities. However, they are not concerned about how the protocol part interacts with the underlying network and the application that uses it. The overall goal is to completely decouple the protocol part from any application or network specifics. On the one hand, the interfaces to application and network should be as general as possible. On the other hand, there are many different ways applications may like to communicate with a protocol entity. Additionally, network interfaces are typically complicated to use and platform dependent. How can generic and flexible interfaces be provided without compromising the independence of the protocol entity.

Solution

We follow the vertical structure imposed by *Data Path Reification* and propose to classify the data paths identified (i.e. the order-type entities) according to their interactions with entities outside the protocol entity. We call our concept **Data Path Classification**. Interactions are limited to getting information or providing information. The entities outside the protocol entity belong either to the application or the network service. After classification according to this two dimensions – application/network, get/provide – we obtain four major types of data paths given in Table 2.1.

Data Path Type	Direction	Interface	Attributes
Acceptance	Output	From Application	Write Access Point for the application
Delivery	Input	To Application	Read Access Point for the application
Emission	Output	To Network	Output Port Number
Reception	Input	From Network	Mapping Information, Input Port Number

Table 2.1: Classified Data Paths

Data paths that are initialized with data from the application are classified as **accepting**. Data paths that are initialized with data from the network are classified as **receiving**. Data paths that deliver data to the application after processing are classified as **delivering**. Data paths that end up emitting data over the network are classified as **emitting**. Frequently, a data path abstraction may belong to more than one classification: accepting and emitting data paths (passing the protocol entity in output direction from application to network) as well as receiving and delivering data paths (passing the protocol entity in input direction from network to application) are very common. A data path that does not belong to any class, is refered to as **internal** data path.

When we apply the classification to the order/order-type abstraction, we obtain the entities depicted in Figure 2.17. Accepting data paths are mapped to **acceptance** and **acceptance-type** entity classes, which are specializations of, i.e. inherited from, order and order-type entity classes. Receiving data paths are represented as **reception** and **reception-type**, respectively. Delivering data paths map to **delivery** and **delivery-type**, emitting data paths to **emission** and **emission-type**.

Read-API entities are proxies of the application to be notified about data delivery. Every delivery-type is associated with only one read-API entity, which may be shared by different delivery types. **SAP** entities are wrappers for network services. Each emission-type entity is associated with one SAP entity used by the created emission entities to emit data. SAP entities are also used to receive data. **Write-API** entities are proxies of the application to give data to the protocol environment. Write-API entities are associated with acceptance-types.

Scenarios

The first scenario describes how an environment *accepts* data from the application.

1. the application gives information to a write-API entity via `WriteAPI.accept()`

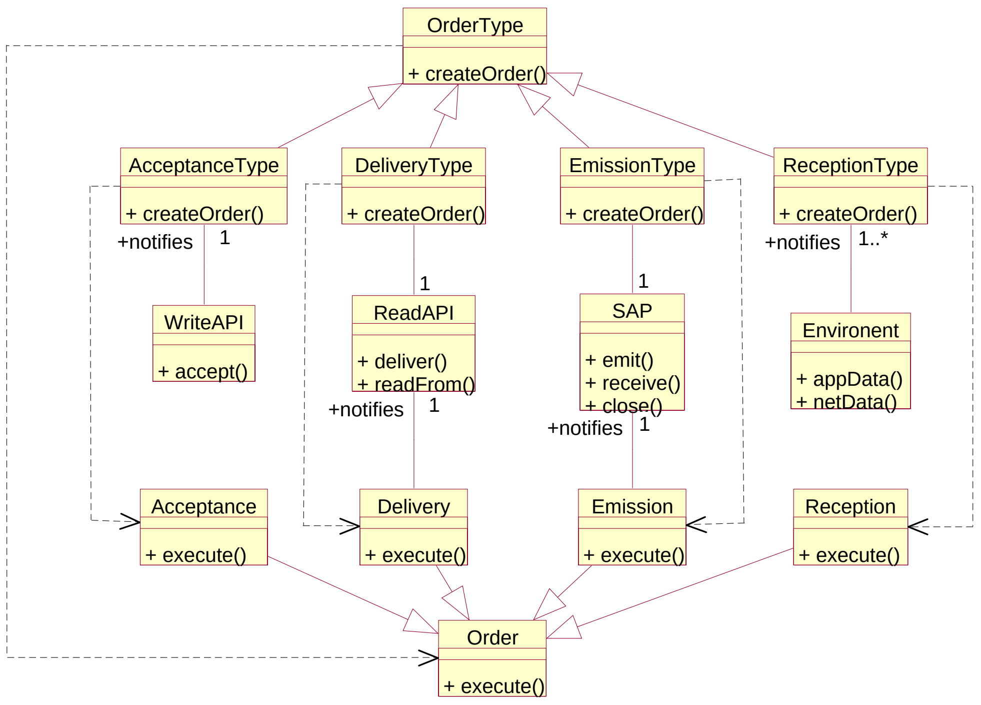


Figure 2.17: Class diagram of Data Path Classification

2. the write-API forwards it to the environment via `Environment.appData()` specifying data and acceptance-type

3. the environment uses the acceptance-type to create and initialize a new acceptance and execute it

The second scenario describes how an environment *delivers* data to the application.

1. within `Delivery.execute()`, when all workers have been called, the values of all visible entries are collected and given to the associated read-API entity calling `ReadAPI.deliver()`, where the information is consumed by the application

The third scenario describes how an environment *receives* data from the network.

1. the environment receives network data via `Environment.netData()` and maps the data to an reception-type entity via `MappingStrategy.map()`

2. the environment creates a new reception via `ReceptionType.createOrder()`

3. the environment executes the reception via `Reception.execute()`

The fourth scenario describes how an environment *emits* data to the network.

1. within `Emission.execute()`, when all workers have been called, the values of all visible entries are serialized into bytes and sent via `SAP.emit()`

Discussion

Data Path Classification extends *Data Path Reification* by carrying over the vertical structure from protocol structuring to interface modeling. An application can access each service provided individually in a straight-forward manner, since each service is represented by its own entity. The application is notified about the arrival of new information without the need of further identifying the information. The information is directly usable by the notifying thread, the application does not need to poll a queue to obtain it. Each application developer can implement the interfaces as it is needed for the application.

The classification into different order-type categories also enhances the structure of the protocol code and makes the interaction of protocol with application code predictable. For example, knowing the number of reception order-types in advance facilitates and improves mapping from network data to order-types.

Open Questions

Data Path Classification is only concerned about the interface to application and network from the perspective of the protocol implementation. It does not prescribe how incoming data is read from the SAP entities and given to the environment entity. Choosing the right thread-model, the appropriate programming style (e.g. reactive versus concurrent), and exploiting the operating system facilities is rather a design issue during application development and not subject to protocol modeling.

There exist a number of design patterns successfully applied in the ACE [127] framework that exactly address these issues: the *Reactor* [122] pattern describes how operating system events like connection requests or data arrival that are delivered concurrently can be handled without coupling application specific and system specific mechanisms. How to adapt the *Reactor* to the specifics of the Java language can be found in [44]. The *Active Object* pattern [123] allows to introduce concurrency and simplify synchronization of different threads. The *Acceptor-Connector* pattern [124] decouples connection establishment from processing.

Known Uses

Most stackable protocol systems provide uniform interfaces to protocol entities. *Data Path Classification* makes interfaces first class abstractions that vary depending on the structure of protocols. However, the notion of input (towards the application) and output (towards the network) is common to all existing systems and reflected in all protocol systems we know.

Some systems use even a similar terminology and allow a direct mapping to the four main data-path categories we identified: x-kernel [68] layers provide the functions `xPushTo` (emit), `xPushFrom` (receive), `xPopTo` (deliver), and `xPopFrom` (accept). STREAMS [145] uses the terms *user write* (accept), *user read* (deliver), *device out* (emit), *device in* (receive).

Since the *Data Path Classification* principle depends on *Data Path Reification*, we don't consider it as a pattern, but rather as a natural refinement of the *Data Path Reification* pattern.

2.4 Summary

In this chapter, we first showed why the traditional layered structure – even when applied as fine-grained layers – is not able to meet the requirements of application-tailoring with regard to flexibility, decoupling, and reusability. We then presented an alternative structuring approach that consists of a set of structuring principles. These structuring principles can be considered architectural or structural patterns. The common goal of these structuring principles is to make all parts of the protocol software reusable by assuring decoupling, fine-granularity, and flexibility.

The main abstractions introduced are:

- **worker**: encapsulates a single protocol operations during processing
- **entry**: encapsulates representation and manipulation facilities on message header fields
- **order**: represents one uni-directional data path built of workers, entries, and the relations between workers and entries

Figure 2.18 presents the dynamics of a system that implements all structuring principles. The arrows symbolize the data flow between applications.

In our model, reusability and flexibility are achieved as follows.

- The streams of different protocol sessions are decoupled (*Outsourced De-Multiplexing*)
- A protocol can be extended by new services without interfering with existing services (*Data Path Reification*)
- A functional unit does not need to know about what need to be done next (*Data Path Reification* and *Outsourced De-multiplexing*)
- A functional unit does not need any information about where in the message its relevant header information can be found (*Data Path Partitioning*)

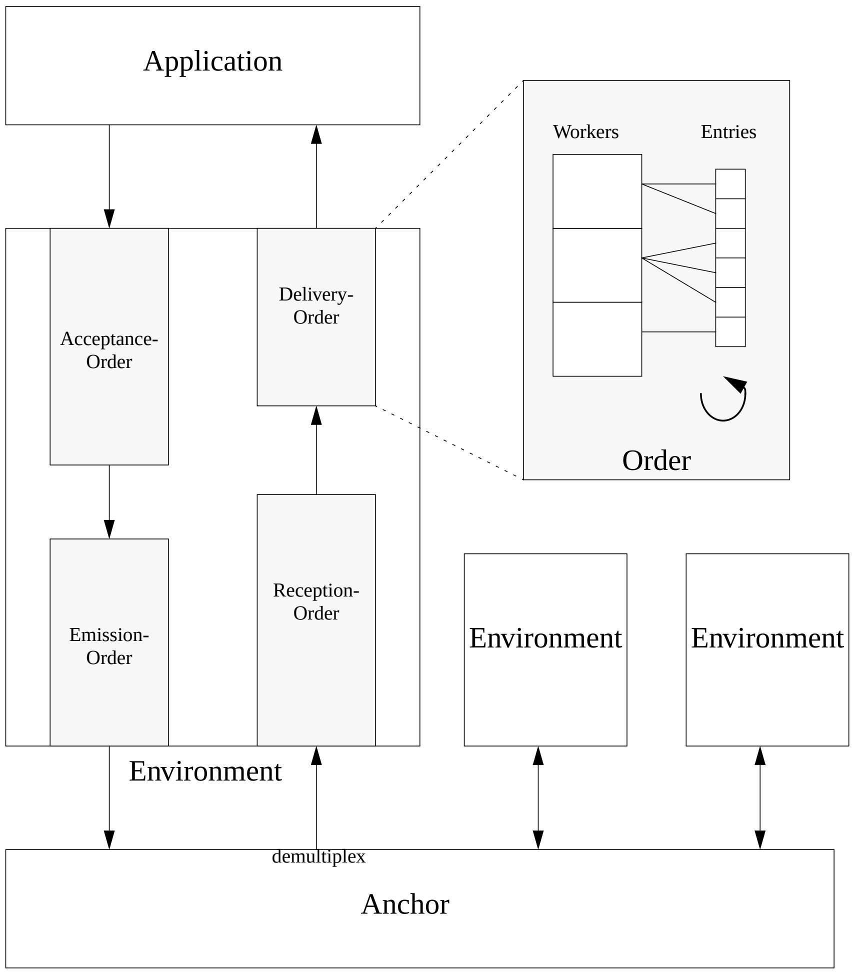


Figure 2.18: Architecture following our Structuring Approach

- Header representation units can be reused across different protocol implementations (*Data Path Partitioning*)

- A functional unit does not need to know where input information comes from, where output information goes to, who is handling the events he fires, who fired the events he is notified of, and which kind of order he creates and what is happening with this order (*Data Path Partitioning*)

- The protocol does not need to know anything about the underlying network or the application that uses it (*Data Path Classification*)

While the principles described focus on the modeling of application-tailored protocols, they may also be considered as a way to structure a multi-functional protocol layer itself. Hence, the vertical structure should be seen rather as a complementary than exclusive concept to traditional layering.

Chapter 3

The PITOU Protocol Development Environment

In this chapter, we present the PITOU[1] framework that aims to support composition, rapid prototyping, and testing of application tailored protocol software. PITOU is based on the structuring principles introduced in Chapter 2 and implemented in the Java [137] programming language.

After a discussion of the principal goals and design issues of PITOU, we sketch PITOU's overall architecture and show how the structuring principles of Chapter 2 have been realized, adapted, and refined with regard to design issues and implementation details. We then describe how PITOU is used to i) implement new components and to ii) assemble implemented components into application-tailored protocol software. Concerning the assembly of components, we first examine the Java Beans [75] component model, and then present an own approach that aims to overcome the drawbacks of Java Beans. Finally, we give an overview of the patterns we used during the design of PITOU and review related work.

3.1 Related Work

There exist a number of frameworks to support the implementation of distributed applications and protocols. Why do we propose another one? The reason is simple: we don't know of any framework that supports flexible and easy composition of protocol software out of reusable components. Most frameworks apply coarse-grained layering. Few frameworks apply fine-grained layering. The only one (Coyote [15]) that applies fine granularity without layering is neither portable, nor easy to use, nor does it support plug-and-play reusability.

Pioneering work in the area of protocol frameworks is the x-kernel environment [68] of Hutchin-

[1] PITOU = Protocol Implementation, Tailoring and Organization Utilities

son and Peterson. The x-kernel framework resides in the operating system kernel and allows protocol systems to be stacked by imposing strict layer boundaries and providing a uniform interface for each layer. A similar approach is followed by the STREAMS [145] environment. Besides the goals to ease protocol implementation and to improve the flexibility of constructing protocol systems, x-kernel and STREAMS have not a lot in common with the PITOU framework. They reside in the operating system kernel, follow a strictly layered architecture, and provide no support for fine-grained composition of application-tailored protocols.

A bit more related to the PITOU framework are a number of user-space frameworks that partly use object-oriented techniques. iBus [93] is a Java framework specialized to group communication protocols. Horus [148] and its successor Ensemble [63] implement fine-grained layering to improve the flexibility of group communication protocols. Channels [22] is implemented in C++ and promotes modularity while it breaks with strict layering. Modules can be connected via de-multiplexing modules and form complicated protocol graphs. However, the reusability of the modules is largely compromised since message information is tightly tangled with the code of each module. Conduits [64] follows a similar approach as Channels and is the first to emphasize the importance of using design patterns to improve the internal structure of coarse-grained abstractions. BAST [60] is an object-oriented framework to support the development of fault-tolerant distributed applications. BAST protocols also follow a hierarchical organization, but instead of stacking layers, existing protocol classes are extended by new functionality. Hence, BAST promotes reuse by inheritance, and can be classified a white-box framework. While BAST allows for fine-grained improvements of services, it is not possible to compose new protocols by putting existing objects together.

The work closest to PITOU is the Coyote [15] framework, an extension of the x-kernel to overcome problems with regard to flexibility and configurability. Coyote structures x-kernel protocol layers into a set of fine-grained modules called micro-protocols that can concurrently read and process messages from message bags. To avoid direct knowledge between modules, modules communicate via events. The overall goal of Coyote is to *simplify development and to increase the configurability of the network subsystem* [14]. Coyote has similar goals as PITOU and encapsulates atomic operations in fine-grained modules. There are, nevertheless, a number of differences with regard to architecture and implementation. Coyote protocols reside in the kernel, PITOU protocols are integrated in the application. Coyote structures protocol layers as a set of loosely coupled fine-grained modules, PITOU as a set of data paths. Coyote uses a simple event mechanism to decouple protocol functions, PITOU applies a rigid structure in combination with explicit reification of interaction to achieve decoupling. Coyote is not supported by component-based development techniques and tools, since it does not provide the degree of reusability to construct new protocols from existing components without code modifications.

The use of protocol languages is another approach to implement protocols. Formal description languages like LOTOS [24], Estelle [102], Esterel [32], or SDL [27] focus on the specification and verification of communication protocols, and provide the automatic derivation of code skeletons from specifications. However, the generated implementation skeleton still needs to be filled with hand-written code. Furthermore, the specification code is not easier to write, understand, or maintain than the code of modern general purpose programming languages. We belief that frameworks are a more promising approach to protocol implementation since they allow rather easily for structural extensions and modifications compared to languages. Due to their inflexibility, most language based approaches fail in covering all aspects of a protocol, and thus require the use of different tools and manual intervention.

The use of SE techniques in software system research is not as new as it is for protocol systems. The 2K operating system [85] is based on software components that can be configured and customized at runtime. The Quarterware [131] system allows to configure and customize middle-ware components. ACE [127] supports the implementation of communication systems rather than composing protocol software.

3.2 PITOU Design Issues

3.2.1 Terminology

In order to avoid misunderstandings, we shortly introduce the most important terms we use.

The Java [137], [61], [55] programming language we used for the implementation of PITOU is an object-oriented language. Hence, our principal programming abstraction is an **object**. We refer to a **class** as the *description of a group of objects with similar properties, common behavior, common relationships, and common semantics* [118]. Objects are also refered to as **instances** of a class. Classes are often called the **type** of an object. However, there is an important difference between a class and a type: a class can directly be instantiated as an object, a type not. Every class has a type, but not every type is a class (see also below for `abstract` class and `interface`). A **method** is the object-oriented term to refer to a function.

Java provides support to separate definition and implementation of classes and methods. Methods that are defined, but not implemented are called **abstract** methods. A data type that defines abstract methods is called **abstract class** and specified by the `abstract` keyword. A data type that does not provide an implementation at all, but solely the definition of accessible methods (which are implicitly abstract) is called a Java **interface**. To avoid confusion with what we commonly call an interface (i.e. shared boundaries between components, devices, or whatever entities), we use `code font` to refer to a Java `interface`. Abstract classes and `interfaces` are data types, but not classes, since they can not be instantiated as runtime objects. They serve as an instrument of typing (`interface`) and as a code skeleton (`abstract class`) to be used to implement new classes.

Inheritance relations in Java are expressed by the keyword `extends`. We therefore use the term *extends* to say *inherits*.

A protocol environment object in the PITOU protocol framework passes through three different phases during its life-cycle:

1. During the **construction phase**, all objects defining the structure of the protocol environment are instantiated, configured, and registered.

2. During the **initialization phase**, which is triggered by calling `Environment.init()`, the internal structure is built.

3. During the **execution phase**, a protocol environment is able to accept and receive information and to execute orders. When the execution phase terminates, an environment object will be deactivated and cannot be used anymore.

Note that these phases do not refer to the process of implementing a protocol. Construction, initialization, and execution phase happen all at runtime.

During the execution phase of a protocol, we distinguish three different operations with regard to orders.

- We say an order is **requested**, when any component or framework class wants a new order to be executed. A request is made to the corresponding order-type object.

- We say an order is **initialized**, when the order obtains all its relevant data to be executed. During initialization of an order, messages may be parsed and all entries are filled with concrete information.

- We say an order is **executed**, when all workers are called.

3.2.2 Framework Tasks and Goals

The development of PITOU is motivated by the following goals:

- **Prove of concept**: the primary goal of the PITOU framework is to show that the structuring principles presented in Chapter 2 hold their promises with regard to flexibility and reusability of protocol components.

- **Protocol library**: tightly coupled with the first goal is the second goal of building up a library of reusable and flexible protocol components.

- **Ease-of-Use**: a major concern in the design of PITOU is that it should be easy to use for any developer, especially for those who do not have experiences in the area of protocol implementation. Once the basic concepts are understood, it should be straightforward to extend the protocol library and plug existing components together.

- **Rapid prototyping and testing**: rapid prototyping and flexibility are considered more important than performance oriented issues.

- **Evaluation of SE concepts**: the PITOU framework also intends to allow for an evaluation of usefulness and power of SE concepts like OO frameworks and component-based development for implementing communications protocols.

3.2.3 Overall Organization – State of the Art

An important issue in implementing protocol systems is how to partition protocol stacks between kernel and user-space. Kernel implementations provide automatically good security and performance, while user-space implementations are easier to write, debug, and modify.

In most Unix [90] and Windows like operating systems, only the highest layer (application) is in user space, while all other layers are in the operating system kernel. This organization is often refered to as monolithic kernel implementation [83] and follows rather the craftsmen-principle than modern software engineering principles. Implementation costs are very high, since writing, debugging, and porting kernel-code is very difficult. Extensibility and potential for tailoring are extremely poor.

Moving the complete protocol stack (besides the device drivers) into user-space can mitigate the implementation costs. One approach dedicates a separate user-level server for each protocol stack [99], [91]. While this organization makes implementation and debugging of the protocol code easier compared to the monolithic kernel implementation, it neither contributes to a better structure of the code, nor allows it to tailor protocols to the needs of the application. This organization is often refered to as monolithic user-space implementation.

A more radical approach organizes protocol code as libraries that can be linked into the application code [141], [47], [30]. Besides the goal of easing programming, debugging, and maintenance, this approach claims to improve protocol performance by exploiting application-specific knowledge.

The concept of Application Level Framing (ALF) by Clark and Tennenhouse [38] proposes to involve the application in the data transmission process and exploit application knowledge in the whole communication system. In particular, the application determines the boundaries of the data units to be processed (refered to as application data unit – ADU), which must be preserved at all levels of the system. The ADU is at the same time unit of transmission, unit of control, and unit of processing. ALF thus contradicts the layering principle. Although its original definition is more restricted, the term ALF has been reduced to be a synonym for any kind of application-tailoring of protocols. RTP [128] is a protocol (or better a definition of message headers) that claims to follow application level framing (although it is located on top of a traditional UDP/IP stack). While very few systems are based on application level framing (see [36]), the ALF paper has brought application-tailoring to prominence and motivated a number of systems that promote adaptable and flexible communication systems.

PITOU is implemented in user-space, and PITOU protocols are incorporated into the application. An overview of the overall organization of our approach is depicted in Figure 3.1.

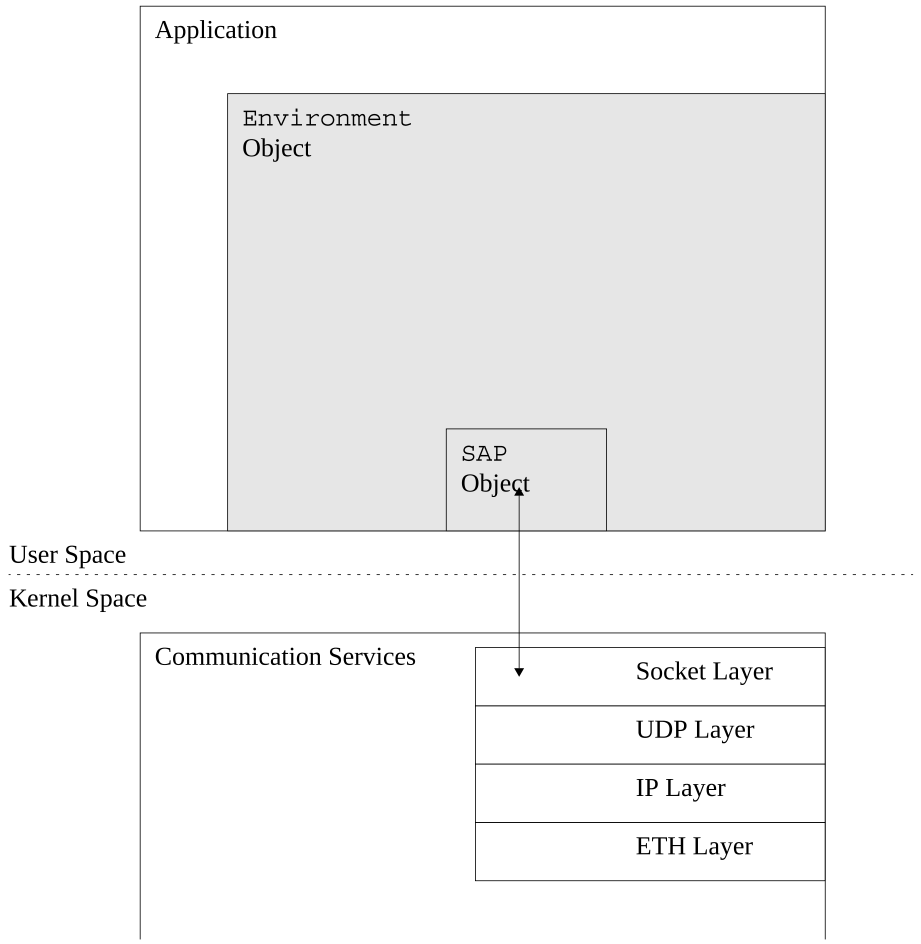


Figure 3.1: The Overall Organization

3.2.4 Programming Language

While Java has become very popular in the last years, there has been little work using Java to implement communications protocols. Krupczak [88] implemented the AppleTalk [129] protocol suite to determine the trade-off between the speed of deploying new protocols and the performance penalty payed for portability. He found that the performance costs of using Java-based protocols is equivalent to four years of hardware performance gains.

Apvrille et al. [8] used Java to implement a protocol to support video on demand applications, and made detailed performance comparisons with a corresponding C implementation in user-space. They find that the Java implementation is not only capable to cope with the applications' requirements, but nearly achieves the same performance as the C implementation.

The JChannels [79], [81], [80] project at Eurecom in cooperation with Siemens had the objectives to experiment with fine-grained protocol implementation and to obtain insights about Java as programming language for network protocols. Despite the difficulties to access the LLC layer (DLP interface for Solaris) using Java and the performance penalty experienced for a modular implementation of TCP [62] that roughly confirmed the results of [88], we decided to use Java for a number of reasons.

- **object-orientation**: first of all, Java is object-oriented and thus allows us to easily map our abstractions to classes and objects.

- **library**: Java comes along with a comfortable library, which facilitates a number of operations needed for protocol implementation, e.g. efficient manipulation of byte-arrays, or parsing.

- **portability**: Java source code is compiled into a specific byte code, which is interpreted during execution by the Java Virtual Machine (JVM). The JVM thus hides operating-system dependent calls, and makes Java programs highly portable.

- **reflection**: Java provides a set of classes that support to analyze code and reflect about program structures at runtime, and allows to dynamically load new applications into the JVM.

- **concurrency**: concurrency support is crucial to simplify implementation of communication systems. Java provides classes and language support for multi-threading and synchronization.

- **garbage collection**: the JVM comprises a garbage collector, which automatically reclaims memory of objects that are no longer referenced to avoid memory leaks, one of the most frequent and hardest to eliminate bug of system software.

- **component model**: another argument for Java is its component model called Java-Beans [75], which is supported by an increasing number of tools and programming environments.

3.3 The Core Framework Design

3.3.1 Principal Design Elements

Based on the structuring principles introduced in the last chapter, we shortly review the elements that will play the principal role in the framework design.

- A protocol **environment** replaces the notion of a protocol stack. Instead of being organized in different layers, a protocol environment provides all protocol services that a particular application requires, and integrates all the corresponding functions. Protocol environments are implemented in the class `Environment`.
- An **order** defines a data path through a protocol. To separate the static characteristics of a data-path (like message formats and defined functional steps) from the dynamic characteristics (like concrete message and execution of the functional steps), a data-path is represented by two kinds of types: the `interface OrderType` and the `interface Order`.
- An **entry** encapsulates a message header field. An entry is represented by the two abstract classes `EntryType` (encapsulating static information) and `Entry` (encapsulating dynamic information, i.e. concrete data)
- A **worker** encapsulates an atomic functional step in protocol processing. A worker is represented by the abstract class `Worker`.

`EntryType`, `Entry`, and `Worker` are so called *hot-spot* classes since they represent the plug-in points, the interface between framework and application. Classes that implement `EntryType`, `Entry` and `Worker` are part of the framework library. However, the classes that implement `OrderType/Order` are needed to make the framework work, and are thus called *core framework classes*.

3.3.2 Applying the Structuring Principles

Since Chapter 2 already talked in detail about the various abstractions and their interactions, we will sketch only briefly how our structuring principles are applied during design and implementation, and point out the differences between the idealized solutions presented in Chapter 2 and design specific modifications.

Outsourced De-Multiplexing

The protocols we intend to implement are user-space implementations on top of UDP. Since UDP already provides multiplexing and de-multiplexing, applying *Outsourced De-Multiplexing* for this family of protocols requires no additional efforts: the Internet protocol stack in the OS kernel acts as anchor layer and de-multiplexes data to the right UDP socket, from where the data is read and given to the associated environment.

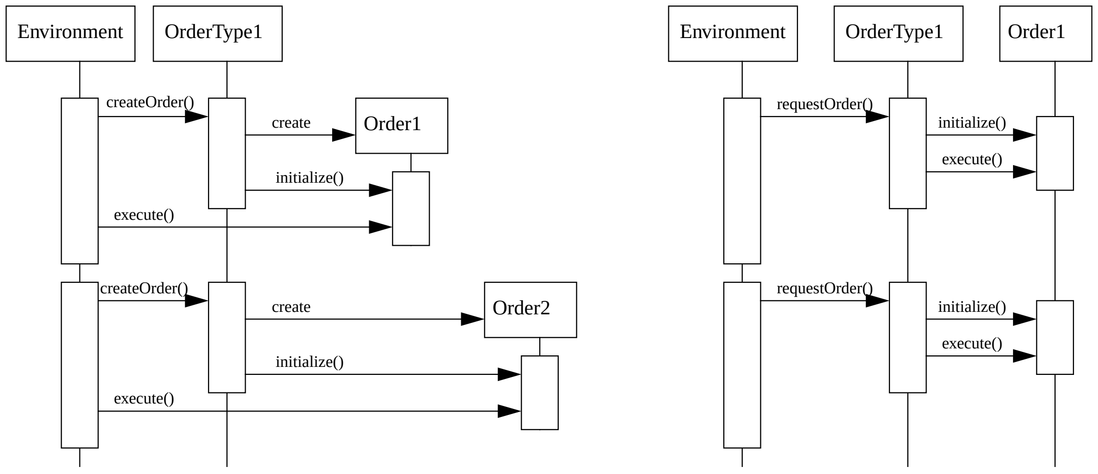
For applications that want to run different protocols over the same UDP socket, we provide a class `Anchor`. However, we have not built an application yet, where the anchor class was needed.

Data Path Reification

Each time new data needs to be processed, *Data Path Reification* proposes to create a new order object, fill it with the data, and execute it. However, object creation is a major cause of overhead in all object-oriented languages. We therefore decided to slightly modify the interactions between order-type and order objects, and apply the concept of **object pools**: instead of creating and throwing away objects with a short life cycle, a fixed number of objects of the respective class are created when the application is initialized, and reused with different information during the whole life-time of the application. The core class `InternalOrderType` implements the `interface OrderType`, and realizes the object-pool by maintaining a *single* reference to an order object ("**one-object-pool**"). This order object is created when the protocol environment is initialized. Instead of creating a new order object for each request via `Order.createOrder()`, a method `OrderType.requestOrder()` is used to re-initialize its order object with the new data and executes it. When the request for another order is made during the execution of the order, the request is written to a queue. After execution the oldest request is taken from the queue to re-initialize and execute the order object again. The class `InternalOrder` implements the `Order interface` to provide the functionality described above. Figure 3.2 illustrates the modified design.

Besides avoiding the creation of a lot of objects during the life-time of a protocol, the use of an one-object-pool has another advantage. It assures that two orders of the same type are never executed in parallel and thus avoid race conditions.

It may be useful that data of different data-path types are sent within a single packet. We call packets that carry information of different data paths **collection packets**. PITOU allows each data-path that ends up in sending data to be configured for collection. Sending will be delayed



Original Model

Execution implies the creation of a new order object.

One-Order-Pool

Execution is done always on the same order object (created during initialization of the correposning order-type object).

Figure 3.2: Use of an order pool

for a configurable time-interval to wait if some other data can be sent in the same packet. The most important benefit of sending collection packets is that network load can be reduced. RTP [128] *compound messages* or TCP [109] picky-backing are examples for sending data collected from different data-paths. Campbell [119] describes collecting packets as a general design pattern.

Data Path Partitioning

The use of a ”one-object-pool” is also applied to reduce the creation of entry objects during protocol processing. Each entry-type object maintains a reference to one entry object, which is given during initialization of the environment to the respective order object. Each time the order object obtains new data to process, it delegates parsing to its entry objects instead of creating new ones every time.

Another important design issue concerning *Data Path Partitioning* is the use of one worker object in different orders. We decided to allow to share workers among objects (in contrary to entry-type objects), because it is an easy way of sharing resources and state among orders, and it reduces the memory footprint. For example, imagine a protocol with a dozen of orders, which all perform checksumming: it is more efficient to use the same checksumming worker object for all orders than creating one for each order. Since different orders may be executed in different threads, all workers should foresee the case of concurrent access. The easiest way to synchronize access is to specify the `call()` method of a worker with the Java specifier `synchronized`.

We discourage to use one entry-type object across different order-types. Of course, all methods of entry-type and entry-objects could be specified `synchronized` to avoid side-effects due to concurrency. Maybe one could even find examples where two different order-types require entry-types with exactly the same properties. However, we think it is more proper to avoid sharing entry-type objects, since each entry-type represents some specific – physically different fields within different messages.

Data Path Classification

Applying *Data Path Classification* means to map the different data-path categories to `interfaces` and combine them with the types `OrderType` and `Order`. The number of combinations lead to a rather complex type hierarchy. Figure 3.3 depicts the combined `interface` types, and the classes that implement these interfaces. The hatched area contains all the `interfaces` needed to represent the different types of data-paths. The other types are the classes implemented in the PITOU framework. The core functionality is implemented in the classes `InternalOrder` and `InternalOrderType`.

The respective `interfaces` are implemented by the classes `AcceptanceType` and `Acceptance`, `DeliveryType` and `Delivery`, `EmissionType` and `Emission`, `ReceptionType` and `Reception`. All these classes are extensions of the classes `InternalOrderType` and `InternalOrder`, respectively. For example, `Delivery` extends `InternalOrder` by a method that allows to specify a `ReadAPI` object, and overwrites the method `execute()` to assure that the data is delivered to the read-API after calling all the workers.

3.3.3 Mapping Classes to Packages

Java allows to group related classes in so called `packages`. We give a short overview of the packages that make up PITOU and which kind of classes these packages contain.

- The package `pitou.core.structure` defines all `interfaces` and abstract classes needed to implement protocol components (so called hot spots) and to implement the core classes like environment or order.

- The package `pitou.core` contains all core classes that implement the framework (like environment and orders) and that are needed to run protocol software.

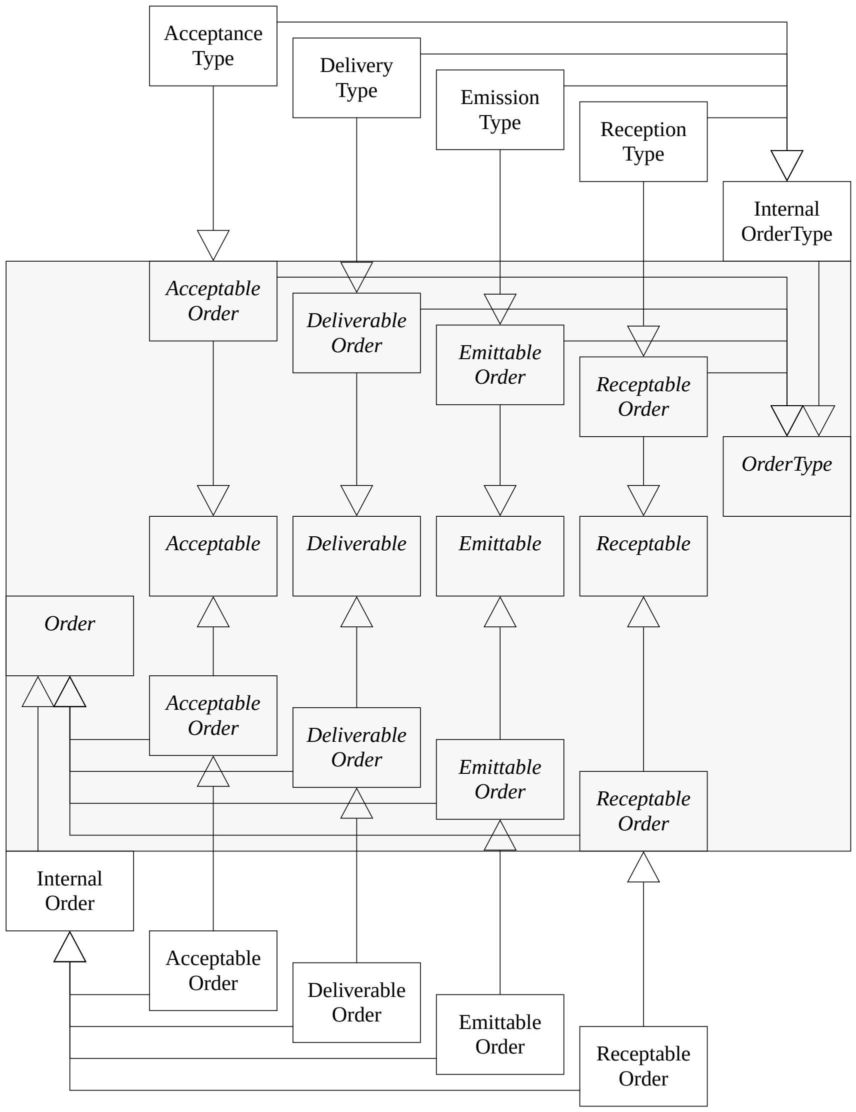


Figure 3.3: Classes for Data-Path Classification

- The package `pitou.core.mapping` defines all `interfaces` and classes that concern the mapping strategy, e.g. a standard mapping strategy that uses order identifiers to perform mapping.

- The package `pitou.core.register` defines all classes used to register the various protocol objects. The concept behind these registration classes will be explained later.

- The package `pitou.net` contains all wrapper classes for network access (e.g. TCP, UDP, IP-Multicast wrappers)

- The package `pitou.util` contains utility classes, e.g. byte array manipulation methods.
- The package `use.pitou.workers` contains implementations of workers (part of the component library).
- The package `use.pitou.entries` contains all classes implementing entries and entry types (part of the component library).

Figure 3.4 shows the different categories of classes and how they contribute to build and run protocol software within the PITOU framework.

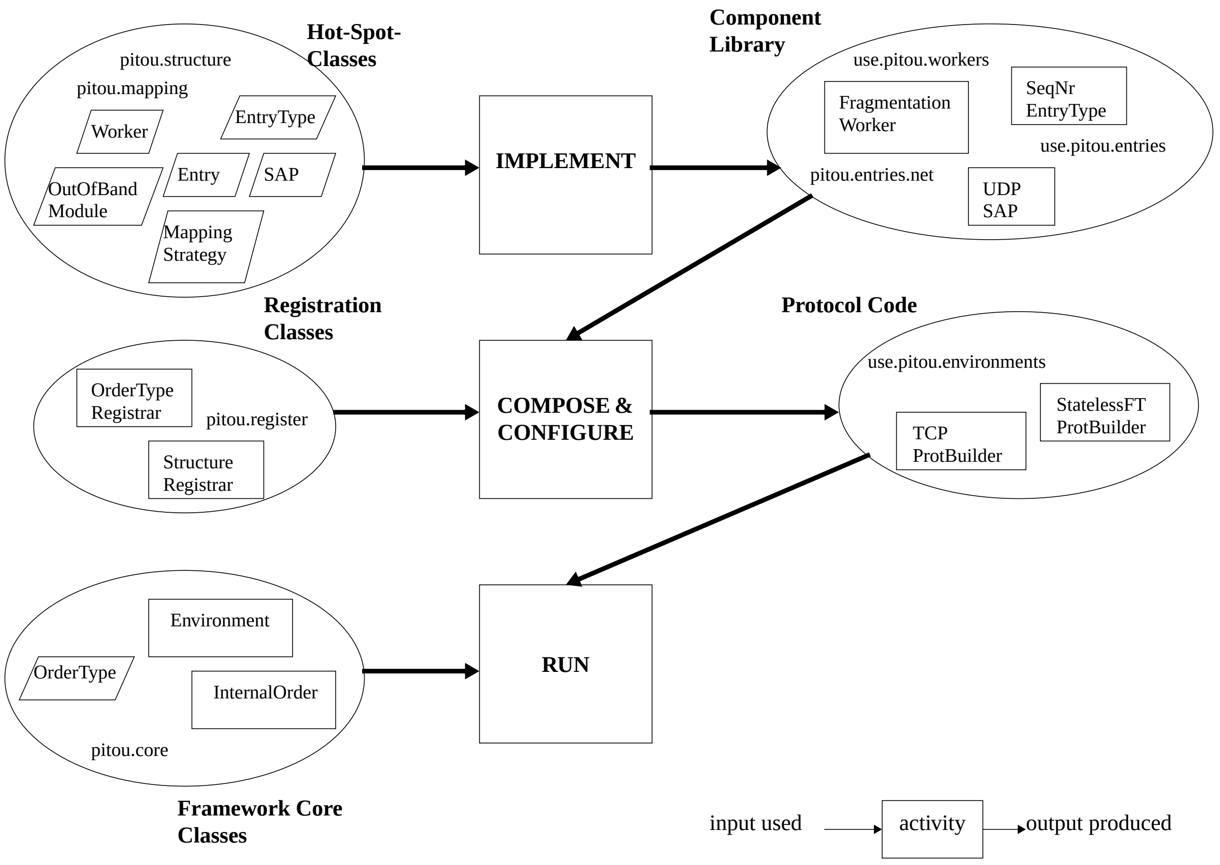


Figure 3.4: The PITOU classes and their Use

3.4 Implementation of Reusable Components

The very first step to build a new protocol is the implementation of components with the behavior required. This step is of course not needed when respective classes with the required services already exist. In fact, the long term goal of using this framework is being able to no longer write code, but reuse, configure, and assemble existing components. In this section, we describe all abstract classes and `interfaces` that may be implemented when to add new reusable components to the framework library.

3.4.1 Implementing Workers

In order to implement a new worker, the abstract class `Worker` must be extended, i.e. the following abstract methods need to be implemented.

- The core protocol function of a worker is implemented in the method `boolean call(Entry[])`. This method is called every time when an order is executed, and provides a worker with the entry objects containing the respective header information of the actual message. Each worker can decide to immediately stop the execution of an order by returning `false` (e.g. when a worker detects a wrong checksum); however, the possibility of determining order-execution should be used with attention and carefully documented in the specification of a worker.
- A worker is initialized when its method `void init(Entry[])` is called. During initialization of a worker, a timer may be started or an initial value for internal variables may be set.
- The method `deactivate()` is called when the protocol session has definitively determined, and gives the worker the possibility to free allocated resources (stop timers, close files, etc.)

3.4.2 Implementing Entries

Implementing entries requires the extension of two abstract classes: `EntryType` defines the static properties of an entry and is used for building the structure of a protocol. `Entry` defines the dynamics of an entry and is used internally in the framework. Entry-types are components, while entries are simple objects.

The abstract class `EntryType` defines only one abstract method.

- The method `Entry create()` returns an entry object of the type related with the entry-type object that creates it (e.g. an object of type `SeqNrEntryType` creates an object of type `SeqNrEntry`). During the creation, the static information defined by the entry-type object should be given to the created entry object (e.g. the initializable flag, the visible flag, the byte-length, etc.).

EntryType defines a number of other methods, which are already implemented, and hence inherited by every new entry-type and entry sub-class.

- The methods `void setInitFlag(boolean)` and `boolean getInitFlag()` allow to configure entry-types *initializable*. An initializable entry object obtains its respective header by parsing an initial byte-array before order execution.
- The methods `void setVisibleFlag(boolean)` and `boolean getVisibleFlag()` allow to configure entry-types *visible*. A visible entry object transforms the header information it represents into an byte-array after order execution.
- The methods `void setSize()` and `int getSize()` allow to configure the number of bytes the corresponding header field represents.

The abstract class `Entry` defines the following abstract methods:

- The method `void fill(byte[],int)` provides an entry with a byte-array and a start point within this array, and so allows an entry to extract its information and represent it as an internal value. This method is called only for entries configured *initializable*.
- The method `void fill()` initializes an entry with a standard value and is called for non-initializable entries every time a new order is initialized.
- The method `byte[] getBytes()` returns a byte-array representing the header field encapsulated by an entry object. If the header field representation is not yet serialized, serialization will be done in this method.
- The method `void setValue(Object)` allows to assign a value to an entry object. The entry itself is responsible to check if the value has the correct type he expects.
- The method `Object getValue()` returns the value of an entry. The worker that uses the value must know its concrete type (e.g. Integer).

In addition to ”simple” entry-types, PITOU also supports **array entry-type**s (similar to arrays in almost all programming languages) and **composite entry-types** (similar to the `struct` concept in C).

Array entry-types are implemented as classes named `ArrayEntryType` and `ArrayEntry` that extend `Entry` and `EntryType`, respectively. Each array-entry type allows to specify an entry-type, which the array will consist of, and a size property indicating the number of elements, which may be changed dynamically from order to order. Each array-entry object will manage a set of entry-objects (depending on the size of the array-entry-type), to which he delegates parsing and serialization.

In order to allow the grouping of different entry-type objects, we define the abstract classes `CompositeEntryType` and `CompositeEntry`, which extend `EntryType` and `Entry`, respectively. Each composite-entry-type allows to register list of entry-types that form the composite-entry-type. The classes `BlockEntryType` and `BlockEntry` are implementations of `CompositeEntryType` and `CompositeEntry`, respectively. A `BlockEntry` delegates parsing and serialization of data to its sub-components. Figure 3.5 depicts the class diagram for composite entry-types.

The collaboration diagram in Figure 3.6 shows how composite entries work. The example is similar to the one in Figure 2.15: an order $O1$ specifies two entries $E1$ and $E2$, whereas $E1$ is an composite entry. $E1$ does not represent a value itself but consists of three entries (refered to as sub-entries $SE1, SE2$ and $SE3$) that represent a value, respectively. $E3$ delegates parsing to its sub-entries.

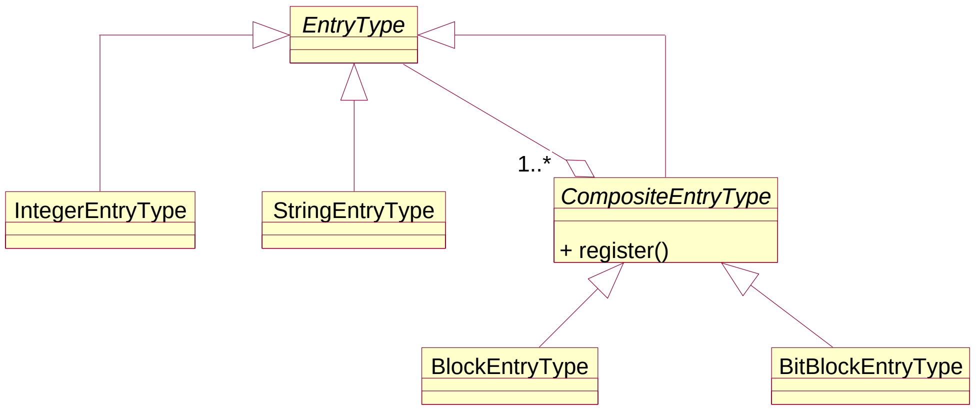


Figure 3.5: Composite Entry-Types

Another composite-entry-type is implemented by `BitBlockEntryType`/ `BitBlockEntry`. Bit-blocks allow to register entry-type objects together with a bit size. During parsing and serialization, the `BitBlockEntry` object assures that values are mapped to bits and not to bytes.

Array-entry-types and composite-array-types can be combined, i.e. we can build arrays of blocks or blocks containing arrays. Blocks can also consist of blocks, and arrays may be arrays of arrays. That way, we provide the typing power and flexibility of most modern programming languages.

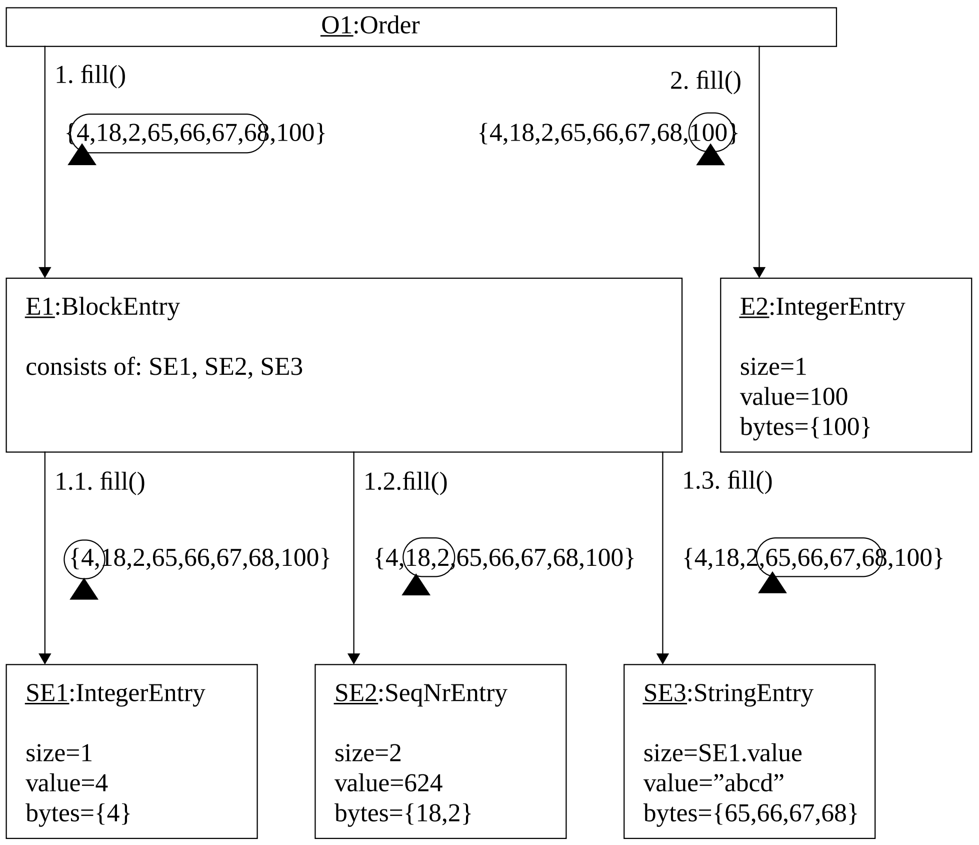


Figure 3.6: Dynamics of Composite Entry

3.4.3 Out-of-Band Modules

There may be reusable functions for protocol implementation that are not part of a data-path, i.e. which do not operate directly on message header fields, but nevertheless provide a general service useful for worker objects. A good example for an out-of-band module are TCP's congestion-avoidance and control algorithms [72].

PITOU provides the `interface OutOfBandModule` for these kind of modules. The `interface` does not define any methods, it just gives a type to out-of-band modules and thus helps to distinguish it from other protocol components.

3.4.4 Network Interface

PITOU provides a couple of abstract classes and `interfaces` to implement wrapper classes for network service access. The most important classes are `SAP` and `Address`. Their goal is to hide protocol and system specifics behind a stable and simple interface. PITOU already provides implementations of `SAP` for TCP, UDP, and IP-Multicast.

The abstract class `Address` encapsulates a network address. It defines no abstract methods to be implemented. We implemented the class `InternetAddress`, which extends `Address` and encapsulates a 4-byte IP address and a 2-byte port number.

The class `Packet` encapsulates a piece of data – a byte-array – and the source address (of type `Address`) of the sender of this data.

The class `SAP` provides access to a communication service access point by encapsulating a resource-handle (like a socket endpoint). SAP classes provide the following methods:

- The methods `void open()` and `void close()` are used to open and release system resources (e.g. a socket).
- The methods `void setPacketSize(int)` and `int getPacketSize()` allow to configure the maximum packet size in bytes that can be handled. All data exceeding this size is cut.
- The method `Packet read()` performs a non-blocking read-operation to the resource-handle. If data has arrived, the data is returned together with the source address as an object of type `Packet`. If no data has arrived, the method simply returns `null`.
- The method `Packet readUntil()` performs a blocking read-operation to the resource handle, i.e. the method blocks until data arrives or it is explicitly interrupted.
- The method `void send(Packet)` performs a send-operation to the resource-handle
- Each SAP object has a default-destination address, which is used when no address is specified for the send-method. Default-destination addresses are configured via `Address getDefaultDestination()` and `void setDefaultDestination(Address)`.
- Each SAP object allows to specify its own address via `Address getOwnAddress(Address)` and `void setOwnAddress()`.

3.4.5 Mapping Strategy

A mapping strategy is an object that maps a piece of data to a reception order-type. Mapping strategies are represented by the abstract class `MappingStrategy`, and strongly cooperate with an environment object.

The following abstract methods must be implemented for mapping strategies:

- The method `void map(Packet)` uses the byte-array of the packet to determine matching reception-type object(s). It then calls the `receive(Packet,ReceptionType)` method of the associated environment object.

- The method `void outgoing(EmissionType)` is called by the environment to allow mapping strategies to append mapping information to outgoing data, which is needed by the mapping strategy on the remote side to identify the matching reception-type object.

Besides defining these abstract methods, mapping strategy classes implement methods to assign an environment object and a list of reception-type objects.

PITOU already provides implementations of various mapping strategies. The `DefaultMappingStrategy` class uses the identifier of each reception-type and emission-type to perform mapping. The method `outgoing()` obtains the identifier via `Order.getID()` and appends it to the byte-array. The method `map()` looks into the byte-array to obtain the identifier, strips the information from the byte-array, and identifies the right reception-type object.

Other mapping strategies provided are:

- `BroadcastMappingStrategy`: all reception-type objects are initialized with the incoming byte-array

- `ByteTupleBasedMappingStrategy`: each reception-type is registered together with a set of tuples each specifying a position in the byte-array and a required value. An incoming byte-array is checked for all tuples using a multi-hashing strategy to determine the correct reception-type.

- `CallMappingStrategy`: each reception-type is registered together with an object of type `Callable`, which implements a method `boolean isForMe()`. For each incoming byte-array, all callable objects are asked if they want the byte-array. The related reception-type object of the first callable object returning a `true` is the one who gets the data.

- `MetaMappingStrategy`: consists of a list of different mapping strategy objects. During `map()` all mapping strategies are tried until one matched.

Due to pre-defined message formats, the implementation of existing protocols (e.g. RTP) may pose constraints to the developer that require the implementation of protocol-specific mapping strategies. In all other cases, it is useful to fall back on the `DefaultMappingStrategy` or one of the other provided mapping strategies.

3.4.6 Interfaces to Application

Since the way applications communicate via a protocol object may also be encapsulated in reusable objects, we encourage the implementation of separate classes to encapsulate interaction between application and environments, instead of letting the application part communicate directly with the environment.

PITOU provides two `interfaces` that can be implemented to adapt interaction to the applications need. The first `interface` is called `ReadAPI` and allows the application to obtain data from an environment object. The second `interface` is called `WriteAPI` and allows the application to feed the protocol environment with information.

`ReadAPI` defines the following abstract method:

- The method `void deliver(Object[], DeliveryType)` must be implemented by an application that wants to be notified about the delivery of data. Data is represented as an array of objects, i.e. in a representation directly understandable for the application (in contrary to e.g. a byte-array delivered by a TCP socket). The argument of type `DeliveryType` allows to distinguish deliveries from different orders.

PITOU comes along with a class `QueuingReadAPI`, which implements the `ReadAPI` `interface` by just appending delivered data to a queue, where it can be read by the application later. The use of the class `QueueingReadAPI` can simplify the implementation of applications, since the application does not need to implement `ReadAPI` itself. On the other hand, that way a context switch is introduced between processing data in the protocol part and using it by the application part.

`WriteAPI` defines the following abstract methods:

- The methods `void setAcceptanceType(AcceptanceType)` and `void setEnvironment(Environment)` need to be implemented to give a write-API access to environment, and to specify which acceptance-type is concerned.

- The method `void accept(Object[])` is meant to feed the environment with new data. The specification of a destination address is necessary when the underlying SAP object is not connection-oriented or has no default destination address. Destination address can be specified by using the method `void accept(Object[],Address)`.

Again, PITOU provides an implementation of `WriteAPI`: the class `StandardWriteAPI` simply calls the `accept()` method of the related environment.

3.4.7 Using Timers

Each environment object has a reference to an object of type `TimerPool`. The idea of a timerpool is to centralize all timer handling within an environment object to reduce the number of objects and threads used. A timer-pool may also be shared by several environment objects.

Each class that wants to use timers (in our context, those are normally workers or out-of-band modules) must implement the interface `TimerCompatible`, which declares the following methods:

- The method `void setTimerPool(TimerPool)` makes the global timer-pool object accessible for each object of type `TimerCompatible`. This method is called during the initialization of the environment object.
- The method `void timerCall(Timer t)` allows the timer-pool to notify a `TimerCompatible` object about the expiry of a timer.

The class `TimerPool` provides the method `Timer getNewTimer(TimerCompatible)` to return an object of type `Timer` to an object of type `TimerCompatible`.

The class `Timer` provides a couple of methods to start, reset, and change time-outs. A timer object can be in three states: the state *unused* means that the timer has not been given to any object. The state *used* means that a timer has been given to an object that has called `getNewTimer()`. The state *activated* means that a timer has started to run. The following methods are implemented for timers.

- The method `reset(long)` resets an activated timer with a new value or activates a timer that has not yet been activated.
- The method `cancel()` deactivates a running timer.
- The method `getAbsTimeout()` returns the absolute time of expiry.
- The method `endUse()` assures that the timer is no longer usable until it is returned again by the timer-pool's `getNewTimer()` method.

A timerpool works as follows. A worker, who implements the `TimerCompatible` interface, obtains a timer object from the timer-pool's list of unused timers by calling `getNewTimer()`. For each timer object, the timer-pool reserves an own thread. The timerpool adds the worker to a table that associates each timer-compatible worker with its timer object. Each time the worker operates on the timer object, the timer-pool is notified. The timerpool maintains a list of all activated timers sorted according to there expiration time. The timerpool implements a thread that sleeps until the first timer in the list is canceled or expires. Upon expiry, the timerpool activates the thread of the respective timer, and calls the related timer-compatible object via the method `timerCall()`, which handles the time-out.

3.5 Assembling Components using Java-Beans

Once all necessary components are implemented, a protocol can be built by just assembling existing components together and configuring them. In this section we show how the Java-Beans component model can be used to build PITOU protocols in a visual manner, and discuss benefits and drawbacks.

3.5.1 Naming Conventions and Composition Process

Now that we know the main components of our framework, we want to show how a tool like Visual-Age for Java (VAJ) [70] can be used to build protocols. VAJ provides a user interface called **visual composition window** (VCWin), which is used to build an application (or another component) visually. The VAJ allows to place bean objects in the VCWin, to configure them, and to connect them.

In order to be Java-Beans compliant, a component implementation must respect certain naming conventions:

1. all classes to be interpreted as beans must provide a standard constructor without parameters
2. configurable attributes (”properties”) are identified via methods that start with `set` or `get`
3. communication between beans is modeled via events; beans that fire an event of type `ABC` must implement two methods named `addABCListener()` and `removeABCListener()`

The Java-Beans compliant visual builder tools use the Java reflection mechanism to identify properties, events, and behavior. The reflection classes allow to reason about the internal structure (class, method, and field declarations) of programs at runtime.

While a parameter-less standard constructor and the implementation of `set/get` methods are common programming practice rather than a constraint on flexibility, the event-model forces us to implement methods that are very specific to the Java Beans component model. Changing the component-model may thus require changing many classes.

The composition process with VAJ looks like follows.

1. VAJ provides a palette of icons, each of which represents a Java-Bean (see Figure 3.7. The programmer selects a bean and puts it on the VCWin. Internally, VAJ instantiates an object of this bean class by using the standard constructor.

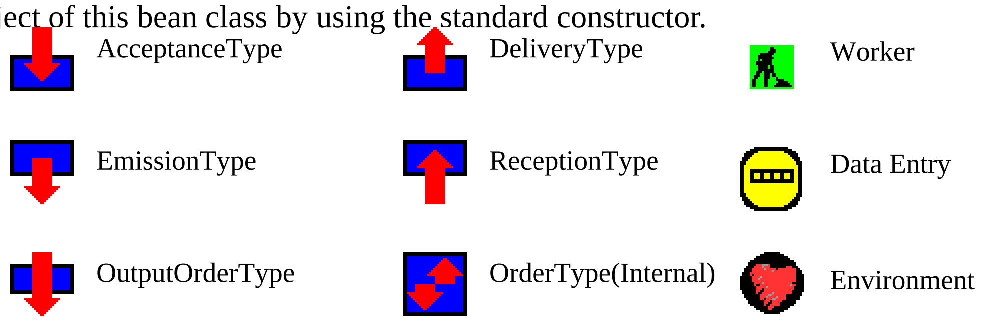
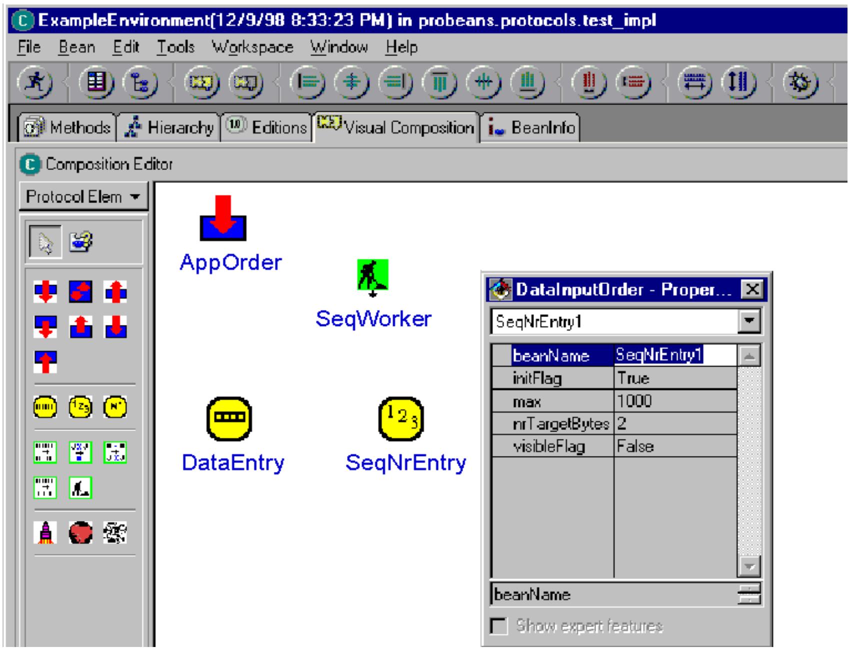


Figure 3.7: Design Elements as Java-Beans Icons

2. For each bean instantiated, VAJ allows to open a window called **property editor** (see Figure 3.8). This little window provides the possibility to configure all properties of the bean. Internally, VAJ uses the `get` method of the bean object to obtain the current value, and the `set` method to write a user-defined value to the object. It is important to note that all beans are instantiated, i.e. configuration happens at runtime on real objects.


ExampleEnvironment(12/9/98 8:33:23 PM) in probeans.protocols.test_impl
File Bean Edit Tools Workspace Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Composition Editor
Protocol Elem
AppOrder
SeqWorker
DataEntry
SeqNrEntry
DataInputOrder - Proper...
SeqNrEntry1
Show expert features

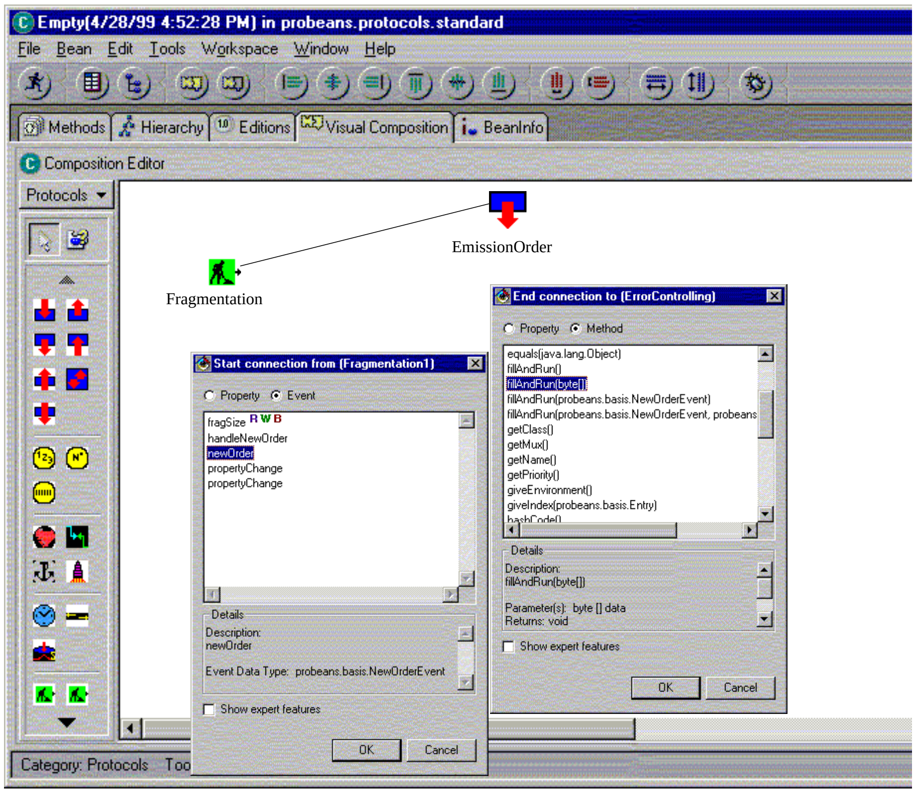
beanName	SeqNrEntry1
initFlag	True
max	1000
nrTargetBytes	2
visibleFlag	False

beanName

Figure 3.8: Property configuration (screen-shot using Visual Age)

3. For each bean instantiated, VAJ allows to open a window that identifies all events the bean fires, and a window that allows to identify all accessible methods a bean provides. Connecting two beans means choosing an event of one bean (the event source) and specifying

the method of another bean (the event destination) that is supposed to be called when the event source bean fires. Since it can not be expected that an event destination implements an event handler for any possible event fired by any other bean, the two beans can not be directly connected. Instead, VAJ internally represents the connection between two beans. When composition has terminated, VAJ produces code (so called event-adapters described in Chapter 2) that maps fired events to the specified method. Connecting beans via events is depicted by Figure 3.9.


Empty(4/28/99 4:52:28 PM) in probeans.protocols.standard
File Bean Edit Tools Workspace Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Composition Editor
Protocols
EmissionOrder
Fragmentation
Category: Protocols Too
OK Cancel
OK Cancel

Start connection from (Fragmentation1)

Property [✓ Event]

fragSize R W B
handleNewOrder
newOrder
propertyChange
propertyChange

Details
Description:
newOrder

Event Data Type: probeans.basis.NewOrderEvent

☐ Show expert features

End connection to (ErrorControlling)

Property [✓ Method]

equals(java.lang.Object)
fillAndRun()
fillAndRun(byte[])
fillAndRun(probeans.basis.NewOrderEvent)
fillAndRun(probeans.basis.NewOrderEvent, probeans
getClass()
getMux()
getName()
getPriority()
giveEnvironment()
giveIndex(probeans.basis.Entry)
hashCode()

Details
Description:
fillAndRun(byte[])

Parameter(s): byte [] data
Returns: void

☐ Show expert features

Figure 3.9: Building event connections (screen-shot using Visual Age)

4. When the composition process is terminated, VAJ generates the code of the application composed visually. That is, VAJ represents the chosen values of the properties and the specified connections between beans as Java code. The generated code may be used itself as a bean.

VAJ provides one default event called *init()*-event, which is in fact a pseudo-event supposed to be fired once when the application composed is instantiated. The init-event allows to realize static connection between beans, e.g. giving one bean the reference of another bean. For composing PITOU protocols, the init-event is an indispensable mechanism, since the biggest part of the composition process consists of specifying static relations between components.

3.5.2 Composition of PITOU protocols

Using VAJ to compose a complete protocol within the PITOU framework basically consists of the following steps.

1. Instantiate all beans needed, at least
   - one `Environment` bean,
   - all `OrderType` beans,
   - for each order-type the respective `EntryType` and `Worker` beans
2. Configure the beans, that is
   - the priority for order-types,
   - the order-IDs for emission-types and reception-types,
   - the init-flag and the visible-flag for entry-types,
   - worker specific properties (e.g. window size, time-out values, etc.).
3. Define the structure. The init-event is used to
   - connect the environment with its order-types,
   - connect the order-types with its entry-types and workers.
4. Define the communication during execution time among workers. That is, worker specific events are used to
   - establish communication among workers to exchange information or for notification,
   - establish communication between a worker and an order-type to create new orders during execution.

Figure 3.10 depicts a screen-shot from VAJ to give the reader an idea how a protocol looks like that has been composed visually. The protocol depicted implements an ARQ-based error control mechanism without assuring ordered delivery, a window-based flow control mechanism, and fragmentation. Although the example protocol is rather simple, its visual representation is already rather complex and difficult to grasp.

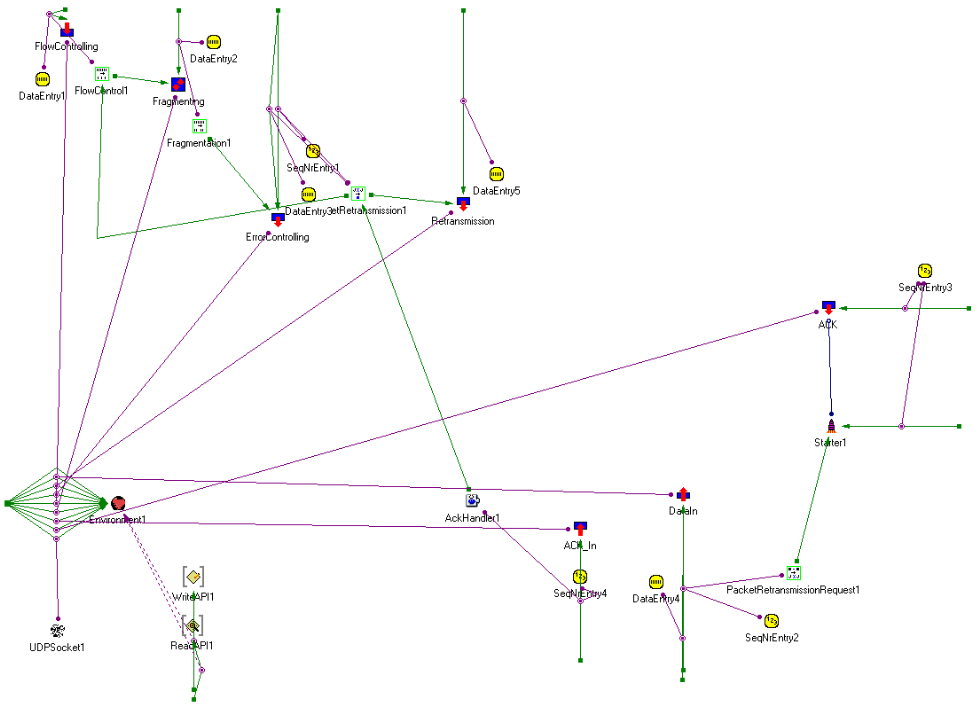


Figure 3.10: An example protocol (screen-shot using Visual Age)

3.5.3 Discussion

Our experiments using VAJ and Java-Beans to compose protocols lead us to a number of observations. Among the benefits of this approach are:

- **rapid-prototyping**: the time to compose new protocols is extremely reduced. Instead writing code for configuration and structure specification, we can use a comfortable graphical user interface.
- **robustness**: since we are guided through the composition process, the probability of e.g. overseeing an important property, is reduced. Additionally, it is much easier to capture a structure and detect flaws when the structure is presented visually – however, depending on the complexity of the structure.
- **ubiquity**: since the approach is based on the Java-Beans model, which has become a quasi standard for component-based software engineering, PITOU protocols can be built by any other tool supporting Java-Beans.

However, we also discovered a number of limits:

- **complexity**: experiments with implementing TCP visually (see also Chapter 5) showed that a protocol may comprise many connections among its components. Composing a complex protocol like TCP leads to a visual complexity that is no longer manageable for human beings. Especially, later modifications are very cumbersome.

- **builder-tool**: not only the human eye, but also VAJ had problems with complex protocols. When a large number of connections are specified, VAJ sometimes changed the order of the connections during the code-generation phase. Since the right order of registration of workers and entry-types is crucial, the misordering of connections can introduce serious flaws into a protocol.
- **code generation**: another problem with this approach is that we have no control over code that has been generated for the event adapters. Sometimes (for debugging, simulation, or logging) it is useful to modify the event-adapter code that implements the communication between components. However, each time the code is modified, these modifications are overridden.

While the use of Java-Beans was an interesting experience, we consider the approach – at least given the todays tools – not yet mature enough. If VAJ would provide different views onto different composition levels, a better support to allow step-wise definition of sub-components, and a better handling of complex interaction between components, the use of Java-Beans would be much more attractive. Until this happens, we help ourselves with a proprietary design for component interactions, which is tailored to the needs of protocols.

3.6 Assembling Components – Our Own Approach

We propose two concepts to overcome the problems with the Java-Beans approach. We first propose to encapsulate the structure of a protocol in special objects called registrars to decouple structural and dynamic behavior of a protocol. Second, we propose to encapsulate interaction between workers (or any other component) into explicit objects to allow the specification of the interaction at runtime without a-priori component knowledge.

3.6.1 The concept of Registrars

In order to decouple the static (definition of the protocol structure) from the dynamic aspects (protocol processing at runtime) of a protocol implementation, we encapsulate the static aspects in separate components called **registrar**s that are used for the registration of the various protocol components and their relations. Instead of registering e.g. order-types directly with the environment object, we register them with an object of class `OrderRegistrar` and give this registrar object to the environment. The same concept applies also for the registration of workers and entries, mapping strategies, or SAPs. The advantage of using registrar objects is that the structure of a protocol is explicitly defined, accessible in a compact manner, and thus usable for reasons of modification or validation. Registration using registrar objects works as follows.

Registration of Order-Types

The class `OrderRegistrar` is used to register all order-type objects for an environment. Order-registrars provide the method `void addOrder(OrderType)` to register order-type objects. Internally, all order-types are sorted according to their class (acceptance, delivery, etc.).

Once all order-types are registered, the order-registrar object can be given to the environment via `Environment.setOrderRegistrar()`.

Registration of Workers and Entries

The class `StructureRegistrar` is used to register all workers, entry-types, and their parameter relations for a certain order-type. The class `StructureRegistrar` provides the following methods:

- The method `void addEntry(EntryType)` registers an entry-type object.
- The method `void addWorker(Worker)` registers a worker.
- The method `void addWorkerParameter(EntryType, Worker)` specifies a parameter relation between a worker and an entry-type object. Entry-type or worker object are registered via the other two methods, if not yet done.

Note that the order of registration for workers define the order in which the workers are called. The order registration for the entry-type objects define the format of the messages sent. It is therefore very important to pay attention to the right order of registration.

Once all workers and entry-types are registered, the structure-registrar object can be given to its order-type via `OrderType.setStructureRegistrar()`.

Registration of Out-Of-Band Modules

The class `OutOfBandRegistrar` is used to register all out-of-band modules for an environment. Out-of-band-registrars provide the method `addOutOfBandModule()`. Once all out-of-band modules are registered, the out-of-band registrar object can be given to the environment via `Environment.setOutOfBandRegistrar()`.

3.6.2 A Design Approach for Component Interaction

Java-Beans uses an event-mechanism and requires certain naming conventions to let arbitrary components interact without compromising their independence. Interactions are thus not an explicit part of the protocol structure and are generated at implementation time. The goal of our design approach is to encapsulate interactions in explicit objects that can be established at runtime.

During our experiments with composing protocols using Java-Beans, we discovered that interactions between our components can be classified into two categories:

- **shared-resource relations**: the characteristic of shared-resource relations is that a certain resource (e.g. a buffer) is provided (i.e. instantiated) by one component and may be used by one or more other components. We call the component that provides the resource a **resource-publisher**, and the components using it a **resource-subscriber**. Resource-publishers must provide an interface to make the resource accessible, resource-subscribers must provide an interface to obtain the resource.

- **notification relations**: the characteristic of notification-relations is that a certain component notifies one or more other components about a certain event (but without specifying an event type or passing parameters). We call the notifying component a **notifier** and the notified component a **notifiee**. Notifiers and notifiees must provide an interface that allows to connect them: notifiers define an interface that indicates notification sources, notifiees an interface that indicates a notification sink.

Any connection between Java-Beans – which are nothing more than parameterized method calls – can be composed out of these two relations. Method calls are mapped to notification relations, parameters to shared resource relations.

Figure 3.11 depicts the class-diagram for notification relations, and Figure 3.12 the class-diagram for shared resource relations.

Since the creation of orders is fundamental to implement protocols using PITOU, we decided to explicitly support a third category of relation called **new-order-relations**. New-order-relations describe the case that one component (normally a worker) fires an event that should lead to the creation/execution of a new order. New-order relations are thus strongly related to notification relations.

Interaction between Workers using Explicit Relation Objects

The idea of our approach is to encapsulate interaction in explicit objects. We therefore define an `interface Relation`, which declares the method `void init()`. This inter-

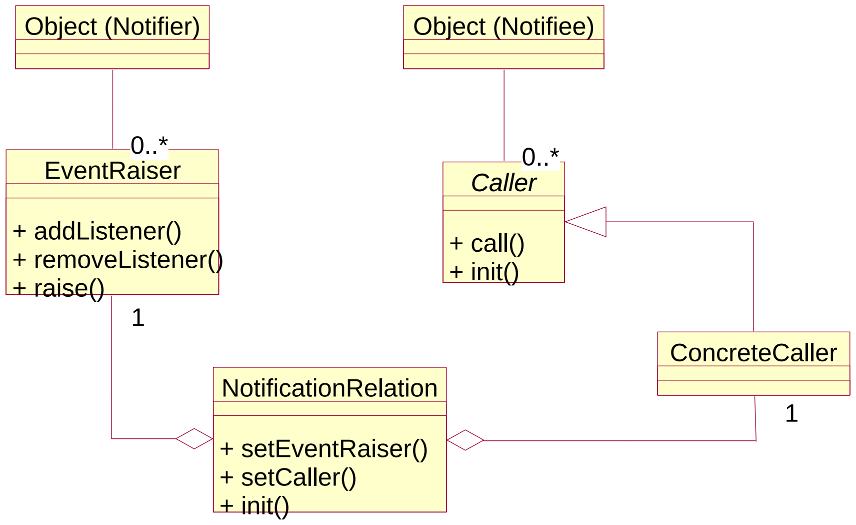


Figure 3.11: Notification Relation (Class Diagram)

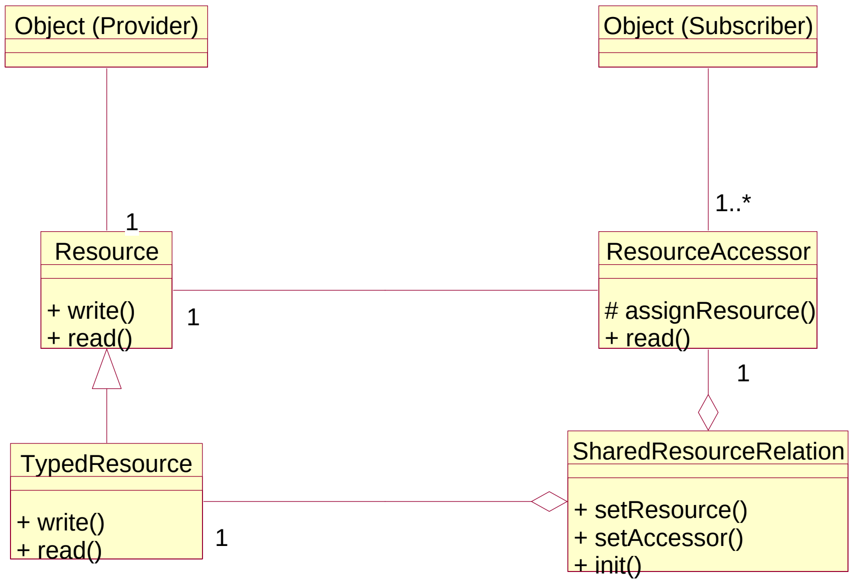


Figure 3.12: Shared Resource Relation (Class Diagram)

face is implemented for each category of relation, hence `SharedResourceRelation`, `NotificationRelation`, and `NewOrderRelation`.

Every component that wants to be identified as a resource publisher, must provide a method that allows to obtain an object of type `Resource`. Every component that wants to be identified as a resource subscriber, must provide a method that allows to obtain an object of type `ResourceAccessor`. Establishing a shared-resource relation then means to assign the obtained `Resource` and `ResourceAccessor` objects in a `SharedResourceRelation` object. Note that the resource-publisher and -subscriber components do not need to implement an interface or be of a certain type, they just provide a kind of place-holder object (depending on their role in the relation) that allows to interact without making any assumption (besides the type of the resource that is exchanged) about the interaction partner component.

The following scenario illustrates how shared-resource relations are established and used.

1. The components A and B are given. Component A is a resource-publisher and provides the method `Resource defineOutput_seq()`. Component B is a resource-subscriber and provides the method `ResourceAccessor defineInput_nextSeq()`.

2. The protocol developer instantiate an object of type `SharedResourceRelation`.

3. He assigns the resource object obtained from A to the relation via `setResource()` and the resource-accessor obtained from B via `setAccessor()`

4. The method `SharedResourceRelation.init()` gives the resource to the accessor.

5. The subscriber component B can now access the resource object by calling the accessor's `Object read()` method.

For notification relations, each notifier components provides a method that returns an object of type `EventRaiser`. Each notifiee component provides a method that returns an object of type `Caller`. Establishing a notification relation thus means to assign the obtained `EventRaiser` object with the obtained `Caller` object. While `EventRaiser` is a reusable class (as `Resource` and `ResourceAccessor`), `Caller` is only an `interface` declaring the method `void redirectNotification()` that must be implemented for each kind of notification a notifiee is waiting for. Within `redirectNotification()` a caller should call the component specific method that reacts to the notification.

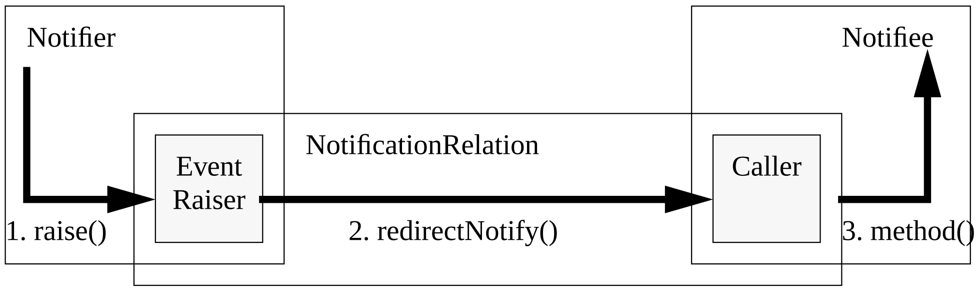
The following scenario illustrates how notification relations are established and used.

1. The components A and B are given. Component A is a notifier and provides the method `EventRaiser defineEvent_timeout()`. Component B is a notifiee and provides a method `void lossHappened()`, which should be called every time packet is detected. B therefore provides the method `Caller defineCaller_loss()`, which returns an object of the class `Caller_loss`, which implements the method `redirectNotification()`. Within `redirectNotification` a simple call to `B.lossHappened()` is implemented.

2. The protocol developer instantiates an object of type `NotificationRelation`.

3. He assigns the event-raiser object from A to the notification relation object via `setEventRaiser()` and the caller object via `setCaller()`.
4. The method `NotificationRelation.init()` registers the caller at the event-raiser.
5. Each time component A calls the method `EventRaiser.raise()`, all registered caller objects are notified via `Caller.redirectNotification()`

Figure 3.13 illustrates the method sequence to execute a notification call.



A Notification Call:
1. the notifying component calls the event-raiser to indicate an event via raise()
2. the event-raiser calls all registered listeners via redirectNotify()
3. the caller invokes the notified component's method that handles the notification

Figure 3.13: Notification Relation (Method Call)

New-order relations are composed by assigning an event-raiser object with an order-type object. That is, any event-raiser provided by notifier components can be used either to notify another component or to create a new order.

We apply the following naming conventions to objects that want to participate in one of the three relations described above. Any component that wants to notify another component about a certain event must implement a method starting with `defineEvent` that returns an object of type `EventRaiser`. A component triggering more than one event must implement the corresponding number of `defineEvent` methods. Any component that wants to be notified about a certain event must implement for each event a method starting with `defineCaller` that return an object of type `Caller`.

For each resource that a resource-publisher component provides, it must implement a method starting with `defineOutput` that returns an object of type `Resource`. For each resource that a resource-subscriber component needs to perform his tasks, it must implement a method starting with `defineInput` that returns an object of type `ResourceAccessor`.

Table 3.1 lists the PITOU naming conventions for explicit interaction between components.

Role	Relation Type	Method Name	Return Type
Notifier	Notification	`defineEvent_...()`	`EventRaiser`
Notifiee	Notification	`defineCaller_...()`	`Caller`
Resource Publisher	Shared Resource	`defineOutput_...()`	`Resource`
Resource Subscriber	Shared Resource	`defineInput_...()`	`ResourceAccessor`

Table 3.1: Naming Conventions for Component Interaction

Some core PITOU classes are by default notifiers or notifiees, shared resource publishers or subscribers.

- All classes of type `OrderType` provide a method `defineEvent_execDone()` to provide an event-raiser that is called every time order execution has terminated.
- The class `Environment` provides a method `defineCaller_terminate()` to be notified about the fact that a protocol session is considered to be terminated.
- Emittable and receptable order-types provide address and order-ID as shared resources defined by `defineOutput_add()`, and `defineOutput_id()`.

Figure 3.14 illustrates the difference between our approach and the Java-Beans approach that uses event adapters. While a Java-Beans tool must create a new adapter class for each relation, we force every component to provide its interface as an object that can be used to be connected with the interface object of another component via a standard object (of class `Relation`).

Relation Registrars

Similar to the registrars already described above, we introduce a class `RelationRegistrar`, which has the objective of storing all relation objects of a protocol environment. Relation registrars provide the method `void addRelation(Relation)` to register relations and to sort them internally into the three categories (notify, shared-resource, new-order). The method `void init()` initializes all relation objects by calling `Relation.init()`.

Once all relation objects are registered, the relation-registrar object can be given to the environment via `Environment.setRelationRegistrar()`.

3.6.3 Discussion

One might say that encapsulating interaction in objects heavily complicates communication between components. New helper classes (`Caller`) must be implemented, the memory footprint of the protocol software is larger, and the overhead for notify relations is increased by

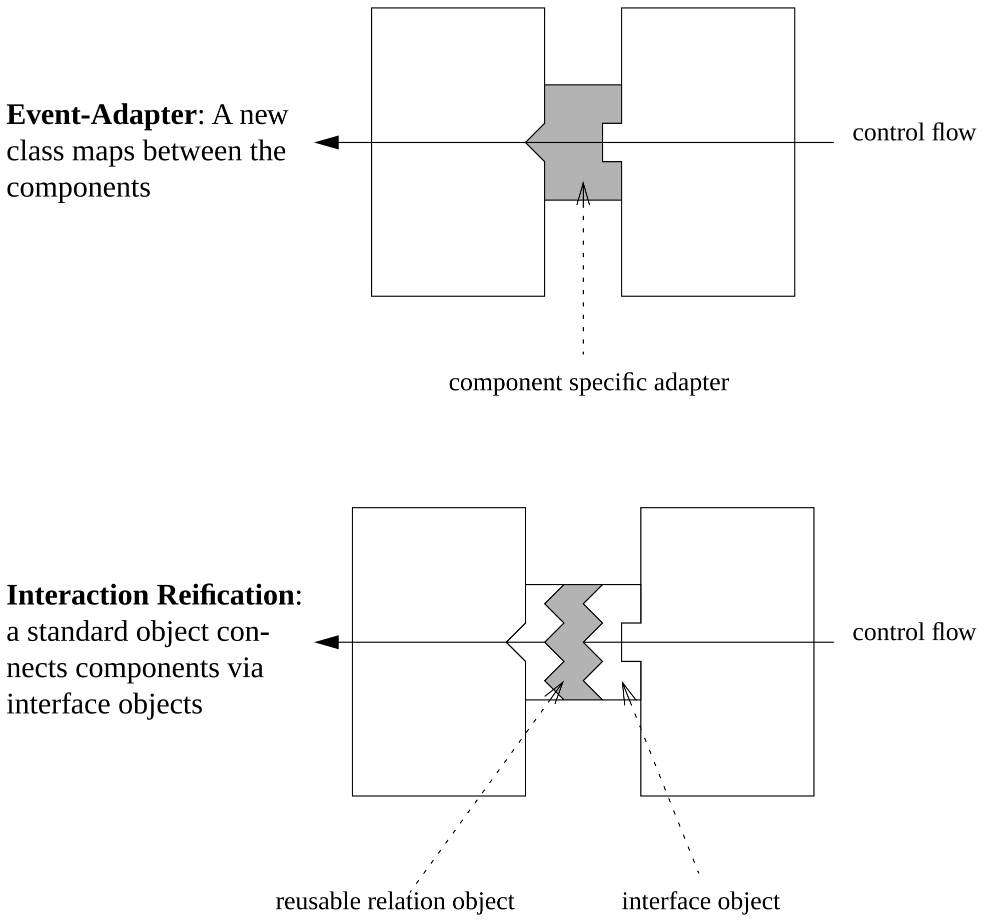

Event-Adapter: A new class maps between the components
control flow
component specific adapter
Interaction Reification: a standard object connects components via interface objects
control flow
reusable relation object
interface object

Figure 3.14: Interaction Reification vs. Event Adapter

an additional method call indirection. However, we have a number of reasons to believe that encapsulating interaction in objects is very useful.

First of all, we ensure complete independence of components while providing full flexibility of interaction. Components can interact without knowing anything about the peer component, the services it implements, or its interface. Components do not even need to agree on an event type. Notification relations are not parameterized, shared-resource relations are based solely on the type of the resource.[2]

Besides the reusability aspects, encapsulating interaction in objects makes an important aspect of the protocol structure – the interactions between its components – explicit and visible. Explicit interactions allow us to reason about correctness and complexity of a protocol: in combination with the naming conventions, the reflection classes of Java can be used to check if

[2]It is up to the protocol developer to understand the semantics of the components to let them cooperate in a useful way. For example, a component triggering a notification upon a packet loss, should not be connected with a notifiee that expects a notification about the termination of a connection.

all notification components are plugged with a corresponding notifiee component, and if all resource-subscribers are provided with the resources they need to work. We think that the price we pay for reusability and correctness is not too high.

3.7 Other Aspects of Composition

This section shows how to compose finite state machines in PITOU, how to choose the underlying network services, and how to specify different thread-models.

3.7.1 Defining Finite State Machines

Finite State Machines (FSM) are a well-explored structuring concept used for many protocol specifications and implementations (the best known example is TCP [109]). We therefore decided to integrate the concept of FSM into our framework and make it cooperate with the vertical structuring approach.

Classes involved

The following classes and `interfaces` are involved in defining and using a FSM. A FSM is represented by the class `SessionStateManager`, which manages and coordinates all states. States are represented by the class `SessionState`. Each state has a name, a set of filters, and a set of transitions. A transition is defined by an object that triggers the transition (`EventRaiser`) and a target state. A filter is an object that defines two states *activated* and *deactivated*. Associating filters with states allows to inhibit operations (in our context execution of orders) given a certain state.

The class `StateManager` provides the following methods:

- The method `void signal(EventRaiser)` informs the current state object about an event.
- The method `void init()` initializes all states.
- The method `void addSessionState(SessionState)` adds a state to the FSM.
- The method `void setStart(SessionState)` sets the initial state of the FSM.

The class `SessionState` provides the following methods:

- The methods `void setName(String)` and `String getName()` allow to configure the name of the state.
- The method `void addTransition(EventRaiser,SessionState)` defines a transition from this state to a target state; the transition is executed when an event is fired from the specified event-raiser object.
- The method `void addFilter()` registers a filter object at a state. Filters are activated when the state is the current one, and deactivated when the current state changes.
- The method `enter()` makes the state active, i.e. makes it the current state in the FSM. The state now activates all filters and expects events for a transition.
- The method `void init(StateManager)` performs optimizations of the internal structure of a state machine and provides all event-raisers with a reference of the state-manager (used to allow the event-raisers to signal events).
- The method `SessionState transit(EventRaiser)` is called by the state-manager to signal an event to the current state. This method then determines and returns the new current state and deactivates all filters.

The `interface Filter` provides the following methods:

- the method `letNotPass()` activates, and
- the method `void letPass()` deactivates a filter.

Integrating Orders and FSM

Our FSM implementation uses the class `EventRaiser` to trigger transitions between states (which we already explained in the context of defining relations between components). That is, every order-type object (via `defineEvent_execDone()`) as well as any worker specified as notifier can trigger a state change.

Additionally, `OrderType` also implements the `Filter interface`. This allows us to avoid the execution of order-types that are not valid in a given state.

Scenario

The following scenario shows how a FSM interacts with the principal PITOU protocol components. Imagine we have an environment consisting of two order-types named *question* (emission) and *answer* (reception). We define a FSM consisting of two states named *idle* and *waiting*.

The initial state is *idle*. Each time a *question* is executed, the FSM changes to *waiting*. Each time a *answer* is executed, the FSM changes to *idle*. In the *idle* state, no *answer* is allowed to be executed. In the *waiting* state, no *question* is allowed to be executed.

The FSM for this simple protocol environment is specified as follows:

1. Create a session state registrar object:
   - `stateReg=new SessionStateRegistrar().`
2. Create two state objects.
   - `IDLE=new SessionState()`
   - `WAITING=new SessionState()`
3. Give names to the states.
   - `IDLE.setName("idle")`
   - `WAITING.setName("waiting")`
4. Specify the transitions between the states.
   - `IDLE.addTransition(question.defineEvent_execDone(), WAITING)`
   - `WAITING.addTransition(answer.defineEvent_execDone(), IDLE)`
5. Specify the filters for each state.
   - `IDLE.addFilter(answer)`
   - `WAITING.addFilter(question)`
6. Register states with the registrar object.
   - `stateReg.addState(IDLE)`
   - `stateReg.addState(WAITING)`
7. Register the state-registrar object with the environment object: `environment.setStateRegistrar(stateReg).`

During initialization, the reference to a `StateManager` object is given to all objects of type `EventRaiser`. The following scenario shows how the state of the environment changes when the first *question* order is executed.

1. After execution of the *question* order, the method `EventRaiser.raise()` is called on the respective event-raiser object

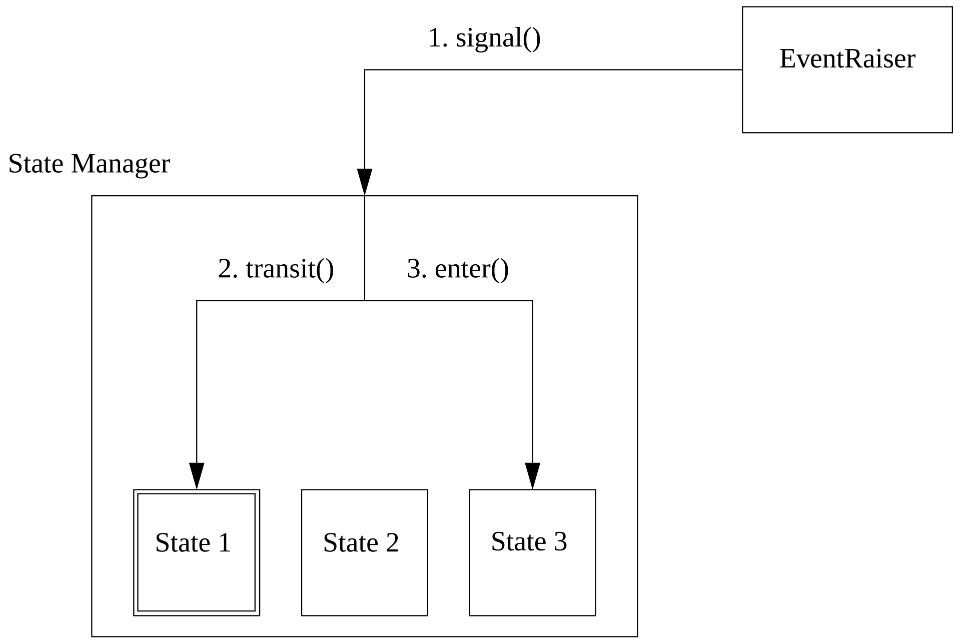
2. The event-raiser object calls `StateManager.signal(EventRaiser).`

3. Within the `signal()` method, the state-manager calls the method `State State.transit(EventRaiser)` on the object representing the current state, i.e. the IDLE state.

4. Within the `IDLE.transit()` method, all filters are deactivated (i.e. in our scenario, the *answer* order-type is no longer blocked) and the new state, i.e. WAITING, returned.

5. The state-manager now knows the new current state and applies the `WAITING.enter()` method on it. Within `enter()` all filters are activated (i.e. the *question* order-type blocked)

The transition between states is also illustrated in Figure 3.15.



State Transition:
1. an event is signaled to the state manager (e.g. by a worker)
2. the state manager determines the new state by asking the current state
3. the state manager activates the new state

Figure 3.15: State Transition

3.7.2 Specifying Thread Strategies

Classes involved

To allow for a maximum of flexibility for associating threads with order-types, we delegate the responsibility of executing an order to a special object of type `Executor`, which defines an `interface` with the following methods:

- The method `void executeOrder(Order,int)` takes the request for execution of an order and handles it.
- The method `isWorking()` indicates if an order is executed at the moment.

Order execution is now modified as follows. Upon the arrival of data, the method `OrderType.requestOrder()` is called as before to initialize an order object with potential data. Instead of calling the method `Order.execute()` directly as before, the order-type calls the method `Executor.executeOrder()`. The executor object of the corresponding order-type decides when and how to call the method `Order.execute()`.

The advantage of this design is that the initialization and execution of orders is decoupled. Executor objects are associated with order-types in many different ways. When each order-type has its own *active* executor object, it is automatically associated with its own thread. When all order-types share one *active* executor object, a single thread is responsible for all order-types. When all order-types share a *passive* executor object, orders are executed by the system thread (similar to a reactive model). Figure 3.16 shows the differences of the object relations when an executor is used.

PITOU provides two implementations of `Executor`. The class `ActiveExecutor` implement its own thread. The method `executeOrder()` puts the order object in a queue, from where the order object is read and executed by a thread. The class `PassiveExecutor` immediately executes an order in the `executeOrder()` method. The two execution models are depicted in Figure 3.17.

Associating Order-Types with Executors

The classes `ExecutorRegistrar` and `ExecutionGroup` are used to define the *thread-model* for a protocol environment object. Execution groups allow to register a set of order-type objects and associate them with an executor object. Executor registrars allow to register execution groups.

`ExecutionGroup` provides the following methods:

- The method `void addOrder(OrderType)` allows to register an order type object.

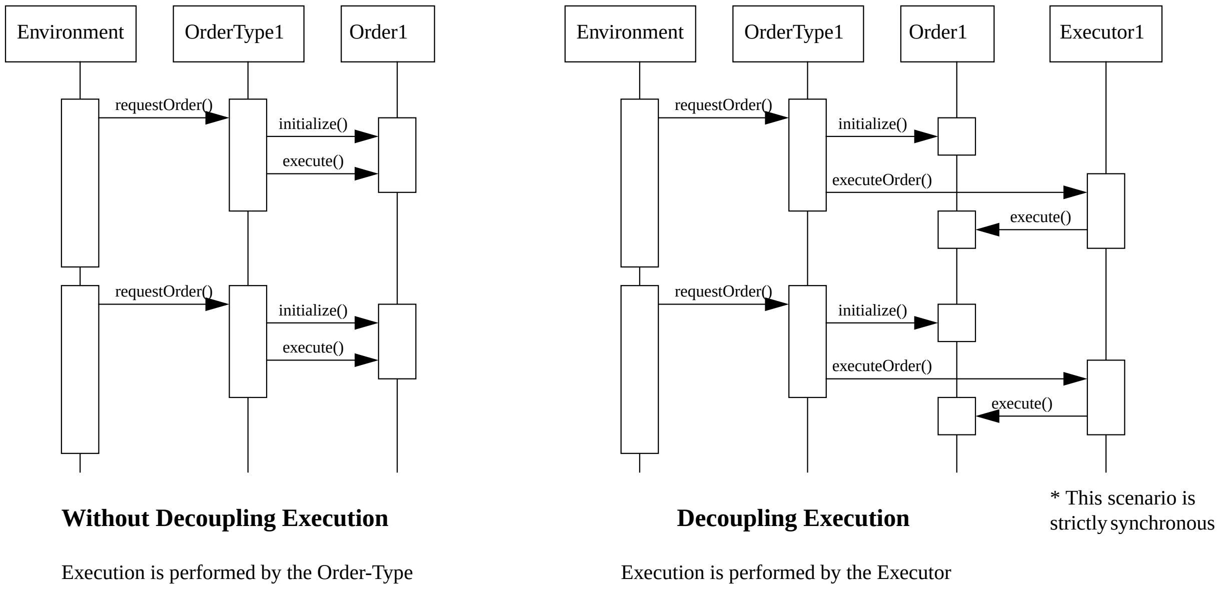


Figure 3.16: Decoupling Execution

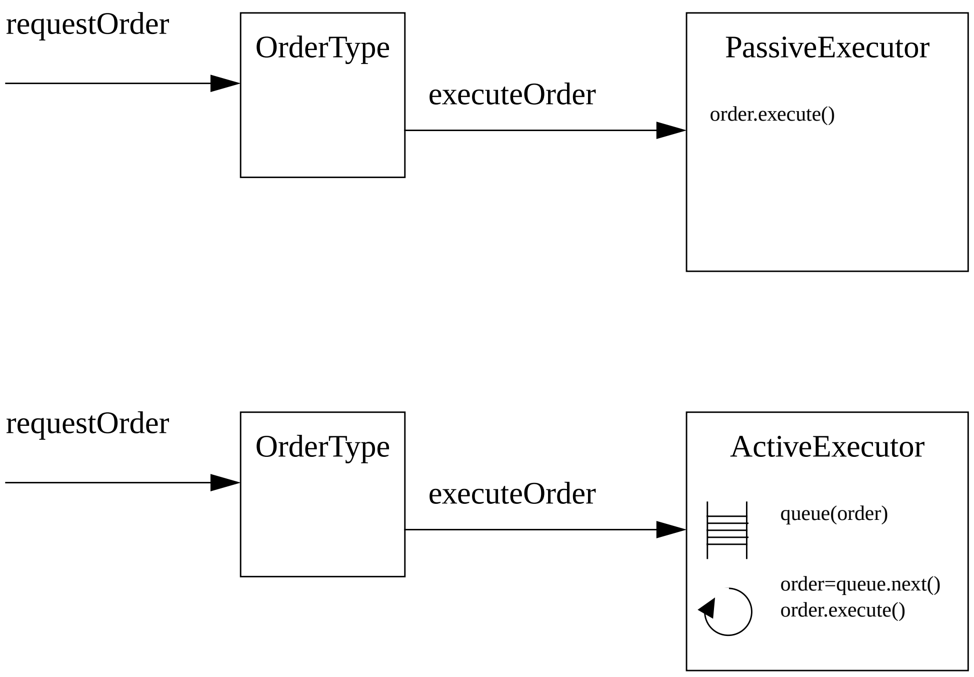


Figure 3.17: Passive vs. Active Executor

- The method `void setExecutor(Executor)` allows to set the executor object responsible for all order-types.

- The method `void init()` gives the executor object to all order-type objects.

`ExecutorRegistrar` provides the following methods:

- The method `void addExecutionGroup(ExecutionGroup)` registers another execution group.
- The method `void init()` calls the `init()` method of all registered execution groups.

Once all execution groups are defined and registered, the executor-registrar object can be given to the environment via `Environment.setExecutorRegistrar()`.

3.7.3 Registration of SAPs

The class `SAPRegistrar` is used to register all kind of SAPs for an environment. SAP-registrars provide the method `addSAP()` for registration.

The order of registration is important, because the SAP objects are referenced by an *index* that corresponds to the position of the respective SAP object in the registration list of the SAP-registrar. Each reception- and emission-type and each mapping strategy object specify a SAP number that corresponds to the position of the SAP in the order-registrar. Figure 3.18 shows how an emission-type object specifies a SAP via an index, and obtains the corresponding object after initialization of the environment.

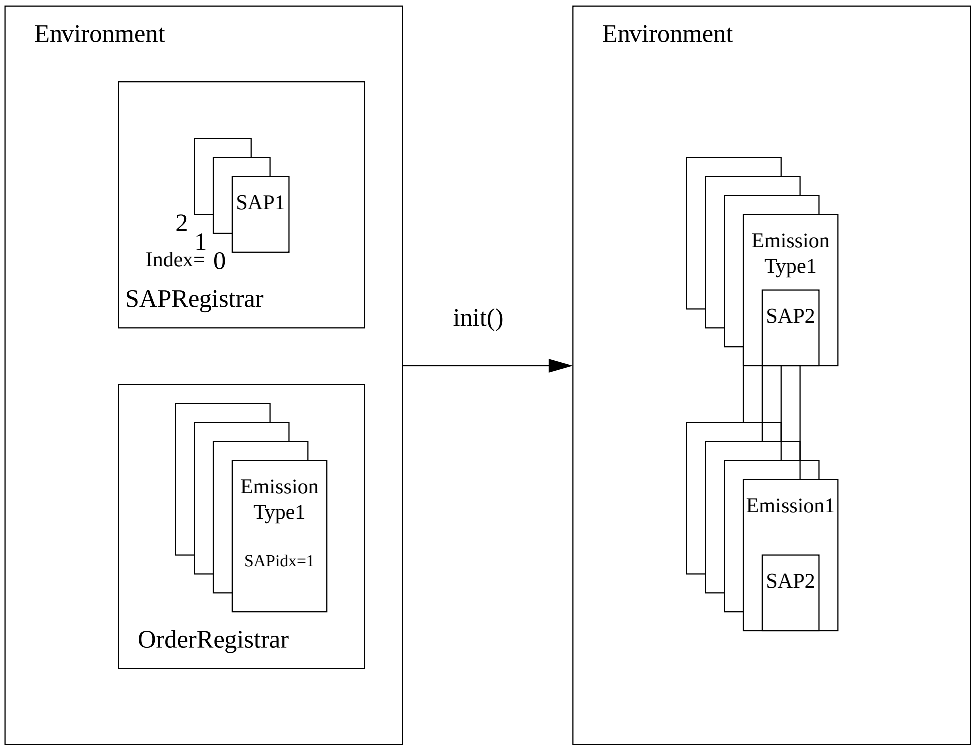
Once all SAPs are registered, the SAP-registrar object can be given to the environment via `Environment.setSAPRegistrar()`.

3.7.4 Specifying Mapping Strategies

The class `MappingRegistrar` is used to register all mapping-strategies for an environment. Each mapping strategy must specify the number of the SAP it belongs to. Mapping-registrars provide the method `addMappingStrategy()`. Once all mapping-strategies are registered, the mapping-registrar object can be given to the environment via `Environment.setMappingRegistrar()`.

3.7.5 Exception Handling

An **exception** is *a signal that indicates that some sort of exceptional condition (such as an error) has occurred* [55]. The Java keyword `throw` is used to signal an exception. The Java keyword `catch` is used to handle it. Exceptions signaled are objects that propagate up in the



During the initialization of the environment, the registered SAP objects are given to the emission-type and reception-type objects that specify the corresponding index (which refers to the order of registration) and to the created emission/reception order objects.

Figure 3.18: Associating SAP objects

hierarchy of method calls until they are handled. The advantage of this language feature is that it allows to group related errors and centralize their handling.

In PITOU, we have two categories of exceptions. The first category comprises exceptions that are signaled during the construction and initialization phase. They are represented by the class `ProtocolConstructionException` and signal a serious flaw of the protocol structure when the following problems occur.

- The type of the parameters of the specified entries are not the ones a worker expects. The exception is thrown in the `Worker.init()` method.
- A session-state defines a target state that is not known by the session-state registrar. The exception is thrown by `SessionStateRegistrar.init()`.
- Objects are registered more than once at the same registrar. This exception is signaled in the registration methods of all registrars (e.g. `OrderRegistrar.addOrder()`).

- Order-types or mapping strategies reference SAP numbers that are not valid, e.g. an reception-type specifies *SAP No. 3* as output, but only 2 SAP objects are registered at the `SAPRegistrar`. This exception is signaled in the `Environment.init()` method.
- The identifiers of emission-types or reception-types are either not defined or not unique. This exception is signaled in the `OrderRegistrar.init()` method.

Exceptions concerning flaws of the protocol structure should not be handled, but directly lead to termination of the program. They are such serious that a protocol environment should not enter the execution phase.

The second categories of exceptions comprises those that are signaled during the execution phase of a protocol represented by the class `ProtocolExecutionException`. The following exceptions are specializations of `ProtocolExecutionException`.

- Any problem experienced during initialization (e.g. the input byte-array is too long or too short), execution (e.g. a worker stops execution), or serialization of an order (e.g. an entry is empty) will result in signaling an `OrderException`. This exception is handled by the respecting order-type object and avoids that an order is further executed, delivered, or sent.
- Any problem that occurs during the operation of an entry (e.g. initialization, serialization, manipulation) will cause an `EntryException`. This exception is handled either by workers (which *may* transform them into a `WorkerException`) or by orders (which *will* transform them into a `OrderException`).
- Any problem that occurs during the operation of a worker (either in `call()` or any other method of a worker), will result in signaling a `WorkerException`. This exception is handled by the executing order object in the `Order.execute()` method and causes the immediate termination of the corresponding order.
- When network data can not be mapped to a reception-type object, a `MappingException` is fired by the mapping strategy in the `map()` method. Handling is done by the environment, which just ignores the data.

The handling of execution exceptions described above is done internally, i.e. inside the protocol classes without notifying the application. Since it would be useful for the application to know about the exceptions that occur, we integrated the following simple mechanism.

We define an `interface ExceptionHandler`, which declares the method `void exceptionOccured(Exception).` The `Environment` provides a method `void setExceptionHandler(ExceptionHandler)`, which allows an application to give the environment an application-specific implementation of the `ExceptionHandler interface`. During initialization, the environment gives the exception handler object to all order-type objects. Every time an order exception occurs (and an exception handler is set by the application), the order-type calls the method `exceptionOccured()` and thus allows the application to intervene.

There are two more exceptions that must be handled by the application itself:

- When an environment has not yet entered the execution phase, it is not ready to accept or receive data. Every attempt to call the methods `accept()` or `netData()` lead to a `EnvironmentNotActiveException`. The same exception type is used when the environment is accessed after its deactivation.

- Every problem during reading from or writing to a SAP object results in signaling a `SAPException`.

3.7.6 Incorporating Environments in the Application

In this section, we shortly sketch the issues to integrate a PITOU protocol into a distributed application.

Creation of Environment Instances

During the construction phase, an environment object is instantiated and supplied with all necessary objects. However, an application normally wants to use more than just one instance of a certain environment. One solution to create new environment instances would be the use of the *Prototype*[59] pattern. That means, the environment class and all other classes that make up the environment (order-type, worker, entry, state, etc.) would need to provide a `clone()` method to reproduce exactly the same structure and configuration.

We propose a simpler way to create new environment instances. We define an abstract class called `ProtocolBuilder` that defines the abstract method `Environment construct()` and throws a `ProtocolConstructionException`. The `construct()` method encapsulates the code executed during the construction phase, and returns an environment object. Applications then use a protocol builder object to create new environment objects.

The application may decide if the `construct()` method also defines the `SAPRegistrar`. If the application wants to be able to change the SAPs during execution (e.g. to cope with SAPException), it may use the `SAPRegistrar SAPRegistrar.create()` method to create clones of the registered SAP types, and assign them at any time to the environment via `void changeSAPs(SAPRegistrar)`.

Communication Subsystem

The application is responsible to implement reading packets from the `SAP` objects, and give the data to the `Environment.netData(Packet)` method. The application may read packets in a non-blocking way within a single thread for all SAPs (e.g. in the `main()` of the application) to minimize the number of threads. It may alternatively implement one thread for each environment or even for each SAP object and perform blocking reads to better utilize CPU resources. The design of the environment is hence independent from application design issues. Since treating all application issues are out of the scope of this dissertation, we don't make propositions how the communication sub-system of an application can be designed for reuse and flexible change. The interested reader is referenced to work of Douglas Schmidt [126].

Closing Environments

In order to be notified about the fact that an environment considers itself *closed* and is ready to be deactivated, an application must implement the `interface CloseListener`, which declares the abstract method `void envClosed(Environment)`. An environment considers itself to be closed if any component triggers the execution of the method `Environment.notifyTerminate()`.

Note that a *closed* environment is not yet deactivated. In order to give the application full control over the environment, we decided to make the application responsible to call the method `void deactivate()`, which then closes all allocated resources.

3.8 Design Patterns Used

In this section, we review the design patterns we used for our framework.

3.8.1 Layers

The principle of layering is also known as an architectural design pattern called *Layers* pattern [31].

At the one hand, we seek to overcome layering in the organization of protocol software, since layering hardly allows flexible composition of fine-grained, reusable components. At the other hand, we apply *Layers* by building the protocol of a distributed application on top of the protocol stack of the operating system, since we can thus reuse standardized, efficient, and robust services. This examples shows the interesting case where a design pattern can both enable and obstruct reusability in the same system.

3.8.2 Configurable Architecture

The intent of *Configurable Architecture* [132] is to *provide a way to influence system behavior and capabilities, not only by programming or run-time input, but by giving a configuration mechanism. This allows the system to vary its behavior from installation to installation without re-compilation*.

Configurable Architecture consists of the following participants:

- **Configurable System**: initializes itself via the configuration values, processes input according to configuration. The *Configurable System* is the generic implementation of the system using the *Configuration Storage* to retrieve its *Configuration* during its start up. In PITOU, the `Environment` class represents a *Configurable System*.

- **Configuration Storage**: represents configuration data, allows editing of configuration by authorized entity. The *Configuration Storage* is the mechanism that keeps the configuration values persistent and provides them during startup. In PITOU, the various registrar objects act as *Configuration Storage* (in contrary to e.g. configuration files, or data bases).

- **Installation**: is the combination of the system with a concrete set of configuration values, serves as a unit of system operation. In PITOU, an environment becomes an installation when all the registrar objects are assigned and initialized.

- **Configuration**: represents the data values controlling the system operation. The *Configuration* is the concrete set of values stored in the *Configuration Storage*. It represents the concrete behavior of the *Configurable System*. In PITOU, all order-type objects with their combination of workers and entry-type objects, states, out-of-band modules, etc. as well as their configured properties form the *Configuration*.

The PITOU framework is an extreme form of a *Configurable System* since it offers a high degree of flexibility. This pattern is a very general and abstract one that lets the designer many degrees of freedom. We found the pattern very interesting since it helped us to weight forces (e.g. flexibility against performance) and identify crucial design issues (e.g. the importance of a good structure).

3.8.3 Manager

The intent of *Manager* [133] is to *put functionality that concerns all objects of a class into a separate managing object*. This separation allows the independent variation of manager functionality and its reuse for different object classes.

- **Manager**: treats the collection of managed objects as a whole, deals with issues related to accessing, creating, or destroying managed objects.
- **Subject**: represent a managed object, which perform a certain task on behalf of a *Client*
- **Client**: retrieve a specific managed object from the *Manager*.

In PITOU, `OrderType` serves as a *Manager* for the `Environment` object (*Client*) by managing objects of type `Order`. Execution of orders (as well as creating, initialization, and deactivation) is possible only by accessing the *Manager*. We used the *Manager* pattern to hide the implementation of thread-management and object pool management behind a stable interface. The class `Environment` is also a *Manager* that controls access to objects of type `OrderType` from the application (*Client*).

3.8.4 Type Object

The intent of *Type Object* [76] is to decouple dynamic from static information of an object. Type Object allows new ”types” to be created dynamically at runtime, lets a system provide its own type-checking rules, and can lead to simpler, smaller systems. *Type Object* consists of two participants: one that represents objects, one that represents the type of an object.

In the PITOU framework, *Type Object* is applied to orders (`Order` – `OrderType`) and entries (`Entry` – `EntryType`). In both cases, the goal is to separate static and dynamic aspects of a protocol. The type objects (order-type, entry-type) are used to define the structure of a protocol, while the dynamic objects (order, entry) are internal framework classes and implement the behaviour of their ”types”.

The use of *Type Object* allows us to decouple protocol composition from execution.

3.8.5 Adapter

The intent of *Adapter* [59] (also known as *Wrapper*) is to *convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces*.

In the PITOU framework, we apply the *Adapter* pattern to SAPs. The adapter consists of the following participants:

- **Target**: defines the domain-specific interface a client uses. In PITOU, the `SAP` interface represents the target of the *Adapter* pattern
- **Client**: collaborates with objects conforming to the Target interface. In PITOU, any class accessing `SAP`s are clients (e.g. `Environment` sending, application sub-system classes reading)
- **Adaptee**: defines an existing interface that needs adapting. In PITOU, all socket interfaces that are encapsulated behind `SAP`s are adaptees (e.g. the Java Socket and DatagramSocket)
- **Adapter**: adapts the interface of Adaptee to the Target interface. In PITOU, all classes in `pitou.net`, i.e. `TCPSAP`, `UDPSAP`, `MCSAP`, which implement the `SAP` interface, are adapter classes.

The use of *Adapter* allows us to decouple specifics of underlying network service interfaces from the protocol. We can thus change the underlying network or transport protocol without affecting the application-tailored protocol.

3.8.6 Strategy

The intent of *Strategy* [59] is to *define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it*.

In PITOU, we apply *Strategy* for mapping, and thread-handling.

Strategy consists of the following participants:

- **Strategy**: declares an interface common to all supported algorithms. In PITOU, `MappingStrategy` defines the interface for mapping algorithms, `Executor` the interface for execution algorithms with regard to the specified threads.
- **ConcreteStrategy**: implements the algorithm using the Strategy interface. `DefaultMappingStrategy` is an example for *ConcreteStrategy* in the context of mapping, `PassiveExecutor` in the context of execution.
- **Context**: is configured with a *ConcreteStrategy* object, maintains a reference to a *Strategy* object, may define an interface that lets *Strategy* access its data. In PITOU, the `Environment` serves as *Context* for mapping strategies, `Order-Type` as *Context* for execution strategies, and `Order-Type` as *Context* for worker strategies.

3.8.7 Observer

The intent of *Observer* [59] (also known as *Publish-Subscribe*) is to *define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*.

In PITOU, we apply *Observer* to allow interaction between application and environment (`Environment`$\longrightarrow$`CloseListener`), and for notification among components (`EventRaiser`$\longrightarrow$`Caller`).

Observer consists of the following participants:

- **Subject**: knows its observers, provides an interface for attaching and detaching observer objects. In PITOU, `EventRaiser` and `Environment` are examples for a *Subject*. `EventRaiser` provides the methods `addListener(ProtocolEventListener)`, `Environment` the method `addCloseListener(CloseListener)`.
- **Observer**: defines an updating interface for objects that should be notified of changes in a subject. In PITOU, `ProtocolEventListener` and `CloseListener` are examples for Observers.
- **ConcreteSubject**: stores state of interest to *ConcreteObserver* objects, sends a notification to its observers when its state changes. `EventRaiser` and `Environment` are examples for *ConcreteSubject*. In our design, *Subject* and *ConcreteSubject* are one and the same object.
- **ConcreteObserver**: maintains a reference to a *ConcreteSubject* object, stores state that should stay consistent with the subject's, implements the *Observer* updating interface to keep its state consistent with the subjects. In PITOU, all classes implementing `Caller` are examples for *ConcreteObserver* that implement `ProtocolEventListener`. Since no call-back is needed in this case, caller classes do not need to maintain a reference to an event-raiser. Applications implementing the `CloseListener` interface are another example of *ConcreteObserver*.

While *Observer* decouples the implementations of information sources and sinks, it nevertheless creates a dependency on interface level – a *Subject* must know the type of its *Observer*. This is the reason why we use interaction reification instead letting components interact via *Observer*.

3.8.8 Composite

The intent of *Composite* [59] is to *compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uni-*

formly.

Composite consists of the following participants:

- **Component**: declares the interface for objects in the composition, implements default behavior for the interface common to all classes, declares an interface for accessing and managing its child components, defines an interface for accessing a component's parent in the recursive structure (optional). In PITOU, `CompositeEntryType` is an example for a *Component*.
- **Leaf**: represents leaf objects in the composition, a *Leaf* has no children. In PITOU, classes like `SeqNrEntryType`, `IntegerEntryType`, or `LongEntryType` are examples for *Leaf*s.
- **Composite**: defines behavior for *Components* having children, stores child Components, and implements child-related operations in the Component interface. In PITOU, `BlockEntryType`, `BitBlockEntryType`, and `ArrayEntryType` are examples for a *Composite*.

The *Composite* pattern allows us to introduce a high flexibility and generality in composing new entry-types. This reduces the need for implementing protocol specific entry-type objects and simplifies the workers that use composite entries.

3.8.9 Active Object

The intent of *ActiveObject* [123] is to *decouple method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control*.

Active Object consists of the following main[3] participants:

- **Servant**: provides a method
- **Client**: wants to invoke a method of a *Servant*
- **Proxy**: provides a similar interface as the *Servant*, but transform method invocations into method request objects and enqueues those at a *Scheduler*
- **Scheduler**: implements a thread that reads method requests from the queue, calls the *Servant*, and writes the result to a so called future object, where the client can obtain the result from

[3]The original description consists of some more participants that we consider rather unimportant to understand the idea of this pattern

Active Object works as follows: a client that intents to asynchronously invoke the method of a *Servant*, invokes the method of the *Proxy*. The *Proxy* creates a method request object (including a future object) and queues it at the *Scheduler*. The *Scheduler* runs in its own thread an reads the method request objects from its queue, and triggers the invocation of a method of the *Servant*.

`ActiveExcutor` is an example for the use of *ActiveObject* in the PITOU framework. `InternalOrderType` is a Client (who wants an order to be executed). `ActiveExecutor` is both *Proxy* (since it provides the `execute(Order)` method), and *Scheduler* (since it implements its own thread and maintains a queue). `InternalOrder` acts as *Servant* (since it implements the `execute()` method).

3.9 Summary

In this chapter, we presented the PITOU framework (**P**rotocol **I**mplementation,**T**ailoring, and **O**rganization **U**tilities). The principal goal of PITOU is to provide a framework for rapid prototyping of protocol software that is easy to use, helps to build a library of protocol components, and provides an evaluation platform for our structuring approach from Chapter 2 and the software concepts used. PITOU protocols are implemented in Java and are supposed to be incorporated in the application that uses them.

Working with PITOU can be divided in two principal tasks: first, the implementation of components for the library, and second, the assembly and configuration of components into a new application. PITOU supports the implementation of components by providing abstract classes or interfaces that must be implemented. Assembly is guided by following a rigid structure and by the emphasis of classes that allow to plug the components implemented into the framework (Hot Spots).

In a first step, we used the Java-Beans component model and the Visual-Age (VAJ) builder tool to compose protocols. However, VAJ has problems to cope with a large number of connections between components. We therefore proposed a design approach that reifies interaction between components in own objects and we separated structure information in so called *registrar* objects from information used at execution time. We finally explained how to specify finite state machines, different thread strategies, and the organization of exception handling in our framework. A description of the design patterns used in the implementation of PITOU completed this chapter.

The design presented in this chapter extends our main model of Chapter 2 by the following concepts:

- the concept of using *object pools* for orders and entries,
- the concept of using *executors* to decouple execution from the thread-model used,

- the concept of using *registrars* to decouple the definition of a protocols structure from its runtime behavior, and
- the concept of using *interaction reification* to allow for interaction between arbitrary components.

Chapter 4

Tools to Support Protocol Implementation

This chapter gives an overview of several tools within the PITOU framework that assist in the implementation of protocols. The PITOU simulator allows to run protocol code without any modification in a virtual time environment over a simulated network. The PITOU protocol editor allows to combine structures of different protocol objects. The PITOU code generator is able to analyze the structure of a protocol object, and to transform it into Java code. The PITOU protocol animator allows to visualize the behavior of protocols.

4.1 Protocol Simulation

4.1.1 Objectives

Protocol software is difficult to test since the conditions of communication change frequently (especially in a best-effort network as the Internet), and it is therefore difficult to reproduce an experienced problem and to locate its origin. Instead of running communication software at different machines, it would be easier to run a protocol session in a single application on a single machine over a virtual network.

The PITOU protocol simulator addresses exactly these issues. However, there are a number of other important objectives we want to realize.

- **no modification of protocol code**: any protocol implemented within PITOU must be able to run in simulation mode as well as for transmissions over a real network without any modification.
- **fine-grain simulation**: instead of assigning a virtual time to a complete protocol instance, our simulator should allow to flexibly assign virtual times to each data path. While most

network simulators do not consider protocol processing as significant, our simulator has the objective to gain insights about the efficiency of an implemented protocol, and to identify bottlenecks and room for optimization.

- **virtual parallelism**: we want to be able to run simulations as a single program, i.e. a single operating system process. Since communication sessions involve at least two independent communication entities (communicating applications) it is important that the simulator provides means to distinguish activities that can happen in parallel (simulated as more than one process) from activities that can not take place at the same time (simulated as a single process).

After reviewing related work, we will give an overview of the simulation model and the classes involved in simulation. We then show how the simulator cooperates with the PITOU framework and how to use it.

4.1.2 Related Work

Most communication simulation tools are rather specialized to network simulation than to protocol simulation. There exist a number of network simulators like NS [104], Entrapid [66], or NetSim [17] with a broad scope and different goals. None of them allows for generic, fine-grained simulation of protocol software.

The PITOU simulator provides a class library for discrete event simulation [97] similar to Simula [107], which can be used to build any kind of process simulation. In contrary to protocol simulators for a certain protocol (e.g. [37]), protocol simulation in PITOU is completely generic. Due to the clear structure and the proper decoupling of protocol and framework code, our simulation tool is capable to simulate very fine-grained virtual processes within a protocol. Any protocol implemented in PITOU can be simulated without changing any line of code.

4.1.3 Simulation Model

Our simulator follows the **discrete event** [97] based simulation model. In contrary to a *continuous* simulation model where the state of the simulation system varies continuously with time, in *discrete event* based models the state changes upon events.

Applied to PITOU, an event can be the arrival of data, the request for executing an order, the request for a transmission of data over a link, or a timeout. Each of these events indicates that a certain **activity** needs to be executed: arrival of data results in mapping data to an order-type, a request for a new order results in its execution, a transmission request in moving data to another communication entity, a timeout in calling a method that handles the timeout. A **scheduler** is the central entity to handle events, to execute activities, and to maintain the virtual time that advances by virtual duration values associated with each activity.

A scheduler maintains different **processors**. Activities that are executed on the same processor can not be scheduled at overlapping virtual time intervals, but must be scheduled one after the other. Activities that are executed on different processors can be scheduled virtually in parallel, i.e. in overlapping virtual time intervals. We say that a processor is *sleeping* when he has no activity scheduled. As soon as an activity is scheduled the respective processor becomes *waiting*. A processor is *active* during execution of an activity. Once an activity is executed it can not be un-done, i.e. there is no rollback mechanism integrated.

4.1.4 Classes Involved

All classes concerning the simulator are grouped in the package `pitou.simulator`.

Activities are represented by an `interface` called `Activity`, which declares the following methods to be implemented:

- The method `void perform()` must implement the code to be executed.
- The method `void setScheduler(Scheduler)` provides activities with a reference to the scheduler, which allows them to schedule other activities or to be notified about the virtual time.
- The methods `int getPriority()` and `void setPriority(int)` allow to configure the priority of activity objects, which is relevant for scheduling.

Processors contain the scheduling information for all their activities scheduled in form of objects of type `ScheduleInfo`, which contain a reference to a scheduled activity, its duration, its execution time, and its priority. Processors are represented by the class `Processor`, which implements the following methods.

- The method `void scheduleAt(Activity,long,long)` produces a schedule-information object with the specified parameters (activity, execution time, duration), and sorts it in the information queue (sorted by the execution time).
- The method `void scheduleAtEnd(Activity,long)` produces a schedule-information object and adds it at the end of the information queue.
- The method `void cancel(Activity)` removes the schedule-information object that refers to the specified activity from the information list.

- The method `Activity getNextProc()` is called by the scheduler to obtain the next scheduled activity to be executed.
- The method `int getNrScheduled()` returns the number of activities scheduled for this processor, i.e. the number of information objects in the queue.
- The method `long getNextExecTime()` returns the execution time of the next activity of this processor to be executed.
- The method `long getNextDuration()` returns the duration of the next activity of this processor to be executed.
- The method `void passivateFirst()` is called by the scheduler to remove the first information object from the list and update internal variables (like the execution time of the next activity).
- `Processor` also implements the `Sortable interface`, which allows the scheduler to manage an ordered list of processors (ordered by the execution time of the first scheduled activity for all waiting processors) by using the methods `boolean equals(Sortable)` and `boolean greater(Sortable).`

The scheduler is implemented by the class `Scheduler`. A scheduler runs in its own thread and implements the interface `TimerCompatible` to detect via a timeout (based on *real* time) that the virtual time has not advanced.

The scheduler provides an interface to register activities together with their processor in advance. During execution, the scheduler will only accept to schedule activity objects that have been registered before.

The scheduler maintains a virtual time and a queue containing references to the processors that have at least one scheduled activity (called *waiting* processors). The queue of waiting processors is sorted by the virtual execution time of the next activity to be executed for each processor. Within its thread, the scheduler takes the first element from the queue of waiting processors, obtains the first scheduled activity from this processor, and executes it. When the processor has still other scheduled activities, it is re-sorted in the queue. Otherwise it is left out. The scheduler then sets the virtual time to the execution time of the last executed activity, and keeps in mind the termination time of the last executed activity (=execution time + duration). When no other processor is waiting, the virtual time is set to the termination time of the last activity executed. Otherwise, the scheduler thread continues to read its queue of waiting processors.

The class `Scheduler` provides the following methods.

- The method `void register(Activity,Processor)` registers an activity together with its processor. Only registered activities can be scheduled during execution time of the scheduler.

- The method `void register(Activity)` registers an activity with a processor. Internally, the scheduler will create and assign an own processor to this activity.
- The method `void deregister(Activity)` deregisters an activity.
- The method `void addVirtualTimeListener(VirtualTimeListener)` allows to register objects that are interested in being notified every time the virtual time changes.
- The method `void removeVirtualTimeListener(VirtualTimeListener)` deregisters objects that have been notified about the virtual time.
- The method `void scheduleAt(Activity,long,long)` schedules an activity by specifying the time when execution should start (execution time) and the duration of the execution.
- The method `void scheduleNow(Activity,long)` schedules the activity as soon as possible (with respect to priorities). The parameter of type `long` specifies the duration.
- The method `void scheduleAfterRest(Activity,long)` schedules the activity as the last one for its processor. The parameter of type `long` specifies the duration.
- The method `void cancel(Activity)` cancels an activity before execution. The scheduler identifies the corresponding processor, and calls `Processor.cancel()`.

Waiting processors are ordered by the scheduler with regard to the execution time of their first activity scheduled. When two activities are registered with the same processor, the following rules apply.

1. an activity is scheduled before another activity of the same processor, if its execution time is earlier
2. if the execution time is the same, the activity with the higher priority is scheduled first
3. if execution time and priority are the same, the activities are scheduled in the order of request

Our scheduling rules correspond to a *non-preemptive* model.

For a detailed example of the operation of the simulator, see Appendix B.

4.1.5 Integrating the Simulator in PITOU

The most important goal of our simulator is to allow for the simulation of protocol software *without* changing any protocol code. That is, the protocols build with PITOU should run both in *real mode* and in *simulation mode*.

In the following, we list the differences between real mode and simulation mode, and shortly sketch what modifications are necessary to allow our framework to support both modes.

- **No real network**: we need a simulated transmission channel, and a service access point that makes access to the channel transparent for the protocol code. The virtual transmission channel is represented by an object of class `Link`, the service access point is represented by the class `SAPvS`, which is an implementation of the `SAP` interface and integrates simulation functionality.

- **Time is virtual**: all timer handling must be based on the notion of virtual instead of real time. In order to assure that workers are not concerned by the different notion of time, we simply replace our timer-pool by another timer-pool, which manages virtual time and virtual timers transparently to other parts of the framework.

- **Control is centralized**: instead of assigning Java threads to order-types, all control is shifted to the scheduler's thread. We mapped the various tasks of the PITOU framework to the following simulation activities:

  - order execution: the initialization (delegating parsing to the entries) and execution (calling the workers) of an order, is scheduled and executed as one activity with a configurable virtual duration. Hence, each order-type obtains its own activity object.
  - sending: sending data over a virtual network is scheduled and executed as one activity with a virtual duration that depends on a configurable data rate.
  - transmission: the transmission process is represented by an activity with a virtual duration that depends on a configurable transmission delay.
  - timeout: each evocation of the `timerCall()` method is performed within an activity scheduled by the timer-pool (the virtual duration is equal to zero).

- **Initialization**: we must respect the interfaces and definitions imposed by the simulator. Where is initialization done, which objects need a reference to a scheduler object, where do we need to introduce wrapper classes?

For a detailed description of our solutions to all these design issues see Appendix C.

4.2 Protocol Edition, Validation, and Generation

In this section, we present two tools that aim to further ease the programming of protocols within the PITOU framework. The first tool is a **protocol editor** (PE), which allows to modify and validate the structure of protocols. The second tool is a **code generator** (CG), which allows to produce Java code given an environment object. Both tools are especially useful when applied together. The corresponding classes are grouped in the Java `package pitou.tools`.

4.2.1 Motivation

After experiencing with protocol software built with PITOU, we realized that it is difficult to combine different protocols or to reuse complete orders across different protocols. We sometimes want to take a part of one protocol implementation and integrate it into another one, build one protocol on top of another, or remove a certain order-type with all its relations from an implementation. All these issues are addressed by the PITOU protocol editor. Furthermore, the editor also allows to validate protocol implementations. The modified protocol structure can be transformed into a Java implementation by using our code generator.

4.2.2 Overview

As we saw in Chapter 3, a PITOU protocol is represented by a set of structured objects in registrar objects. All static protocol information, i.e. all information concerning structure and configuration of a protocol, is encapsulated in environment instances (i.e. in the respective registrar objects) accessible during runtime. Accessing the structure of a protocol at runtime allows us also to modify it at runtime. Designing our protocol editor was thus a rather easy task: we could simply operate on objects and dynamic lists instead of e.g. defining a representation language, writing a parser to identify the structure, and mapping it back to Java.

Using the protocol editor usually comes in four steps:

1. Instantiate a protocol editor object with an environment object.
2. Manipulate the structure of the environment object by using the various features of the protocol editor.
3. Validate the correctness of the protocol structure.
4. Export the environment object with the new structure.
5. Generate Java code for the protocol environment.

The protocol editor is implemented in the class `ProtocolEditor`. A protocol editor object is instantiated together with an object of type `Environment`. After instantiation, the protocol editor analyses the environment structure by looking into all related registrar objects, and builds up an internal information structure that allows him to access information during modification in a faster way.

Once the internal information structure is built, the protocol editor is ready to accept modifications. After the modifications, the method `Environment produceEnvironment()` is used to return the environment object with the new structure.

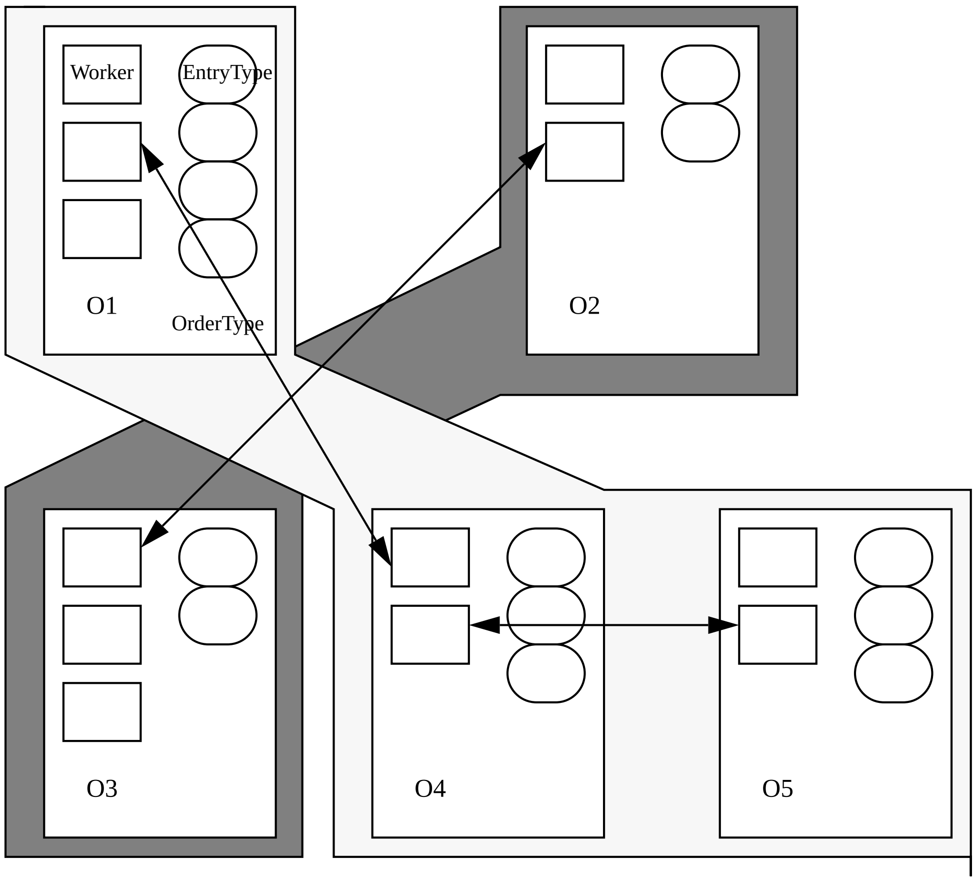
4.2.3 Main Operations on Protocols

Simple modifications like inserting and removing worker, entry-type, out-of-band module, or relation objects can be done within the various registrar objects. The idea of the protocol editor is rather to provide advanced operations on larger parts of the protocol software rather than on objects, up to merging and layering complete protocols. In the following, we describe only the most important features of the protocol editor. We leave out some other useful features like the merging of mapping strategies or the merging of finite state machines.

Identifying Clusters

All order-types with workers that maintain relations to workers of other order-type objects somehow depend on each other. A notifying worker needs a worker to be notified. A resource subscriber needs a worker that provides him with the resource required. For example, removing an order-type without considering the relations of its workers will lead in most cases to flaws. We therefore introduce the notion of a **cluster** of order-types. A cluster is a set of all order-types within an environment that are directly or transitively connected via relations between their workers (similar to the *transitive closure* of mathematical relations or graphs). Figure 4.1 depicts an example protocol environment that consists of two clusters. A cluster may consist of only a single order-type, when this order-type has no relations with another order-type (i.e. no shared resource or notification relations of its workers) and is not created by a worker of another order-type.

Our protocol editor maintains all dependencies between the order-type objects (also those created by out-of-band modules), and allows to identify, export, and remove for each order-type the corresponding cluster of order-types.



Cluster 1: O1,O4,O5
Cluster 2: O2, O3

Figure 4.1: Cluster of Order-Types

Merging and Layering Order-Types

Merging order-types allows to reuse complete order-types across different protocol environments. One just needs to take the respective order-type object from another protocol environment, insert it into the protocol editor, and **merge** the order-type with another one. Merging order-types means simply appending worker lists, entry-type lists, and parameter relation lists of two order-types. Thereby, the first order-type will continue to exist while the second one will be removed from the environment.

Imagine now that you have one order that calculates a checksum out of all entries of an order. When you would merge another order-type with this order-type, you would need to adapt the parameter relations of the resulting order-type since a worker of one order-type needs to operate on entries of the other order-type. To support this case generally, our protocol environment provides **layering** of order-types. The difference between merging and layering two order-types is that layering hides the information of a higher layer to the lower layer. All entry-types of the order-type representing the higher layer are mapped to one entry-type of the lower layer order-type specified as **payload-entry**. During the layering operation, the payload-entry-type will be replaced by a composite entry-type that comprises all the entry-types of the higher layer order-type. All parameter relations (of the lower layer part) that involve the payload-entry are replaced by parameter relations between workers of the lower layer order and the set of entries belonging to the higher layer order. Figure 4.2 illustrate the difference between merging and layering.

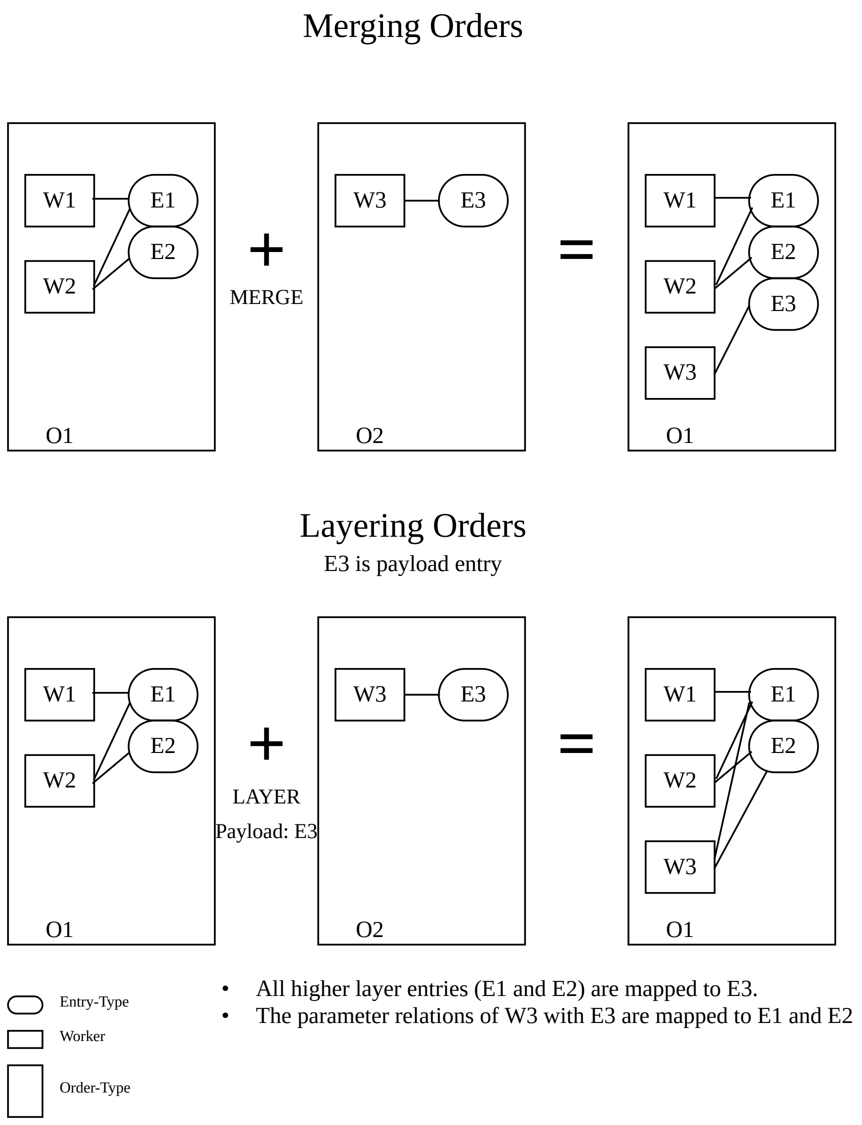


Figure 4.2: Merging vs. Layering of Order-Types

Cloning

When the functionality of an order-type should be merged or layered with *more than one* order-type of an environment, we need to create a clone of that order-type (otherwise the order-type becomes a part of the order-type it is merged with and is not any longer accessible for another merge operation). The protocol editor provides three different types of cloning order-types.

- **Flat cloning** produces a new order-type object with an order-registrar that contains the same worker objects as the original order type. The entry-type objects are cloned and the parameter relations are established between the workers and the new entry-type objects. The interaction relations between workers remain unchanged.

  Flat clones are useful when two orders want to share the same worker instances (e.g. for reasons of efficiency or to share state).

- **Deep cloning** produces a new order-type object with an order-registrar that contains not only clones of the entry-type objects, but also clones the worker objects. The parameter relations are established between the new worker and entry-type objects. The interaction relations are cloned and registered only if they make sense. Since two notifiers for one notifiee or two publishers for one subscriber would make no sense, the programmer must establish interactions between the concerned worker by himself.

  Deep clones are useful used when worker instances should not be shared among different orders.

- **Cluster cloning** produces deep clones of all order-type objects that are part of a cluster. That is, the complete structure including order-types, workers, entry-types, and all interaction relations is completely rebuild.

  When the cloned order-type maintains a rather complicated relation structure to other order-types, it is easier and safer (with regard to the correctness of the protocol developed) to clone the whole cluster of order-types instead of creating some deep clones and adapt relations by hand.

The different types of cloning order-types are illustrated in Figure 4.3 for a simple example (always cloning order-type O2). The shaded parts represent the new objects that result from the clone operations.

4.2.4 Protocol Validation

Before the edited PITOU protocol environment is transformed into code, it is possible to validate the correctness of the environment structure. A hand full of simple checks are sufficient to identify serious design flaws.

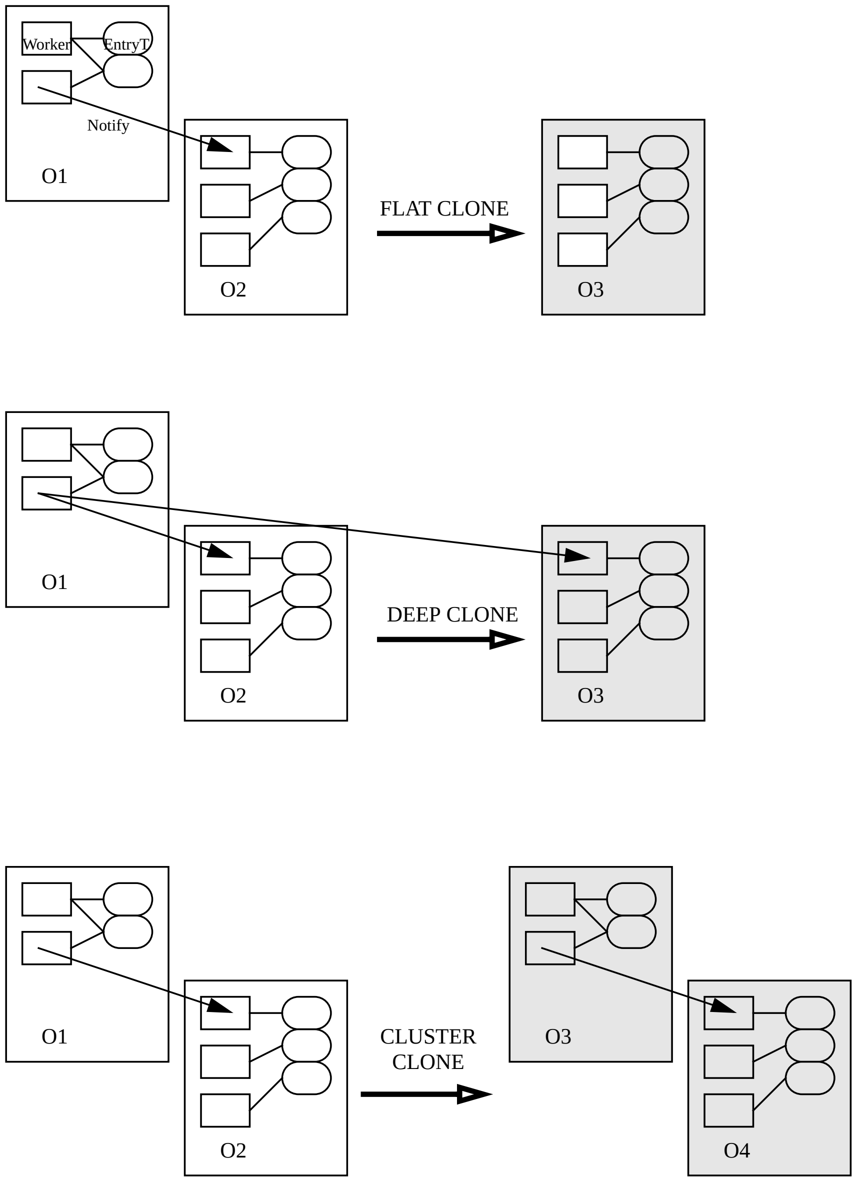


Figure 4.3: Cloning Order-Types

- has each worker been associated with the right number of entry-types of the correct type?
- has each notifier been connected with a notifiee?
- has each resource-subscriber been associated with a shared resource?
- are the identifiers of receptions and emissions unique and corresponding?
- is the FSM correct, i.e. is each state reachable, are all transitions unambiguous?

Of course, all entry-types, entry, worker, and out-of-band module objects must operate correctly. Additionally, in order to compose a reasonable protocol, the protocol composer must respect the semantics of the components he connects. Semantical correctness is not assured by PITOU.

4.2.5 Code Generator

Any protocol environment object can be transformed into Java code using our protocol generator (PG). The code generator reads all objects specified in the various registrar objects and gives them unique names. It identifies configurable properties of all protocol components by looking for methods that start with `get` and `set` (using the Java reflection classes similar to how JavaBeans tools do it). It finally produces code to rebuild all worker-entry relations for order-types, notification, shared-resource, and creation relations, the FSM, the defined mapping strategy, and the executor groups. The code generator is implemented in the class `CodeGenerator` of the `pitou.tools` package.

4.2.6 Outlook

There are still a number of open questions with regard to the protocol editor. First of all, the protocol editor should have a graphical interface to make it easy to use for anybody. Though the protocol editor lays the ground for a specialized tool to support the visual composition of protocols, the important part for the end-user – the graphical interface – is missing.

The protocol editor only supports merging and layering of order-types. We could also think of an operation to merge or layer whole clusters or even complete protocol environments. For that case, we would need to define rules that determine under which conditions which kind of order-types can be merged or layered.

The validation of protocol environments could be further developed. For example, a sender protocol could be validated against its receiver protocol to check if the order-types with the same identifier specify the same entry-types with corresponding properties.

Our structuring approach seems to hold a large potential for theoretical investigation since it can be formalized using sets with the corresponding operations such as union or intersection. We believe that important characteristics and structural patterns of protocols may be derived that allow insights about how to combine protocols and how to determine coarse-grain abstractions of protocol software.

4.3 Protocol Animator

The PITOU **Protocol Animator** (PA) is a tool that allows to visualize the behavior of any protocol that has been implemented with PITOU. The idea behind the PITOU-PA is to write information incurring during a protocol session into a log-file. After the session has finished, the file can be analyzed and the information visually represented.

The step-wise animation of protocols is not only useful for debugging purposes and better understanding the characteristics of a protocol. It may also be a valuable instrument for teaching and documenting protocol services and functions.

We have one overall requirement to be met: every protocol implemented within PITOU should be able to be visually animated without any modification. Modification should only concern the framework, never the workers and entry-type/entry classes. Hence, we want to animate protocols in a completely *generic* way.

4.3.1 Related Work

There have been a number of projects to visualize algorithms [49] and network dynamics [66]. Only few projects are concerned about the visualization of the internal structure of protocol software. The Technical University of Berlin had a project to extend the CHANNELS [22] protocol framework to allow for visualization. However, the animation of CHANNELS protocols is on a very coarse-grained level, and is not supported in a generic way without modification of the involved protocol modules. Both Estelle [6] and LOTOS [151] had been extended with features that allow to specify protocols that can be visualized. However, none of them is generic (the specifications are only meant for visualization, not for real execution). Additionally, visualization is restricted to the visualization of FSMs. The ProVis project [96] has the goal of visualizing protocols for teaching purposes using Java applets. It provides a toolkit to construct animation rather than visualizing real protocols.

4.3.2 Architecture

Our protocol animation tool performs three major tasks:

- **writing** protocol information into a log-file
- lexically analyzing and **parsing** the log-file and transform it into event objects readable by the part of the program that provides the graphical user interface
- visually **representing** possible events

Each of this tasks is performed by a seperate module. The architecture of the animation tool is depicted in Figure 4.4.

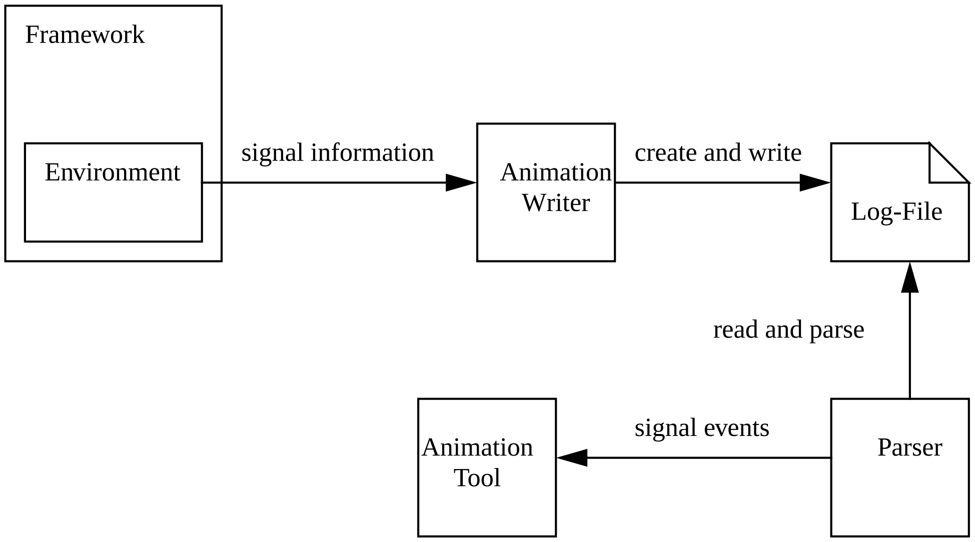


Figure 4.4: Architecture of the Animation Tool

Events

First of all, we must answer the question what information produced within the PITOU framework is worth of being represented visually. We thereby distinguish structural information (what are the components of an implemented protocol) from dynamic information (what happens during execution). Since we already talked exhaustively about the structure of PITOU protocols, we concentrate on the dynamics.

The principal events that can happen during a protocol session in PITOU are listed below.

- **acceptance event**: data from the application is accepted for a certain order-type
- **reception event**: data from the network is received and mapped to a certain order-type
- **new-order event**: a new internal order is requested to be created
- **delivery event**: an order is delivered
- **emission event**: an order is emitted
- **order parsing event**: an order is parsing (or better: delegates parsing to its entries)
- **entry parsing event**: an entry is parsing information

- **order executing event**: an order is executed
- **worker executing event**: a worker is called during execution
- **notification event**: one component notifies another one
- **resource changed event**: one component changes a resource
- **state change event**: the current state of the FSM changes

Exception handling adds the following events:

- received data could not be mapped to an order
- initialization of an order didn't work
- execution is interrupted

Besides those events, it is also useful to have some information always accessible like the value of a current entry after parsing and during execution or internal variables of workers during execution.

Producing a log-file

At various points of the framework, we must modify code to allow for producing a log-file. For example, *acceptance events* and *emission events* can be produced in the `accept()` and `emit()` methods, respectively, of `Environment`. To log *order/entry parsing events* the method `InternalOrder.fill()` must be modified. To log *order/worker execution events*, the method `InternalOrder.execute()` must be modified.

Ideally, we should re-implement the whole framework for the purpose of animation. That way, execution of real protocols is completely separated from execution of protocols for the purpose of animation. However, since the modifications needed are negligible and we are more interested in a multi-purpose running prototype than to have two different, but optimized tools with industrial strength, we decided to integrate log-file production into the code and allow a flag to be set that indicates, if the log-file code is used or ignored.

We define a class `AnimationWriter`, which has no `public` constructor, and provides a `static` method `AnimationWriter giveGlobalWriter()`. There exists only a single object of `AnimationWriter`, which is created when the `giveGlobalWriter()` method is called the first time (all further calls return the same object). That way, the object is made globally accessible from anywhere. This solution corresponds with the *Singleton* design pattern [59]. An example for changing code to produce a log-file is shown in Figure 4.5.

```
public void execute() throws OrderException {
        ...

        for (int i = 0; i < nrWorkers; i++) {

                //start of ANIMATION code
                if (AnimationSpec.anim_switchedOn) {
                        AnimationSpec.getAnimationWrite().executedWorker(myWorkList[i], myOrderType.giveEnvironme
                }
                //end of ANIMATION code

                if (!myWorkList[i].call(myWorkParam[i])) {

                        // start of ANIMATION  code
                        if (AnimationSpec.anim_switchedOn) {
                                AnimationSpec.getAnimationWrite().orderException(this +" stopped at worker " + i,
                        }
                        // end of ANIMATION code

                        throw new OrderException(this +" stopped at worker " + i);
                }
        }
        ...
}
```

Figure 4.5: Example Code to Write Animation Information

Once a protocol session is finished and a log-file is produced, the PITOU framework is no longer involved. We run a lexical analyzer and parser, which reads the information from the log-file, transforms it into event-objects, and notifies the animator. The language that represents the events is very simple. Each event is mapped to one keyword followed by additional information, which are separated by line-feeds. The parser just needs to read line by line, check for the syntax, and produce event objects that are used to notify the animation tool.

The Animation Windows

The animation tool provides different views on an animated protocol. Each view is represented by an own window.

- The **initial window** opens when the animator tools starts. It allows to load a log-file, and is supposed to support features like printing or showing statistics (not yet implemented).
- The **environment window** shows environments comprising all order-types and out-of-band modules, and the current state of an environment. Different categories of order-types have slightly different symbols: triangles are used to indicate output or input direction, and application or network interface. It is possible to represent more than one environment in this window at the same time. The environment window is depicted in Figure 4.6.

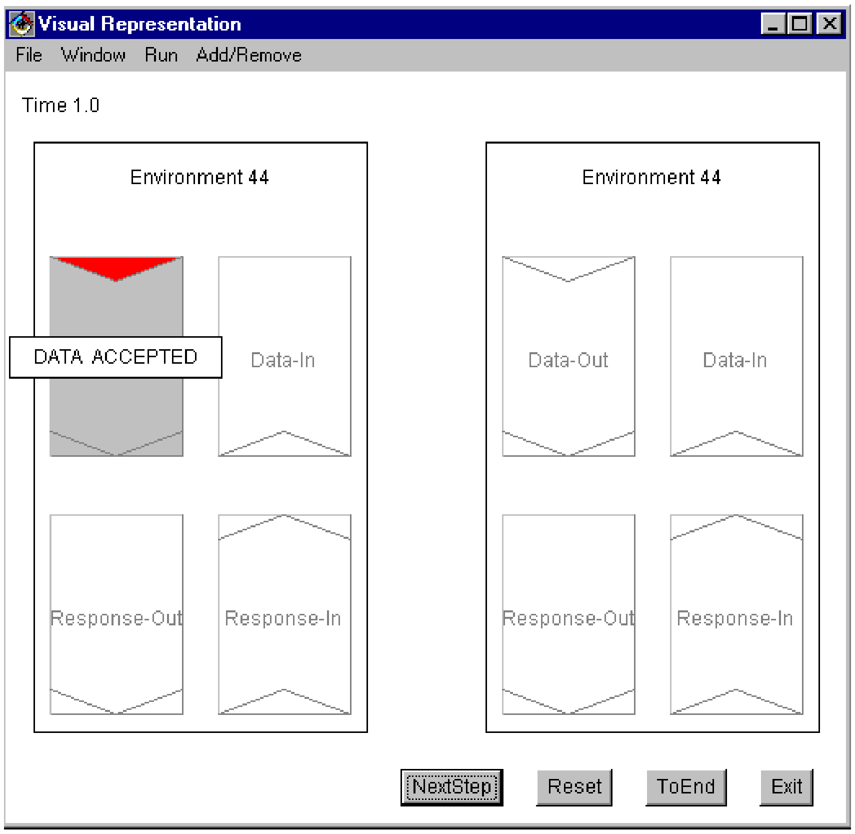
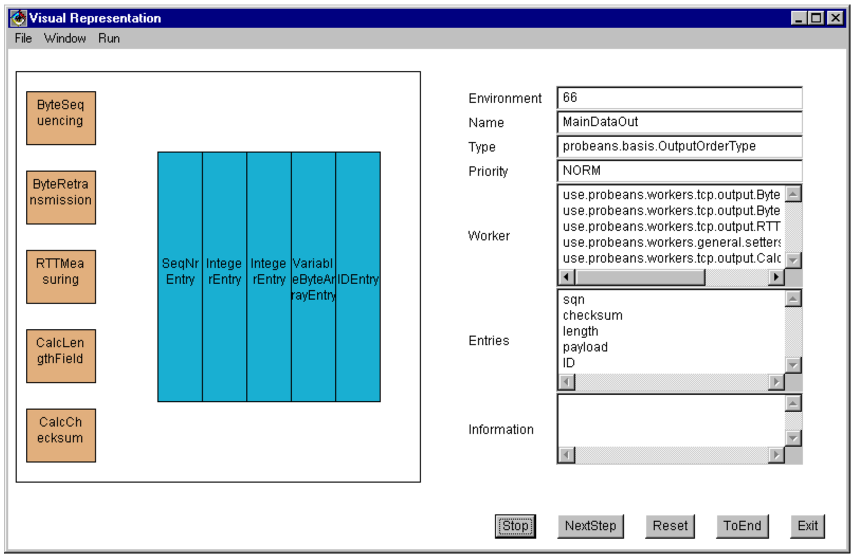


Figure 4.6: Animation Tool: Environment Window

- The **order window** opens when the user clicks on an order-type symbol of the environment window. It shows all workers, their entry-types, and static information about an order-type (the environment it belongs to, its name and type, its priority). The order window is depicted in Figure 4.7.

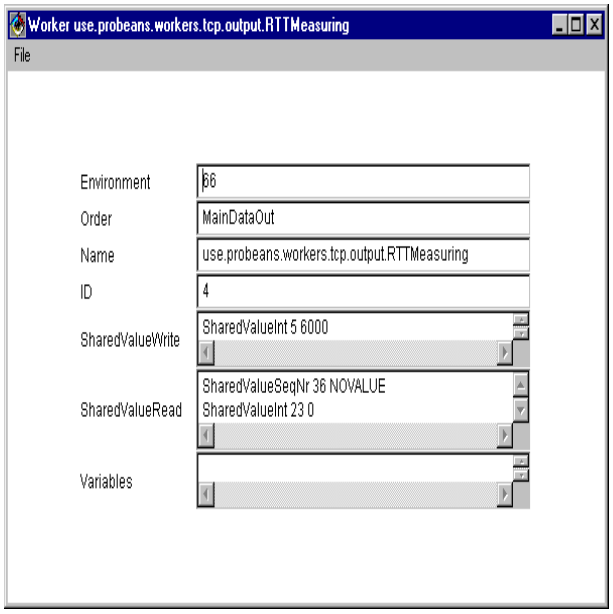
- The **worker window** opens when the user clicks on a worker symbol in the order window. It shows additional information about a worker (its name, the environment and order-type it belongs to, the resources they publish or subscribe, and internal variables). The worker window is depicted in Figure 4.8.

- The **entry window** opens when the user clicks on an entry-type symbol in the order window. It shows additional information about an entry-type (environment and order-type it belongs to, its type and name, visibility and initializability, etc.) and the current value of the corresponding dynamic entry object. The entry window is depicted in Figure 4.9.



Environment 66
Name MainDataOut
Type probeans.basis.OutputOrderType
Priority NORM
Worker
use.probeans.workers.tcp.output.Byte
use.probeans.workers.tcp.output.Byte
use.probeans.workers.tcp.output.RTT
use.probeans.workers.general.setters
use.probeans.workers.tcp.output.Calc
Entries
sqn
checksum
length
payload
ID
Information

Figure 4.7: Animation Tool: Order Window

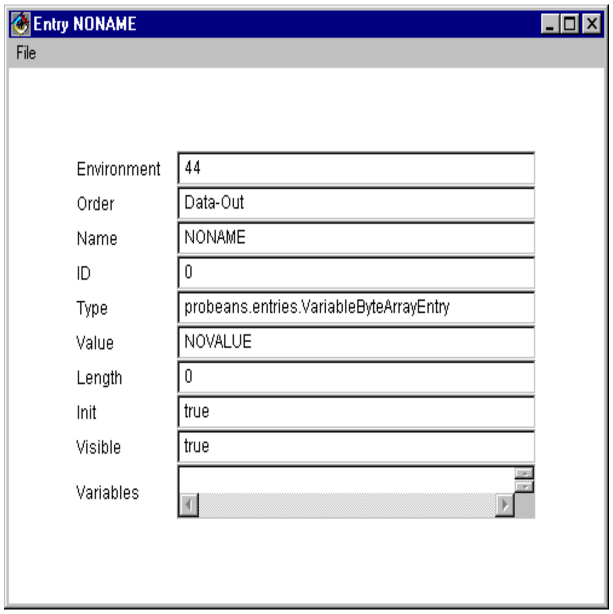


Environment 66
Order MainDataOut
Name use.probeans.workers.tcp.output.RTTMeasuring
ID 4
SharedValueWrite SharedValueInt 5 6000
SharedValueRead SharedValueSeqNr 36 NOVALUE
SharedValueInt 23 0
Variables

Figure 4.8: Animation Tool: Worker Window

How the Events are animated

Environment window: the triggering of an *acceptance event* is represented by giving the triangle of the related acceptance-type a red color. Additionally, a little message pops up. *Reception events* are treated similarly. The creation of a new order (*new-order event*) is represented by a labeled arrow (see Figure 4.10). The arrow starts from the order-type containing the worker



Environment	44
Order	Data-Out
Name	NONAME
ID	0
Type	probeans.entries.VariableByteArrayEntry
Value	NOVALUE
Length	0
Init	true
Visible	true
Variables	

Figure 4.9: Animation Tool: Entry Window

from which the creation was requested, and ends at the order-type supposed to create a new order.

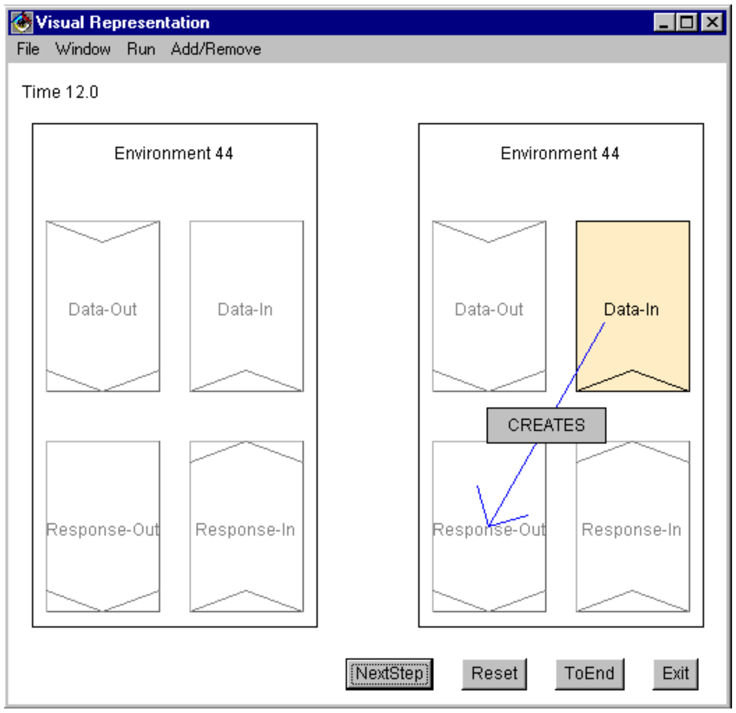


Figure 4.10: Animation Tool: Order Creation

When an order parses data (*parsing order event*), the color of the concerned order-type changes to grey and a message pops up. During execution of an order (*executing order event*), the concerned order-type changes to yellow. When emission or delivery orders terminate execution, their triangle changes to red to indicate that the data is delivered. Clicks to red triangles allows to pop up a little window that shows the value of the data to be accepted, received, delivered, or emitted. *Notification events* and *resource changed events* are represented in the environment window by a labeled arrow between the order-type symbols that are involved. *State changed events* simply lead to a change of the name of the current state.

For all kind of exceptions, the environment window shows a little message indicating the problem.

Order window: During the parsing of an order, the order window highlights the entry-type that is currently parsing (*entry parsing event*). During the execution of an order, the order window highlights the worker that is currently executed (*worker executing event*). Furthermore, it pops up a little message when workers are involved in notification or order creation, or when a worker accesses a shared resource.

4.3.3 Discussion and Outlook

The implementation of the PITOU-PA has been done in a master thesis project [146]. We presented a rather high-level overview of the tool to avoid details like the language representation, the involved classes, and implementation issues of parser and animator. The PITOU-PA is just a prototype and much less robust and mature than the other parts of PITOU. However, the implementation shows that the structure of PITOU protocols is powerful and general enough to allow for generic and fine-grained visualization of real protocols.

We can think of a number of animator features that would be useful to be integrated. Animation could be explicitly supported by integrating a mechanism that allows to add worker specific events. The timer-pool could provide information to animate timeouts. Instead of taking the detour of writing information in a file, analyzing, and sending it to the animator, the information could directly be sent to the animator and visually represented (which would not be a big problem, since all tools are independent from each other).

4.4 Summary

In this chapter, we presented a number of tools to support the implementation, debugging and testing of protocols within the PITOU framework. The first tool is the PITOU *protocol simulator*. We presented the main classes and showed how to integrate simulation facilities into the framework to run any PITOU protocol software in a virtual environment. Our simulator makes it possible to simulate different thread models and to associate virtual durations to order-execution. It operates on a much finer granularity than common network simulators. One important feature is that simulation is possible without any support or code modification of the protocol component code.

The protocol editor and code generator are tools for support and modification of protocol implementations on a coarser level of structure. The editor allows to clone, layer, and merge order-types, to identify and modify coherent parts of protocols (cluster), and thus reuse complete order-types and clusters across different protocols. We also described how the composed protocols can be easily validated for correctness. The protocol *code generator* is able to generate Java code out of the specified composition.

The last tool presented is a tool to animate the structure of a protocol and its behavior during runtime. In contrary to teaching tools, which allow to build animations of network or protocol behavior, our tool is completely generic and allows to animate any protocol developed with PITOU without any modification.

It is important to note that our structuring approach, and the decoupling of framework core classes and protocol code are the key to provide the flexibility to use real protocols also within simulations or even animations. The simulation and animation facilities are thus a spin-off of the flexible model and design.

Chapter 5

Case Studies

This chapter studies concrete examples of using PITOU for protocol implementation. The goal of this section is to evaluate the expressiveness of our proposed structure with regard to the potentially ugly details of real protocols. We decided to implement RTP/RTCP (Real-Time Transport Protocol/Real-Time Control Protocol) [128], since it defines sophisticated message syntaxes. We choose TCP (Transmission Control Protocol) [109] as implementation target, since it provides a number of sophisticated mechanisms, which make TCP an honest benchmark for any protocol architecture.

5.1 RTP/RTCP

We shortly summarize RTP/RTCP and then show how a simple RTP snoop application (an application that reads all messages exchanged during a session) for the MBone has been structured and built within PITOU.

5.1.1 RTP/RTCP Overview

RTP/RTCP are defined as end-to-end transport and control protocols for real-time applications, e.g. applications transmitting audio, video, or simulation data. Real-time data and transmission specific information like sequence numbers, time-stamps, or source identifiers are transmitted in RTP messages. The RTCP message headers define information to monitor and control the quality of service within an on-going session (e.g. loss, jitter). There are four different message types used in RTCP:

- **Sender Reports** (SR) carry statistics from participants that are active senders
- **Receiver Reports** (RR) carry feedback from receiving participants

- **Source Description** (SDES) carry information about a participant
- **Bye Messages** (BYE) indicates the end of a participation

RTP/RTCP are meant to be built on top of either uni-cast or multi-cast network services. While RTP/RTCP are designed to work independently of the underlying transport and network layers, they are typically used on top of UDP [110] to make use of UDP de-multiplexing and check-summing services.

RTP and RTCP only specify header formats, they do not specify concrete services or algorithms. An application profile specification allows to define payload formats and how to map the payload types to payload formats. In the MBone [51], RTP/RTCP are assigned with two consecutive UDP ports, one for RTP messages, one for RTCP messages.

All RTCP packets can be sent within one single packet called **compound packet**, which defines its own syntax.

5.1.2 Applying Data Path Reification and Classification

Applying *Data Path Reification* is trivial for RTP/RTCP since we can map each message to one data path. Since the environment we compose is only meant to snoop RTP/RTCP sessions on the MBone, all order-types are of type input, i.e. all orders interface both with network and application. We have hence the following order-types:

- *RTP-in* for processing incoming RTP messages
- *SDES-in* for processing incoming SDES messages
- *SR-in* for processing incoming SR messages
- *RR-in* for processing incoming RR messages
- *BYE-in* for processing incoming BYE messages

During the instantiation of a RTP/RTCP environment obtains two `SAP` objects (refered to as SAP 0 and SAP 1) of type `MC_SAP`. *RTP-in* is configured to use SAP 0, while all other order-types are configured to use SAP 1.[1].

Since on SAP 0 only RTP messages can arrive, we register a mapping-strategy object of type `TakeFirst` to be used for SAP 0, which maps data to the first (and only) reception order of the environment. For SAP 1, things are more complicated, since four different message types

[1]A RTP/RTCP session is associated with two UDP ports. One is used for RTP messages, the other for RTCP messages.

can arrive. We therefore implement a class `RTCPMappingStrategy`, which looks into the header to identify the packet type (Packet Type: SR=200, RR=201, SDES=202, BYE=203), and register it with SAP 1.

5.1.3 Applying Data Path Partitioning

Since our protocol is only for snooping purpose, no functionality needs to be implemented, hence no workers are used. Our order-types only comprise a set of entry-type objects.

The RTP message format consists of the following header fields to be represented by entry-type objects (see also Figure 5.1).

- The **version** field consists of 2 bits.
- The **padding** field consists of 1 bit to indicate if the packet contains padding bytes (e.g. used for encryption algorithms that require fixed block sizes).
- The **extension** field consists of 1 bit to indicate if the header extended by a user defined format.
- The **CSRC (contributing source) count** field consists of 4 bits and contains the number of CSRC identifiers in this message.
- The **marker** field consists of 1 bit to indicate significant events defined by a profile.
- The **payload type** field consists of 7 bits and specifies the format of the payload (either default or user defined types are possible).
- The **sequence number** field consists of 2 bytes and should be incremented for each RTP message sent.
- The **time-stamp** field consists of 4 bytes and reflects the sampling instant of the first byte in a RTP message.
- The **SSRC (synchronization source) identifier** field consists of 4 bytes and is destined to identify the source that synchronized all contributing sources specified in this message.
- The **CSRC (contributing source) list** consists of 0 to 15 items (4 bytes each) representing contributing sources.

The first six header fields (version, padding, extension, CSRC count, marker, payload type) are represented by a bit-block entry-type composed of an integer for version, booleans for padding and extension, an integer for CSRC count, a boolean for marker, and an integer for payload type.

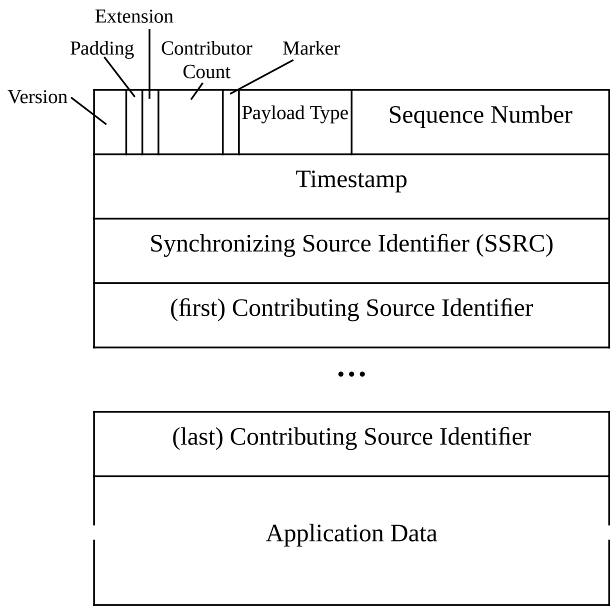

Extension
Padding
Contributor Count
Marker
Version
Payload Type
Sequence Number
Timestamp
Synchronizing Source Identifier (SSRC)
(first) Contributing Source Identifier
...
(last) Contributing Source Identifier
Application Data

Figure 5.1: RTP Header Format

The sequence number field is represented by a sequence number entry type, the time-stamp by an integer, and the SSRC identifier by another integer. The CSRC list field is represented by an array of integers. In Table 5.1 you can find the exact class types.

RTP Header Field	**PITOU Entry-Type**
flags block (version, .., payload type)	`BitBlockEntryType`
version	`IntegerEntryType`
padding	`BooleanEntryType`
extension	`BooleanEntryType`
CSRC count	`IntegerEntryType`
marker	`BooleanEntryType`
payload type	`IntegerEntryType`
sequence number	`SeqNrEntryType`
time stamp	`IntegerEntryType`
SSRC	`IntegerEntryType`
CSRC block	`ArrayEntryType`
CSRC field	`IntegerEntryType`

Table 5.1: Mapping from RTP Header Formats to PITOU Entry Types

Since the types of RTCP message fields are very similar to those of RTP, we don’t show the mapping from header fields to entry-types for each RTCP message (see Figure 5.2). Sender and receiver reports consist of a variable number of information blocks. These are mapped to an `ArrayEntryType` object, which maintains an `BlockEntryType` object, which consists of a number of simple `EntryType` objects. These report blocks are examples for the combination of arrays and composites to compose sophisticated structures out of entry-types.

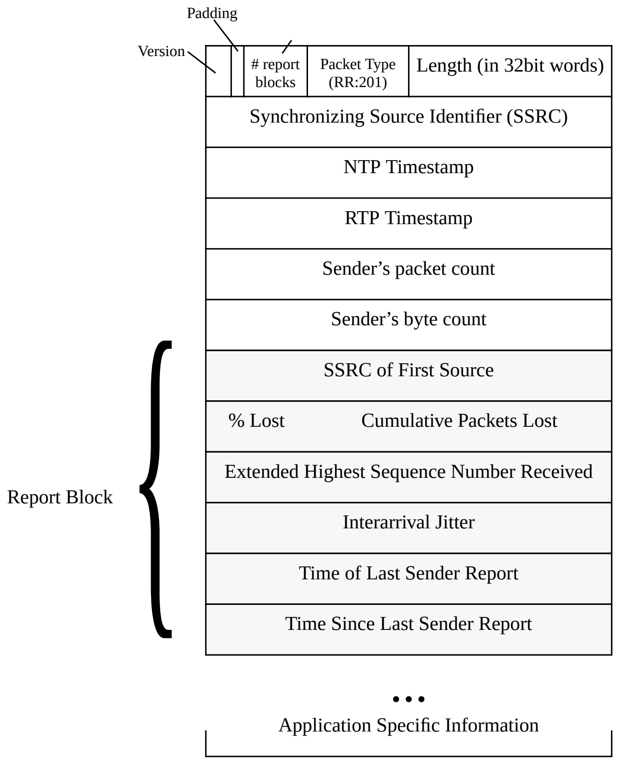


Figure 5.2: RTCP Receiver Report Header Format

5.1.4 Conclusion

The implementation of our RTP snoop protocol showed us three things. First, the various header fields can be mapped to few entry-types like `IntegerEntryType`, `BooleanEntryType`, `SeqNrEntryType`. Configuring the byte-size of an entry-type is in most cases the only form of adaption needed. Second, the composite and array entry-types play an important role in composing RTP/RTCP headers. Without composite and array-entry-types, the implementation would have been extremely cumbersome and expensive (as a first implementation showed us when composite and array entry-types had not been supported yet). Third, we need a RTP specific mapping strategy, which not only identifies packet types and maps them to order-types, but also implements the mechanism to de-construct compound packets.

5.2 TCP

We will first give an overview of TCP's functionality, before we show how we map TCP services to data-paths and how we divide the data-paths in workers and entries. We will report on the problems encountered, and finally discuss benefits and drawbacks of applying our structuring approach to TCP. Our implementation is limited to the data transfer phase of TCP.

5.2.1 TCP Overview

TCP is a transport protocol that supports the bi-directional exchange of a contiguous **stream of bytes** between two hosts. The stream is fragmented and sent in TCP *segments*. A TCP implementation is free to packetize the data streams to optimize performance.

TCP implements a three-way hand-shake protocol to establish a **connection** between two hosts before the data transfer can start. When the data transfer phase is completed, the connection is closed.

TCP ensures **reliability** by protecting the byte stream against corruption, loss, duplication, and misordering. A checksum allows to detect corruption due to bit-errors on the transmission medium. A sequence number allows the receiver to filter duplicates and to re-order segments. The sender buffers all data before sending and waits until the receiver has acknowledged (ACK) the reception. If an acknowledgment is not received within a time-out interval, the data is retransmitted.

The timer-interval is based on measurements and estimations of the round-trip-time and has exponential back-off in case of retransmission. The algorithm implemented today [72] is not the same as proposed in the original specification. A measurement captures the time between sending data and receiving an acknowledgment for that data. The result of a measurement is thrown away when a retransmission is required, since it is not possible to know if the ACK received is for the original or for the retransmitted packet. This problem is known as retransmission ambiguity problem [82].

TCP follows an cumulative acknowledgement strategy. The acknowledgment number represents an acknowledgment for all data assigned a sequence number smaller than this value. In order to reduce the number of segments sent, acknowledgments are not sent immediately, but delayed for a short time interval (normally 200ms). During this time the application process can read data from the receive buffer to avoid changing of the advertised window (see below), and –in case that communication is bi-directional– application data can be sent together with the acknowledgment (piggy-backing).

TCP features window-based **flow control**. The receiver signals the sender the remaining space in the receive buffer called *advertised window*. The receive buffer contains the bytes that are already acknowledged and delivered to the application, but not yet read by the application process. The sender is not allowed to send more bytes than the receiver advertised (up from the last byte that has been acknowledged).

When the advertised window shrinks to zero, the sender is not allowed to send anything and stops. It may happen that the next segment of the receiver containing the new non-zero window-value is lost. In that case, the sender stops and the receiver waits. To break this deadlock, the sender should set a timer called *persist-timer* when the advertised window value is zero. When the timer expires he sends a packet containing one byte of data to provoke the receiver to send a segment (probe-packet) that may contain a non-zero window value.

Another phenomena with regard to window-based flow control is the *silly-window syndrome*, which leads to small amounts of data exchanged instead of full segments and hence inefficient usage of bandwidth. It happens when the receiver advertises small windows and the sender immediately sends data upon receiving the permission of sending small data. Following the philosophy of TCP ”be liberal in what you accept, be conservative in what you do” sender and receiver as well should avoid this situation. The receiver should not advertise small windows by waiting until the window can be increased by the size of a full segment or by the half of the total receiver buffer space. The sender should send data only when he can send a full segment or half of the maximum window size ever advertised by the receiver (only needed for older primitive hosts), or when the send buffer contains no further data and no data is outstanding, i.e. no acknowledgements are expected.

Due to severe Internet **congestion** collapses in the 1980s, the implementation of TCP was modified to include the slow-start and congestion-avoidance algorithms by Van Jacobson [72]. This mechanism introduces a congestion window representing the assumed number of packets the network can accept and takes lost packets as an indication for congestion. The flow-control is restricted to send the minimum of congestion window and advertised window. The fast-retransmit and fast-recovery algorithms [74] are optimizations of this idea and are implemented in any standard TCP software.

In order to avoid sending packets with very little data and high header overhead, Nagle proposed to block data until a segment with maximum data payload can be sent, or all data already sent is acknowledged [103]. This algorithm is useful especially for interactive transfer of small data. Since this algorithm may cause performance problems for certain scenarios, it can be disabled.

5.2.2 Applying Data Path Reification and Classification

The first step towards a vertically structured TCP implementation is to identify and isolate the various data paths in TCP. We decided to leave out connection establishing and termination in our description (including the different states a TCP session passes through), and concentrate on the data transfer part of our implementation, which we belief is the more interesting and challenging part and better illustrates the main difficulties and benefits of a vertical structure than the rather simple connection part.

Any piece of data obtained by the application process is packed in a segment, gets a sequence number, is buffered in the retransmission buffer, is checksummed and then sent. Additionally, each new segment triggers a measurement of the round-trip-time. All this can be done within one thread and will be packed into one order that we call **data-output-order**. This order is of type *emission* since it finishes by emitting a packet to the network.

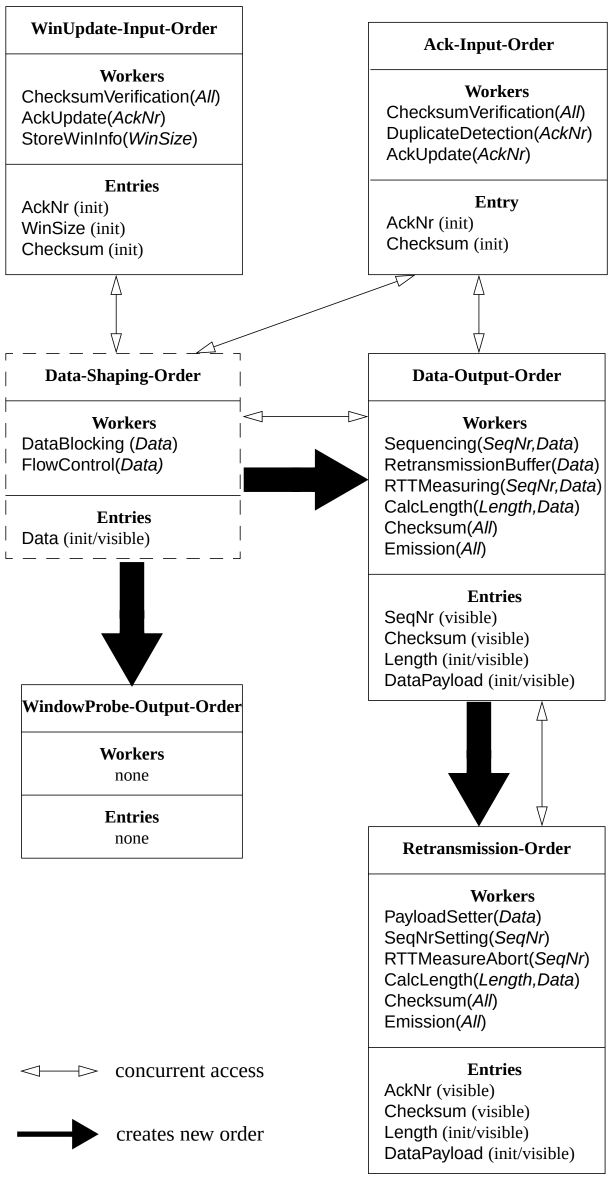


Figure 5.3: Orders of the sender entity in a Thread-Oriented TCP

Nevertheless, before preparing a TCP segment, the advertised window and the congestion window must be checked if sending of data is allowed. Additionally, the sender silly-window avoidance must be performed, small data segments should be blocked, and large data blocks must be fragmented to make it fit into a segment of limited size. All these functions result in producing a piece of data that is ready to be processed by the *data-output-order*. We collect these functions – data blocking/fragmenting and flow-control – in one order that we refer to as **data-shaping-order**. This order is of type *acceptance* since it is triggered by the application process.

The data-out-order produces a packet, which triggers an order in input direction at the receiver side. At the receiver the incoming data is first de-multiplexed and then triggers the execution of an order called **data-input-order**. First, the checksum is verified and the data possibly dropped if the checksum fails. When the checksum is correct, the sequence number is checked to know if duplicate data must be dropped or data must be re-ordered. The valid data is delivered to the receive buffer of the receiver application process.

The reception of data must be acknowledged by the receiver. Acknowledging is represented in our architecture by one order for each direction called **ack-output-order** and **ack-input-order**. The *ack-output-order* is created by the *data-input-order* and just adds the actual acknowledgement number (sequence number of last correctly received byte + 1) to the outgoing segment and calculates a checksum. The *ack-input-order* verifies the checksum, checks for duplicates, and calculates the number of bytes that are newly acknowledged. Depending on this value, the Nagle algorithm is asked to free possibly blocked data, a round-trip-time measurement is completed, the retransmission buffer freed, the congestion window increased, and the information about the first non-acknowledged data is updated.

When an acknowledgement packet is lost on the network, the retransmission timer expires. This causes the retransmission of buffered data. A retransmission is also modeled as an own order referred to as **retransmission-order**. Data of a *retransmission-order* will get a sequence number (using the value of the first non-acknowledged sequence number), is checksummed, and sent.

Still missing is the flow-control handling of the advertised window. The advertised-window value changes only when a receiver process does not read data fast enough from the receive buffer. In TCP the advertised window is sent along with the data acknowledgements. Since we want to separate error-control handling from flow-control handling, we define an own order for window-handling. In the **window-update-output-order**, the flow-control information is written to a TCP segment (consisting of the sequence-number of the last delivered byte+1 and the space in the receive buffer), a checksum is calculated, and the data is sent. In the **window-update-input-order**, the flow-control information is used to update the sliding-window when the checksum verification on the received data was positive.

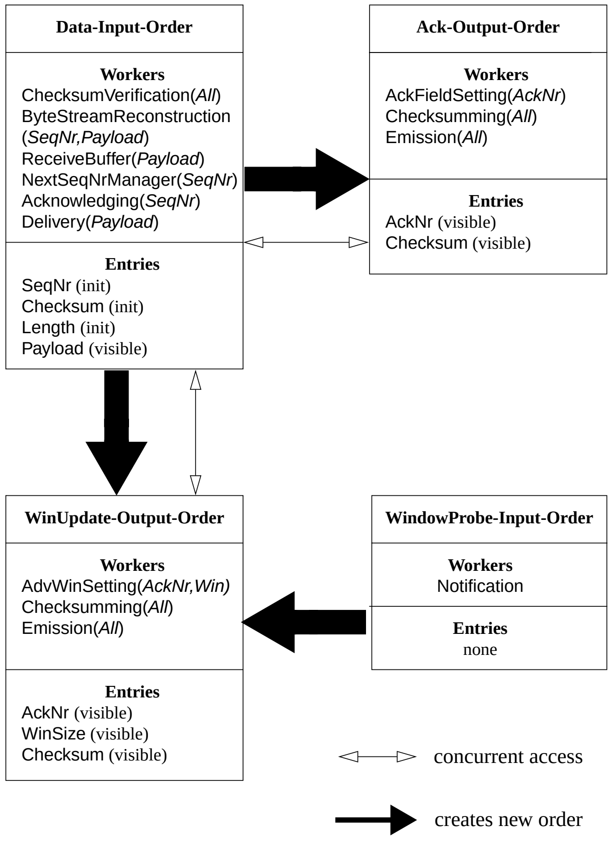


Figure 5.4: Orders of the receiver entity in a Thread-Oriented TCP

In the case that the flow-control information is lost and the persist timer expired, TCP creates a window-probe segment. The corresponding orders are the **window-probe-output-order** and **window-probe-input-order**. An outgoing window-probe order contains no functionality. It is only sent to provoke the emission of a *window-update-output-order* at the receiver's side.

In Figures 5.3 and 5.4, we can see an overview of the orders we used to structure TCP and the relations between these orders. We also see what information is used as parameters to be processed by the various functions. *New-order-relations* are marked by the bold arrows. *Notification relations* and *shared resource relations* are represented by the thin double-sided arrows). A more detailed description of the various relations can be found in Table 5.2 (notification relations of the sender side[2]), in Table 5.3 (shared-resource relations of the sender), and Table 5.4 (shared-resource relations of the receiver).

[2]There are no notification relations on the receiver side.

Notifier	Notifiee	Description
win-update input order (execution done)	FlowControl worker	sliding window is updated
ack input order (execution done)	DataBlocking worker	blocked data can be freed
ack input order (execution done)	RTTMeasuring worker	start new measurement
ack input order (execution done)	ByteRetransmission worker	buffered data can be freed
ack input order (execution done)	FlowControl/CC worker	update congestion window
ByteRetransmission worker (timeout)	FlowControl/CC worker	loss indication
StopMeasuring worker (data passed)	RTTMeasuring worker	abort measuring
DuplicateDetection worker (duplicate)	ByteRetransmission worker	retransmission request

Table 5.2: TCP Notification Relations (Sender)

5.2.3 Applying Data Path Partitioning

Data-Shaping Order: The `DataBlocking` worker implements Nagle's algorithm to avoid sending packets with very little data. It takes a byte array from the application and buffers it when the data that can be sent (buffered plus new data) is smaller than the maximum segment size. As soon as an acknowledgment arrives or a full segment can be sent, the buffered data is freed for further processing. The `FlowControl` worker is responsible that no more data is sent than the receiver application and the network can handle. It gets its information from two sources. The first information is the *advertised receiver window* obtained from the `AdvWinUpdate` worker, which is used to calculate the *usable window* of a sliding window. The second information comes from the `CongestionControl` module[3], which implements the slow-start, congestion-avoidance, fast-retransmit and fast-recovery [134] algorithms to adapt a congestion window in response to detected loss of packets and duplicate acknowledgments. The flow-control worker sends only the minimum number of packets allowed by congestion and sliding window. It performs sender silly-window avoidance by sending only full segments and managing a timer, which may trigger window-probe packets to ask the receiver for a window-update.

Data-Output Order: The `Sequencing` worker assigns a sequence number to the first byte of each byte array processed. It starts with a random sequence number generated when the connection was established. The `RetransmissionBuffer` worker buffers each byte until an acknowledgment arrives. It implements a timer with a time-out interval that is set to the value calculated by the `RTTMeasuring` worker. When the timer expires, the first segment (i.e. a configurable number of bytes) is re-sent to the receiver. The `RTTMeasuring` worker measures the time between sending a data packet and arrival of its acknowledgment. Measuring is aborted by retransmission-orders (see below) to avoid ACK ambiguity (Karn's algorithm [82]).

[3]The congestion-control-module is an out-of-band module; it is not listed in Figure 5.3

The measures are used to estimate variance and average of the round-trip-time, which are used to determine the time-out value for *retransmission timer* and *persist timer*. The `CalcLength` worker writes the length of the payload to the segment. The `Checksum` worker uses the complete header and the data payload to calculate the Internet checksum (see [28]).

Retransmission Order: Retransmitted data must get a new sequence number since the segment borders may be changed within message processing in TCP. Therefore the sequence number information about the first not yet acknowledged data byte is kept in a shared resource maintained by the `AckUpdate` worker (see ack-input order). This resource is accessed by the `SeqNrSetting` worker, which just writes this information into the outgoing message. A `RTTMeasureAbort` worker notifies the `RTTMeasuring` worker to abort RTT measuring for the concerned sequence numbers. Finally, a `Checksum` worker protects against corruption.

Acknowledge-Output Order: The `AckFieldSetting` worker just writes the sequence number of the next data byte expected (which equals the last successfully received data plus one) into the outgoing message. It gets this information by reading from a shared resource, which is maintained by the `NextSeqNrStateManager` worker (see data-input order). A `Checksum` worker protects against corruption.

Window-Update-Output Order: The `AdvWinSetting` worker just writes the advertisement information obtained from the `ReceiveBuffer` worker into the outgoing message. This information comprises the sequence number of the start byte of the sliding window (next expected) and the space of the receive buffer of the application. A `Checksum` worker protects against corruption.

Shared Resource	**Provider**	**Subscriber**
advertised window/ack	AdvWinUpdate worker	FlowControl worker
last ack	AckUpdate worker	RTTMeasuring, SeqNrSetting worker
estimated RTT	RTTMeasuring worker	ByteRetransmission, FlowControl worker
length of retransmitted packet	Length entry	RTTMeasuring worker

Table 5.3: TCP Shared Resource Relations between Workers (Sender)

Data-Input Order: A `ChecksumVerification` worker compares the value of the checksum field calculated by the peer-TCP with an own calculation, and throws the data away, i.e. stops the execution of this order immediately, when the result is negative. A `ByteStreamReconstruction` worker brings incoming segments into the right order and drops data already delivered by managing a buffer that contains out-of-order data temporarily. A `ReceiveBuffer` worker maintains information about the receive buffer, i.e. the buffer from which the application reads delivered data, used by the `AdvWindSetting` worker. Receiver silly-window avoidance and delaying window-updates by using a timer is also performed by the `ReceiverBuffer` worker. A `NextSeqNrManager` worker keeps the sequence number of the last byte delivered (written to the receive buffer) used by the `AckFieldSetting` worker as acknowledgment number. An `Acknowledging` worker triggers ack-output-orders. ACKs for duplicate data are requested immediately, other acknowledgments are delayed to cumulate ACKs.

Acknowledgment-Input Order: A `ChecksumVerification` worker drops corrupted data. A `DuplicateDetection` worker checks if the acknowledgment number is duplicate and notifies the `CongestionControl` module to perform the fast-retransmit algorithm when a configurable number of subsequent duplicate ACKs have occurred. A `AckUpdate` worker calculates the number of acknowledged bytes and use it to free bytes of the buffer in the `RetransmissionBuffer` worker. `AckUpdate` also notifies the `DataBlocking` worker, the `CongestionModule`, and the `RTTMeasuring` worker about the arrival of an acknowledgment.

Window-Update-Input Order: A `ChecksumVerification` worker drops corrupted data. A `AdvWinUpdate` worker calls the `FlowControl` worker to update its state with the new information gathered from the window-update (start of sliding-window, advertised window).

Window-Probes: The window-probe-input order consists of a worker, which triggers the sending of a window-update each time a window-probe input order is executed (`Notification` worker). In input direction, no functionality is defined.

Shared Resource	**Provider**	**Subscriber**
next expected sequence number	NextSeqNrManager worker	AckFieldSetting worker, ByteStreamReconstruction worker, and Acknowledging worker
window size/ack advertisement	ReceiveBuffer worker	AdvWinSetting worker

Table 5.4: TCP Shared Resource Relations between Workers (Receiver)

5.2.4 Mapping

In order to make our implementation compatible, we must implement a TCP specific mapping strategy, which i) transforms all messages produced by the various emission orders into packets that respect the TCP format, and ii) maps incoming TCP segments to one or more order-types.

Due to the need for backward compatibility, new mechanisms have been integrated in TCP implementations over the years, while the message format remained the same. As a consequence, there exist now dependencies among header fields, which make it impossible to map a TCP message to a data-path without accessing internal information from various modules and thus violating the principle of information hiding. To illustrate this problem, we have a look at the code of a standard TCP implementation described by Stevens [135] (see Figure 5.5) that concerns the identification of a pure acknowledgement (in contrary to a *duplicate* ack). The identification of a *pure* acknowledgement involves variables like the flow-control window, the congestion-control window, and the payload field of a message. All these variables must be checked to identify pure acknowledgments (the identification of duplicate ACKs requires another couple of checks) although their relation to error-control is not obvious.

The goal of our TCP service implementation was to demonstrate that a sophisticated protocol like TCP can be structured and implemented in a modular fashion. We therefore did not implement a TCP specific mapping strategy to make our implementation compatible, which would at the same time violate modularity. Instead we decided to emphasize the beauty of a structured TCP equivalent protocol, which allows to be tailored to application requirements by removing, adding, and configuring order-types and workers.

```
001 if (ti->ti_len == 0) {
002    if (SEQ_GT(ti->ti_ack, tp->snd_una) &&
003        SEQ_LEQ(ti->ti_ack, tp->snd_max) &&
004         tp->snd_cwnd >= tp->snd_wnd {
005        ... //process pure ACK
```

A TCP segment is a pure ACK [135] if

1. the segment contains no data,i.e. `ti_len` is 0 (001), and
2. the acknowledgement field in the segment `ti_ack` is greater than the largest unacknowledged sequence number `snd una` (002), and
3. the acknowledgement field in the segment `ti_ack` is less or equal to the maximum sequence number sent `snd max` (003), and
4. the congestion window `snd_cwnd` is greater than or equal to the current send window (`snd wnd`). This test is true only if the window is fully open, that is, the connection is not in the middle of slow start or congestion avoidance (004).

Figure 5.5: Identification of a pure ACK (Stevens, 1995)

5.2.5 Conclusion

The structuring and implementation of a TCP like transport protocol within PITOU gave us a number of valuable insights. First of all, the implementation was successful. The fact that our approach allows to implement all services of such a complex general purpose protocol as TCP confirms its expressiveness and usefulness. When PITOU is able to cope with the complexity of TCP, it should more than that be able to facilitate the implementation of application-tailored protocols that are supposed to be much simpler than TCP.

Second, the syntax of the header-format has a decisive impact on the possibility to modularize and automate protocol implementation. TCP has been extended over years to integrate mechanisms without adapting the header format. Due to the force of staying compatible, information for which no header field exists must be derived from other fields and local state. These dependencies among header fields lead consequently to dependencies among functions that otherwise could be encapsulated in own modules.

5.3 Summary

The goal of this chapter was to validate our claims of flexibility and reusability against implementations of real protocols. We implemented RTP/RTCP to see if our concept of separating parsing (entry) and processing (worker) works also for more sophisticated header formats. We decided to re-structure and modularize a TCP like protocol implementation to see if our model can cope with sophisticated services and complex interactions among many components. From our experiences, we can state that our model is expressive enough to capture all aspects of even complex protocols. However, we also saw that the compatibility constraints during the implementation of existing protocols can complicate or even contradict the objectives of modular implementation. The definition of the header formats has thus an important impact on the implementation.

In Appendix D, we present another case study that shows how the flexibility of our approach allows to integrate different QoS requirements into distributed object systems by generating tailored protocol and middleware code based on Java `interface` definitions.

Chapter 6

Conclusions

In this chapter, we give an overall assessment of our work with regard to the requirements of network protocol software in general (like compatibility, performance, quality of service), and particularly with regard to our claims stated in Chapter 1 (like rapid prototyping, ease-of-use, flexibility). We further evaluate the software engineering techniques used during modeling, design, and implementation, and discuss possible improvements and future work.

6.1 Introduction

The overall goal of our research is to provide support for the implementation of application-tailored protocol software. The foundation of our work is a set of structuring principles, which allow to decouple protocol functions and encapsulate them in independent components that can be configured and composed to application-tailored protocol code. The PITOU framework serves as a prove of concept implementation of our structuring approach, and provides support for visual composition and configuration, rapid-prototyping and validation of application-tailored protocols. PITOU comes along with a number of tools for protocol edition, simulation, and animation. Any PITOU protocol can be used without any modification in a simulation environment or even graphically animated. We implemented TCP and RTP/RTCP within our framework to get insights about the expressiveness of our approach, and demonstrated the applicability of application-tailored protocols for the development of QoS aware middleware systems.

A second goal of this thesis is to examine how modern software engineering techniques can contribute to our goals of implementation productivity and cost minimization due to flexibility and reusability of protocol code. Special focus was put on component-based development (CBD) using the Java-Beans component model, the composition based on a black-box framework, and the identification of design patterns.

6.2 Evaluation of our Approach

The merit of our work lies in the applicability and expressiveness of our structuring approach, the ease and comfort to compose protocol software under the PITOU framework, and the quality of the code that is produced using the PITOU framework and its tools. Due to the nature of our work, it is difficult to evaluate our contributions in terms of quantitative measures. The evaluation is rather qualitative and based on our experience.

6.2.1 Productivity

The developer of a new protocol does not need to worry about designing a system architecture or to deal with concurrency issues since these problems are already solved by the PITOU framework. He can concentrate on the implementation of the workers and entry-types he needs. Since worker and entry are fine-grained components with a well-defined and simple task, they can be implemented rapidly, and tested independently from the context they will be used in. Confirming Dijkstra's "divide and conquer", the fine granularity of our approach greatly reduces the implementation time for new components.

Once all necessary worker and entry components are available, they can be easily plugged together to compose a complete protocol implementation. The process of composition and configuration is explicitly supported by our framework and its accompanying tools. Even visual guidance is possible using Java-Beans tools. In contrary to common object-oriented approaches, which suffer from implicit dependencies among objects, our approach makes dependencies explicit. Experiences showed that the process of building a new protocol based on existing components is a question of some minutes. The composition of such a complex protocol as the TCP like protocol presented in Chapter 5 took not even half an hour. The separation of component implementation and component assembling allows even people without many experiences in protocol engineering to compose new protocol implementations rapidly.

Our design philosophy of considering a protocol as part of the application gives the application developer full control over protocol design and offloads him from developing a message syntax to be respected: he just chooses and configures the workers that are needed, while the message format is just a by-product of the workers parameter requirements. Modifications, experimentation, and optimization of a distributed application are greatly simplified when there are no constraints concerning message formats.

We believe that our framework is very easy to use. Due to the Java Virtual Machine concept it can run on any operating system and runs everywhere Java is installed. Due to its clear structure, the necessary implementation steps are limited to a few classes, and new protocols can be composed and configured visually from existing components even by people who do not know Java. The time to get used to PITOU is very short as the experiences with two student projects have shown.

The maintenance phase of the software life cycle can be very expensive when the software does not anticipate changes due to bugs detected or the integration of new features in response to user feedback. Our framework is designed for changes on all levels ranging from simple changes of parameters (like time-out values) or the size of header fields, the modification of the operations of a data path (adding and removing workers), until the addition or removal of complete services (adding and removing order-types). Design for change is perhaps the most important contribution of our work.

Finally, the number of tools that come along with the core execution framework, significantly improve the productivity for protocol implementation. Simulation and animation reduce the testing and debugging efforts, the protocol editor is useful to combine existing protocols to new ones.

We briefly resume how our approach contributes to improve productivity for protocol implementation:

- fast implementation due to fine-grained modularity
- rapid-prototyping due to reusability, flexibility, configurability
- ease-of use of framework due to visualization
- easy maintenance due to design for change
- testing phase shorter due to visual animation and simulation

6.2.2 Code Quality

In the first chapter we gave a number of properties that characterize ”good” code. Most of those are already mentioned in the context of productivity above. We therefore only sketch why we believe that our approach contributes to the quality of the resulting software.

- Correctness: fine-granular, independent components – as promoted by the PITOU framework – facilitate testing and debugging compared to large software blocks. Moreover, the strict structure imposed on the use and interaction of our protocol components allows to detect errors within a few validation steps before testing and executing the software: have all workers the right entry-types as parameter types? does each reception order-type has a corresponding emission order-type on the peer that supports the same message format? are all notifiees connected with a notifier? are all resource-subscribers connected with a resource-publisher? Easy validation greatly contributes to the correctness and robustness of the produced code.

- Maintainability: the fine-granularity of the protocol components and the design for change make PITOU protocol code easy to maintain.

- Modifiability: the structure of the PITOU framework allows to add and remove protocol components and to configure them while minimizing the impact on other parts of the system.

- Reusability: PITOU fulfills the technical prerequisites for reusability. It decouples all components in the hot-spots and makes the remaining dependencies explicit. Using the reflective mechanism of the Java-Beans model, components are highly adaptable and support the visual configuration of their properties. However, assessing the reusability of the workers and entries we implemented for our example protocols is not easy since reuse is a long term concept and requires empirical validation. From our experiences, we are convinced that almost all entry-types are highly reusable. For example, the class `IntegerEntryType` appears in almost all protocol implementations we experienced with. The reusability of a worker component is certainly lower than the reusability of an entry-type since there are more different and specific algorithms in communication protocols than different header field types. The protocol editor provides also reuse of complete order-types and even clusters of order-types. However, a lot of protocol development projects would be necessary to reliably assess the reusability of larger abstractions (like clusters).

- Efficiency: though it would be interesting to see how our approach would compete with other structuring approaches if it would have been realized in a high-performance implementation (possibly in the kernel), the focus of our research was modeling and designing protocol software for flexibility, reusability, and rapid-prototyping.

  We wanted to show the benefits when protocol software is structured beyond layering, and therefore left out performance issues within our framework. Additionally, since the use of Java suggests already a performance penalty of about an order of magnitude compared to C, we considered comparisons with protocol software in other programming languages as not particularly useful.

  It is clear that our framework is not meant to re-implement a bulk data transport protocol as FTP/TCP. It would even contradict the concept of reuse to ignore the existence of a well-explored and highly optimized protocol as TCP. General purpose protocols are meant to be implemented only once as efficient as possible. We address application-tailored protocol implementation where flexibility and modifiability are more important than optimization of performance.

- Robustness: due to the modular structure, which allows independent testing, and the fact that PITOU protocol software is composed out of existing and tested modules, the robustness of PITOU protocols is achieved much faster than in coarse-grained layered protocol software modules.

6.3 Software Engineering Techniques for Protocol Implementation

Component-based development technologies, design patterns, and (especially object-oriented) frameworks are recent concepts and important research topics in the domain of software engineering. Our experiences applying these concepts for protocol development allow us to reach several conclusions about the usefulness, applicability, and limitations of these software engineering technologies in the area of communications protocol implementation.

6.3.1 OO Frameworks

The concept of using frameworks to support the implementation of a family of related applications has become very popular with the advent of object-oriented languages. Although frameworks have been already implemented and used without the OO paradigm – and as we saw in the related work section of Chapter 4, have a long tradition especially in the area of protocol implementation – they gained special attention and stimulated research only when the OO paradigm won recognition. OO modeling and programming are simply more appropriate than function-oriented languages to design the boundaries between what is *abstract* and what is *concrete*, what is *application* specific and what is *framework* specific, since OO development provides explicit support for modeling and designing the *hot-spots* of a framework (abstract classes, Java `interfaces`, C++ `virtual` methods). OO frameworks are thus generally easier to document, to understand, and to use than frameworks in function oriented languages.

Evolving OO Frameworks

In order to evaluate the concept of OO frameworks for protocol implementation, we interrelate our experiences during the implementation of PITOU with an article of Ralph Johnson [115], which defines a language of patterns for the development and evolution of object-oriented frameworks. This article describes a common path during the development of a framework, which is depicted in Figure 6.1.

The pattern *Three Examples* proposes to develop three applications one believes that the framework should help to build.

After implementing the first example application, the pattern *White-Box Framework* proposes to

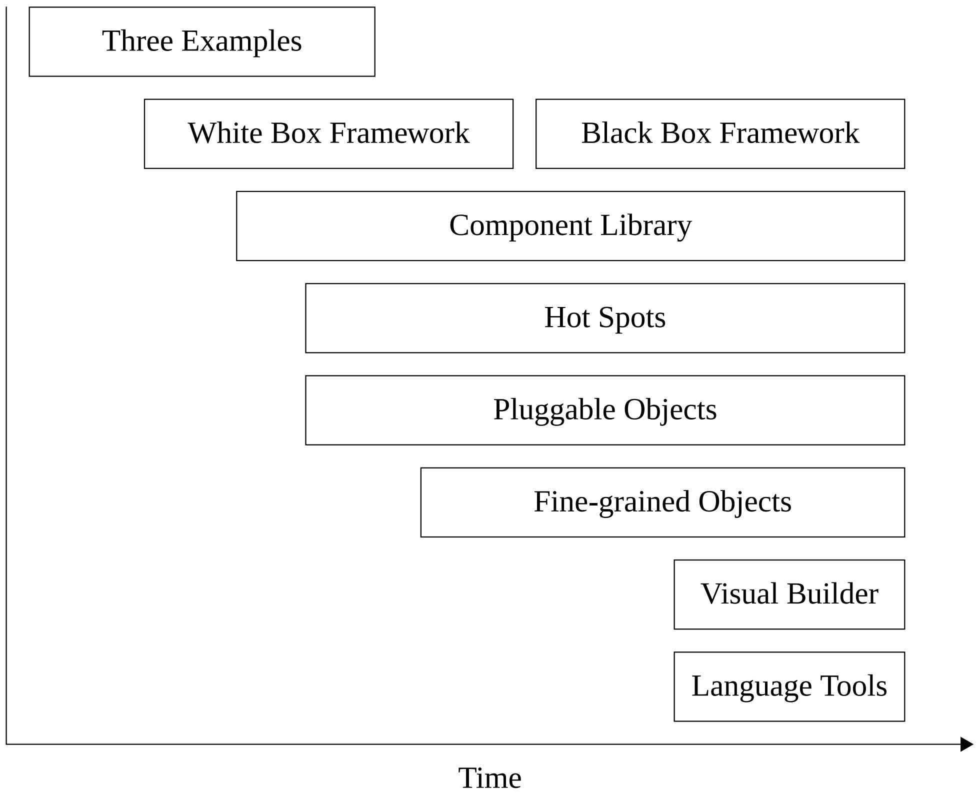


Figure 6.1: Evolving Framework (Johnson 1998)

generalize from the classes in the individual applications by using the inheritance feature. Since probably not enough information is available that allows to identify immutable code, *White-Box Framework* is a way to reuse existing code and provide a base for later transformation into a system that allows for composition.

During development of the subsequent example applications, *Component Library* proposes to start building a simple library of obvious objects, add additional objects every time when needed, and remove them when they are not frequently used. *Component Library* thus helps to identify reusable parts of an application and distinguish changing from static parts of an application.

During the development of a component library, *Hot Spots* proposes to separate code that changes from the code that doesn't. The varying code should then be encapsulated within objects that can be used for composition.

At the same time *Pluggable Objects* proposes to make objects adaptable by parameterizing them. Complementary to *Hot Spots*, *Pluggable Objects* wants to avoid encapsulation when the variation of code is minimal and can be captured by configuration.

Fine-Grained Objects proposes to break the objects of the library into finer granularity (until it makes no sense any further) to make them more reusable.

At that point, we have a white-box framework that heavily relies on inheritance. Inheritance means a strong coupling between components and requires code modification to adapt a component. It requires a good knowledge about the internal details of a component, but gives the programmer full control to change whatever he needs and the original designer never thought of. In order to improve reusability and flexibility in application development, *Black-Box Framework* proposes to restrict inheritance only to organize the component library and use composition to combine the components into applications. The application developer then does not need to worry *how* components accomplish their tasks, and simply needs to plug them together.

Once a black-box framework is established, *Visual Builder* proposes to make a tool that lets the developer specify the objects that will be in the application and how they are interconnected in a visual manner. The tool should generate the code for an application from its specification.

Once a builder is created, *Language Tools* proposes to create specialized inspecting and debugging tools to deal with the specialized composition relationships of the framework.

The Evolution of PITOU

Good knowledge in the area of communication protocols and personal experiences with the implementation of transport and application protocol software helped us to start the modeling on a rather high level of abstraction. We therefore did not need to rely on the *Three Examples* pattern, and instead constructed several scenarios with different kinds of protocols that served as a base for our model.

In contrary to the development of a framework for a new family of applications, communication protocol frameworks have a more than 10 years history that allowed us to build upon the experiences of predecessors frameworks like Conduits [64], X-Kernel [68], or BAST [60]. From the beginning, our design goals flexibility and reusability lead our research in the direction of a black-box framework. The question of how to decompose protocol software into decoupled, reusable, and configurable entities was the most important question during the whole development process. We therefore did not take the detour of *White-Box Framework*, and directly modeled and designed for composition proposed in *Black-Box*.

The development of a class library as proposed in *Component Library* is a by-product of every new protocol we implement in PITOU. While for Johnson building a class library is an trial-and-error like instrument to gain insights about the later design of the framework, for us it is rather an instrument of reuse for workers and entries, and started later in the evolution process of our framework compared to Figure 6.1. Again, the existing research and personal experiences made it possible to save some iterations of changing model and design.

Hot Spots is one of the major evolution steps in PITOU. Our goal was to minimize the number of hot-spots to ease using our framework. Finally, any beginner can build a simple protocol by understanding the interfaces of `Worker`, `EntryType/Entry` and `SAP` classes, which are the most important varying classes (hot-spots) in our design. The advanced programmer may also change `MappingStrategy` or `Executor`. Classes like `Environment`, `AcceptanceType`, `Emission`, etc. represent the fixed parts and what we called core framework classes.

Pluggable Objects is applied every time a new protocol is implemented. Normally, the determination of configuration parameters (in Java-Beans terminology ”property”) is a very natural and intuitive process. Examples are time-out values, packet-size, and forward-error-correction code parameters. We use the same idea as the Java-Beans component model to define configurable parameters, i.e. naming conventions (`set` and `get` methods) together with reflection.

Applying *Fine-Grained Objects* for PITOU is partly done during defining the structure and the hot-spots. On the other hand, it is the task of the programmer of a protocol to assure the fine-granularity of the components he implements. Theoretically, the implementation of a worker could be done in a way that the worker contains e.g. the complete output functionality of a TCP layer. There may be also possibilities to structure workers into finer-grained objects using e.g. the *State* or *Strategy* pattern.

Visual Builder has been realized partly within PITOU. Instead of building a specialized visual builder tool, we first used a general visual builder tool (Visual Age for Java) that supports the implementation of any kind of component-based development that conforms with the Java-Beans mode. Due to the experienced deficiencies, we modified the component-model (registrars and reification of interactions) and implemented a protocol editor that allows to edit protocol software – but not supported by a graphical interface.

Language Tools deal with the complexity of compositions and provide inspection and debugging support. The PITOU tools for validation, simulation, and animation fall can be considered as implementation of the *Language Tool* pattern.

In general, we can say that the evolution of the PITOU framework matches well with the proposed pattern language. There are however some specifics of the application family we address (application-tailored protocol software), which rendered the implementation of PITOU easier than expected:

- **Use of Java**: Java itself can be considered as a framework, since it provides a.o. automated garbage collection, comfortable thread support, and reflection classes (to introspect the properties of classes and their methods). Some important and possible tricky design issues have such been solved before they appeared for us.

- **Research on protocol software**: we could tap into a pool of rich experience in the domain of protocol structuring, implementation, and frameworks. That way, we could avoid a number of mistakes and thus iterations during the design of our framework.

- **Nature of protocol software**: protocol software does not consist of large and complex components as e.g. accounting software, and is rather easy to survey.

All these points allowed us to concentrate from the beginning on the key issue of finding good abstractions and an expressive structure for protocol software, which could then be implemented straight-forward in our framework.

We can clearly state that the development of a OO framework to compose protocol software is an extremely powerful and successful mean to achieve all goals stated above. We don't see a viable alternative to developing a framework. However, we do not want to hide the problems we experienced with regard to using frameworks.

- **Stability of the hot-spots**: the framework design largely depends on the stability of the hot-spot classes. If e.g. the abstract class `Worker` changes one of its abstract methods, all concrete worker implementations (our framework currently consists of around 50) must be adapted to conform with the modified `interface` of the worker class. We made this unpleasant experience once when we changed a method name of `EntryType`.

- **Stability of the core-classes**: a similar argument as for the hot-spot classes applies when classes are modified that concern the fundamental structure of the framework. When the animation tool design started, PITOU still relied on the Java-Beans connection model. The modification to `RelationRegistrar`s required the modification of large parts of the animation tool.

- **Static interfaces**: method declarations are tightly coupled with the class they are implemented for and not accessible as dynamic entities. This coupling of interface and implementation can be circumvented by reifying methods in first type classes as we have done this to allow for interaction of arbitrary workers at the price of having potentially more ”helper” classes than ”full” classes. It would be nice to have flexible and dynamic interfaces as OO language feature to make frameworks more flexible and easy to use.

- **Documentation**: we found it rather difficult to document the PITOU framework. Should we first explain how to use the framework and risk that the reader has not enough information to understand our explanations? Should we first give a detailed documentation of classes and system details and risk that the reader gets tired and spends a lot of time before he can start working with the framework?

6.3.2 Component-Based Development

Component-based development is a key concept to provide code reusability. The focus on well-defined interfaces and configurability provides the technical requirements for reusability. In the context of protocol implementation, we found the following benefits of using component-based technologies.

- **Team-development**: components can be plugged together even when they are developed by different people and teams (as long as they conform with the interface specification of the component technology used). The user of a component just needs to know about the component's interface and its tasks (in contrary to inheritance based reuse). Different people contributed to the worker library. Even a module that was not designed to be used within PITOU, could be integrated by being encapsulated in a worker wrapper class.
- **Standardization**: the example Java-Beans showed that one visual builder tool can be used for different applications. The use of VAJ allowed us to experiment with visually composing and configuring protocol software without the need of taking the expensive efforts of implementing a specialized tool.
- **Localization of change**: when interface definition and concrete implementation are decoupled (as done for e.g. workers), changing the implementation does not affect any other part of the software. CBD thus allowed to make changes transparently for the application.
- **Dynamic character**: CBD supports black-box composition, and thus allows to perform changes even at runtime. In PITOU, we do not allow to change the protocol structure or configuration during the execution of the protocol. However, our protocol editor and the validation tool work with runtime instances instead of formal specifications.

Although the benefits of CBD are indisputable and contributed to a large degree to the success of PITOU, we still present a list of problems and possible improvements that are partly related to the character of protocol implementation.

- **Components are just a part**: components can not capture global aspects of an application. This makes optimization rather difficult if not even impossible. Additionally, building components does not free the designer from developing an appropriate architecture for his application. We believe that CBD is useful only in the context of a good design or better a OO framework.
- **Complexity**: to be flexible, our components are very fine-grained. On the other hand, fine granularity shifts complexity from the components itself to the interactions of the components. It is not easy to keep a clear view of all interactions, especially when the

builder tool used has limitations. Most errors we encountered using a builder tool were due to forgotten or flawed specification of interaction.

- **Separation of domain and implementation knowledge**: in the domain of protocol implementation, the domain knowledge (understanding communications protocols) is very close to the implementation knowledge. Most protocol experts have already implemented a protocol. This is why we believe that one key benefit of CBD – separating domain and implementation knowledge – is not very relevant to protocol implementation.

- **Dependencies on the component model**: each component model imposes certain conventions to be respected. In order to connect workers via Java-Beans tools, we must implement the methods indicating an event source (`addABCListener` etc.). When we replaced Java-Beans by our own connection mechanism using explicit interactions, we had to change a lot of classes. These dependencies on the component model is a large obstacle to evolution and reuse.

- **Visual builders**: using common Java-Beans visual builder tools is easy, but their capacities are limited and most tools are not mature enough. Building an own tool is very expensive and should be done only when no fundamental changes are expected any more – but when are we sure about that?

6.3.3 Use of Patterns

Patterns are considered a useful instrument to document successful solutions to recurring problems. In this section, we discuss our experiences in identifying, documenting, and using patterns during analysis, design, and implementation of PITOU.

There are not many solutions or patterns to structure protocol software. The *Layers* pattern is an architectural pattern, but does not provide design and implementation support. The work of Hüni [64] is the first one that proposes to exploit the design patterns of Gamma [59] to improve the quality of protocol code. However, although the design patterns of Gamma are useful during design and implementation, they are very general and do not indicate how to structure protocol software. We do not know of any work that identifies domain (”analysis”) patterns for communication software. Maybe this is due to the fact that the idea of flexible, application-tailored protocols is rather new and not yet explored.

We appreciated the use of domain patterns especially as a documentation tool. After several attempts to describe our structuring approach of Chapter 2, we tried it in a pattern style by defining problem and context, weighting the forces, and proposing a solution. We found that using a pattern format was the easiest and clearest way to describe a solution, since a pattern offers a more general and abstract perspective to a solution and focuses on the main elements while leaving out details. That way, our domain patterns from Chapter 2 also facilitated the description of the final framework, since they provided a vocabulary and permanent guidance through structure and architecture of this dissertation.

Domain patterns are not only useful to document our framework, they are also directly applicable for applications with similar requirements. Their structure is valid and useful, even when no framework exists. For example, one could imagine to apply *Data Path Reification* also for high-performance network and kernel protocol implementation.

The use of design patterns (particularly those of Gamma et al. [59]) have been useful as an instrument to justify design decisions and to establish a culture of design to improve the mapping from design to implementation. They give guidance on a rather low level once the overall structure and architecture has been finalized. It was interesting to see that design patterns can also bring issues into mind, one would not have thought about originally.

However, the use of patterns does not replace the need for a careful analysis and design, neither do patterns lead to direct reuse of code. Naively used, they may even narrow the perspective of developers that look for solutions instead of understanding the problem. The number of existing patterns, their heterogeneity with regard to the level of abstraction, the fluent boundaries between domain, design, and architectural patterns, the naming ambiguities of existing patterns, and the lack of structured design in the domain of protocol implementation made it difficult for us to find, study and apply relevant patterns. We therefore used patterns as an instrument to justify and confirm design decisions rather than as a cook-book to build software.

6.4 Outlook

Besides framework refinements and optimizations, the implementation of more protocols with different requirements would be the next logical step to do. The size of the PITOU component library is important to reach conclusions about the degree of reusability achievable, to identify communication patterns within protocols, to identify classes of workers, and to get insights about what kind of optimizations are necessary.

Although PITOU allows to visualize protocol implementation based on the general Java-Beans component model, it would be appealing to have a graphical tool that guides the complete process of composing and configuring protocols in a comfortable manner adapted to the specifics of protocol implementation.

It would also be interesting to see how our structuring approach or at least some of the structuring principles proposed can cope with the needs of high-performance networking. In the context of kernel-protocols, our idea may serve as a playground for prototyping and experimenting with new network and transport protocols. Possible applications might also be small devices (Java-Cards, PDAs, Mobile Phones).

Our structure *Order-Worker-Entry* seems generic and suggests that it could be applicable also for applications other than protocols. Generalizing and using the PITOU framework as a generic framework for other frameworks or application families would surely be an interesting research direction.

Finally, we believe that building a QoS aware middle-ware system (as we sketched in Appendix D) would be an interesting follow-up project. The question of how to express quality of service requirements as structures of workers and entries is thereby of particular importance.

6.5 Summary of our Contributions

The thesis addresses the implementation of application-tailored protocols, i.e. protocols that deal with application and transport semantics and that are part of the application code. We make several contributions to improve software productivity and quality for this class of applications.

Protocol structuring: We demonstrate the problems of applying the traditional layered approach to protocol software structuring, and propose a set of structuring principles that allow to tailor protocol software to the needs of its application. The common idea behind all our principles is a vertical structure based on the notion of a data-path. Our structure reduces cross-talk between different streams to allow for application-specific and service-specific quality of service control, provides high flexibility in applying different thread-strategies, and divides protocol software into independent, fine-grained, and configurable components.

Protocol implementation support: Based on our structuring approach and recent advances in software engineering, we make another significant contribution by realizing a Java framework that supports the composition, configuration, and rapid-prototyping of application-tailored protocols. We demonstrate that it is possible to generate new protocols by assembling and configuring existing components using (Java-Beans compliant) visual builder tools without writing any line of code. In order to overcome the problem of the Java-Beans approach to cope with the complexity of component interactions, we propose to *reify interactions* into first class objects. We showed how this solution increases the robustness of protocol code (since it facilitates the validation of the correctness), and how it contributes to seamless simulation and animation of assembled protocol software.

Another contribution of this thesis is the design and implementation of a protocol simulator that can operate on the same components used to compose ”real” protocol software and that allows to attribute virtual time to fine-grained, internal tasks of implementations. Another tool allows to represent protocol sessions in a graphical manner, and – similar to the simulation tool – does not require any modifications of the original protocol code.

Evaluation: We demonstrated the expressiveness and applicability of our structuring approach by implementing existing protocols. Our structuring approach can deal with the complex header structure of the RTP/RTCP protocol and with the sophisticated functions of TCP. We further sketched how our approach can contribute to integrate quality of service management for middle-ware systems.

Finally, we contribute to software engineering research by reporting on our experiences with the software engineering concepts we have used, and by evaluating OO frameworks, component-based development, and patterns in the context of protocol implementation.

Appendix A

Signatures of the Main PITOU Interfaces and Classes

A.1 Hot Spot Classes

```
public abstract class Worker {
  public boolean call(Entry[]);
  public void init(Entry[]);
  public void deactivate();
}

public abstract class EntryType {
  public Entry create();
  public void setInitFlag(boolean);
  public void getInitFlag(boolean);
  public void setVisibleFlag(boolean);
  public void getVisibleFlag(boolean);
  public void setSize();
  public void getSize();
}

public abstract class Entry {
  public void fill(Packet,int);
  public void fill();
  public byte[] serialize();
  public void setValue(Object);
  public Object getValue();
}

public interface OutOfBandModule {
}

public abstract class Address {
}

public abstract class SAP {
  public void open();
  public void close();
  public void setPacketSize(int);
  public int getPacketSize();
  public Packet read();
  public Packet readUntil();
  public void send(Packet);
  public Address getDefaultDestination();
  public void setDefaultDestination();
  public Address getOwnAddress();
  public void setOwnAddress(Address);
}

public abstract class MappingStrategy {
  public void map(Packet);
  public void outgoing(EmissionType);
}

public interface ReadAPI {
  public void deliver(Object[],DeliveryType);
}

public interface WriteAPI {
  public void setAcceptanceType(AcceptanceType);
  public void accept(Object[]);
  public void accept(Object[],Address);
}
```

A.2 Utility Classes

```
public class Timer {
  public void reset(long);
  public void cancel();
  public long getAbsTimeout();
  public void endUse();
}

public class TimerPool {
  public Timer getNewTimer();
  protected void reset(Timer,long);
  protected void cancel(Timer);
  protected void endUse(Timer);
}

public class StateManager {
  public void signal(EventRaiser);
  public void init();
  public void addSessionState(SessionState);
  public void setStart(SessionState);
}

public class SessionState {
  public void setName(String);
  public String getName();
  public void addTransition(EventRaiser,SessionState);
  public void addFilter();
  public void enter();
  public void init(StateManager);
  public void transit(EventRaiser);
}

public interface Filter {
  public void letNotPass();
  public void letPass();
}
```

A.3 Classes for Registration and Assembly

```
public class OrderRegistrar {
  public void addOrder(OrderType);
  public void init();
}

public class StructureRegistrar {
  public void addEntry(EntryType);
  public void addWorker(Worker);
  public void addWorkerParameter(EntryType,Worker);
  public void init();
}

public class OutOfBandRegistrar {
  public void addOutOfBandModule(OutOfBandModule);
  public void init();
}

public class MappingStrategyRegistrar {
  public void addMappingStrategy(MappingStrategy);
  public void init();
}

public class SAPRegistrar {
  public void addSAP(SAP);
  public SAPRegistrar create();
  public void init();
}

public class RelationRegistrar {
  public void addRelation(Relation);
  public void init();
}

public class ExecutorRegistrar {
  public void addExecutionGroup(ExecutionGroup);
  public void init();
}
```

```
public class EventRaiser {
  public void addProtocolEventListener(ProtocolEventListener);
  public void removeProtocolEventListener(ProtocolEventListener);
  public void raise(ProtocolEventObject);
}

public interface Caller implements ProtocolEventListener {
  public void call();
  public void init();
  public void setNotifiee(Object);
}

public interface Relation {
}

public class NotificationRelation implements Relation {
  public NotificationRelation(EventRaiser,Caller);
  public void setEventRaiser(EventRaiser);
  public void setCaller(Caller);
  public void init();
}

public class NewOrderRelation implements Relation {
  public NewOrderRelation(EventRaiser,Order);
  public void setOrder(Order);
  public void setEventRaiser(EventRaiser);
  public void init();
}

public interface Resource {
  public Object read();
  public void write(Object);
}

public class ResourceAccessor {
  public Object read();
  protected void assignResource(Resource);
}
```

```
public class SharedResourceRelation implements Relation {
  public SharedResourceRelation(Resource,ResourceAccessor);
  public void setResource(Resource);
  public void setAccessor(ResourceAccessor);
  public void init();
}
```

A.4 Framework Core Classes

```
public class Packet {
  public Packet(Address,byte[]);
  public Address getAddress();
  public byte[] getBytes();
}

public interface Acceptable {
  public void setWriteAPI(WriteAPI);
  public WriteAPI getWriteAPI();
}

public interface Deliverable {
  public void setReadAPI(ReadAPI);
  public ReadAPI getReadAPI();
}

public interface Emittable {
  public Address getAddress();
  public int getSAPNr();
  public void setSAPNr(int);
  public int getOrderID();
  public void setOrderID();
}

public interface Receptable {
  public Address getAddress();
  public int getSAPNr();
  public void setSAPNr();
  public int getOrderID();
  public void setOrderID();
}

public interface Order {
  public boolean execute();
  public boolean initialize(Packet);
  protected byte[]
  protected byte[] serialize();
  protected Object[] objectivate();
}

public interface OrderType implements Filter {
  protected void setOrderTypeManager(OrderTypeManager);
  public Order createOrder();
  public void setStructureRegistrar(StructureRegistrar);
  public void activate();
  public void deactivate();
  public void requestOrder(Packet);
  public void executionTerminated();
  public void setPriority(int);
  public int getPriority();
  public EventRaiser defineEvent_execDone();
}

public interface AcceptanceType extends Acceptable, OrderType {
}

public interface EmissionType extends Emittable, OrderType {
  public Resource defineOutput_add();
  public Resource defineOutput_id();
}

public interface DeliveryType extends Deliverable, OrderType {
}

public interface ReceptionType extends Receptable, OrderType {
  public Resource defineOutput_add();
  public Resource defineOutput_id();
}

public interface Acceptance extends Acceptable, Order {
}

public interface Emission extends Emittable, Order {
}

public interface Delivery extends Deliverable, Order {
}

public interface Reception extends Receptable, Order {
}

public abstract class OrderTypeManager {
  protected assignExecutor(Executor);
  protected assignEnvironment(Environment);
  protected assignOrderType(OrderType);
  public void requestOrder(Packet);
}

public interface Environment {
  protected appData(Object[],Address,Acceptable);
  protected netData(Packet,int);
  public void setOrderRegistrar(OrderRegistrar);
  public void setExecutorRegistrar(ExecutorRegistrar);
  public void setOutOfBandRegistrar(OrderRegistrar);
  public void setRelationRegistrar(RelationRegistrar);
  public void setMappingStrategyRegistrar(MappingStrategyRegistrar);
  public void setSessionStateRegistrar(SessionStateRegistrar);
  public void changeSAPs(SAPRegistrar);
  public void notifyTerminate();
  public void deactivate();
  public void init(InitManagerImpl);
  protected void exceptionOccured(Exception);
  public void setExceptionHandler(ExceptionHandler);
  public void addCloseListener(CloseListener);
  public Caller defineCaller_terminate();
```

```
}

public interface CloseListener {
  public void envClosed(Environment);
}

public interface ExceptionHandler {
  public void exceptionOccured(Exception);
}

public interface Executor {
  public void init();
  public void deactivate();
  public void execute(Order,int);
  public boolean isWorking();
}

public class ExecutionGroup {
  public void addExecutionGroup(ExecutionGroup);
  public void init();
}

public interface InitManagerImpl {
  public void init(Environment,OrderType[],ExecutorRegistrar);
}

public abstract class ProtocolBuilder {
  public Environment construct();
}
```

A.5 Simulation Classes

```
public class Activity {
  public void perform();
  public void setScheduler(Scheduler);
  public int getPriority();
  public void setPriority(int);
}
```

```
public class Processor implements Sortable {
  public void scheduleAt(Activity,long,long);
  public void scheduleAtEnd(Activity,long);
  public void cancel(Activity);
  public Activity getNextProc();
  public int getNrScheduled();
  public long getNextExecTime();
  public long getNextDuration();
  public void passivateFirst();
  public boolean equals(Sortable);
  public boolean greater(Sortable);
}

public class Scheduler {
  public void register(Activity,Processor);
  public void register(Activity);
  public void deregister(Activity);
  public void addVirtualTimeListener(VirtualTimeListener);
  public void removeVirtualTimeListener(VirtualTimeListener);
  public void scheduleAt(Activity,long,long);
  public scheduleNow(Activity, long);
  public void scheduleAfterRest(Activity,long);
  public void cancel(Activity);
}

public interface VirtualTimeListener {
  public void newVirtualTime(long);
}

public class Link implements Activity {
  public void addTarget(SAPvS);
  public long getOneTripTime();
  public void setOneTripTime(long);
  public void put(byte[],Address,Address);
}

public class SAPvS implements SAP {
  public void arrive(Packet);
  public long getDataRate();
  public void setDataRate(long);
}
```

Appendix B

Simulation Scenario

To illustrate how the simulator works that we described in Chapter 4, we have a look at the following scenario. We assume that all activities involved have been registered at the scheduler using the `register()` methods.

Imagine, we have four activities $A1..A4$ registered with two processors $P1, P2$. The activities are distributed as follows over the processors: $P1 = \{A1, A2\}$, $P2 = \{A3, A4\}$. $A1..A3$ have a priority of 1, $A4$ has a priority of 2. At the beginning, no activities are scheduled, i.e. all processors are sleeping.

Step 1: A1 is scheduled as soon as possible with a duration of $d_1 = 100ms$.

- The user schedules $A1$ via `Scheduler.scheduleNow(A1,100)`.
- The scheduler identifies the processor $P1$ as responsible, and schedules $A1$ at the current virtual time $vt = 0$ via `P1.scheduleAt(A1,0,100)`.
- The processor produces an information object $I_1 = (A1, 0, 100)$, and adds it to its queue $Q_{P1} = \{(A1, 0, 100)\}$.
- The scheduler inserts the processor $P1$ in its queue: $Q_S = \{P1\}$, and resumes its thread (which has been sleeping until now, because there has been no activity).

Step 2: A2 is scheduled as soon as possible with a duration of $d_2 = 150ms$. No activities have been executed yet.

- The user schedules $A2$ via `Scheduler.scheduleNow(A2,150)`.
- The scheduler identifies the processor $P1$ as responsible, and schedules $A2$ at the current virtual time $vt = 0$ via `P1.scheduleAt(A2,0,150)`.

- Since $A2$ overlaps with $A1$, it must be scheduled after $A1$, hence at $vt = 100$.
- The processor produces an information object $I_2 = (A2, 100, 250)$, and adds it to its queue $Q_{P1} = \{(A1, 0, 100), (A2, 100, 150)\}$.
- Since $P1$ is already present in the schedulers queue, the queue remains unmodified: $Q_S = \{P1\}$.

Step 3: A3 is scheduled at virtual execution time $e_3 = 50$ and duration $d_3 = 70$. Still no activities have been executed yet.

- The user schedules $A3$ via `Scheduler.scheduleAt(A3,50,70)`.
- The scheduler identifies the processor $P2$ as responsible, and schedules $A3$ at virtual time $vt = 50$ via `P2.scheduleAt(A3,50,70)`.
- The processor produces an information object $I_3 = (A3, 50, 70)$, and adds it to its queue $Q_{P2} = \{(A3, 50, 70)\}$.
- The scheduler sorts $P2$ in his queue of waiting processors. Since the first activity scheduled for $P2$ is executed later than the first activity scheduled of $P1$, $P2$ is inserted after $P1$ in the queue: $Q_S = \{P1, P2\}$.

Step 4: A4 is scheduled at virtual execution time $e_4 = 60$ and duration $d_4 = 30$.

- The user schedules $A4$ via `Scheduler.scheduleAt(A4,60,30)`.
- The scheduler identifies the processor $P2$, and schedules $A4$ via `P2.scheduleAt(A4,60,30)`.
- The processor produces an information object $I_4 = (P5, 60, 30)$, and inserts it in its queue $Q_{P2} = \{(A3, 50, 70), (A4, 60, 30)\}$ – attention: there is an overlapping between the two activities. $P2$ modifies its queue by adapting the execution times such that $A4$ starts only when $A3$ has terminated: $Q_{P2} = \{(A3, 50, 70), (A4, 120, 30)\}$.
- Since $P2$ is already present in the schedulers queue, the queue remains unmodified: $Q_S = \{P1, P2\}$.

An graphical view on the scheduled activities can be seen in Figure B.1.

Step 5: Execution of the next activity scheduled.

- The scheduler thread takes the first processor from its queue: $P1$ is now *active processor*.

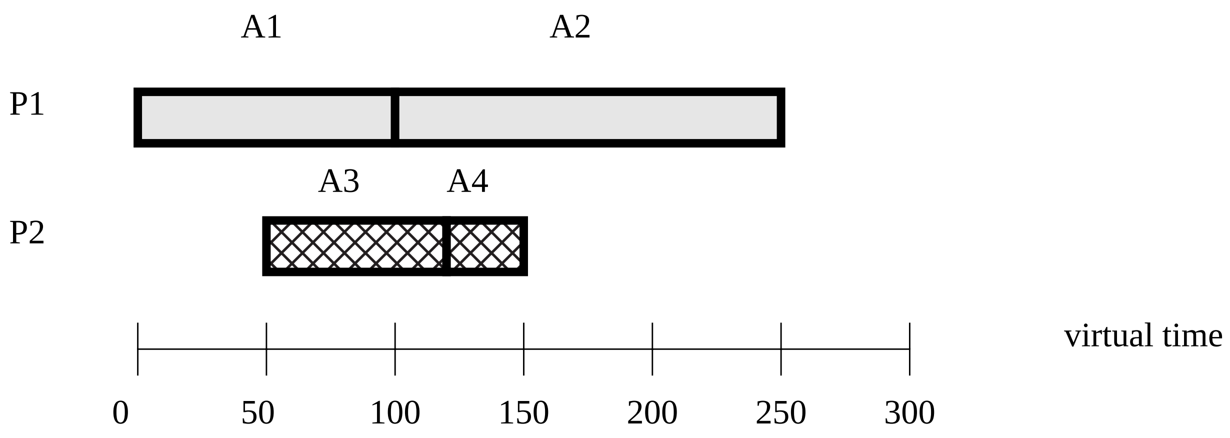


Figure B.1: Scheduling Scenario after Step 4

- The scheduler obtains the activity to be executed via `P1.getNextProc()`, its execution time via `P1.getNextExecTime()`, its duration via `P1.getNextDuration()`.
- The scheduler sets the virtual time to the execution time, hence the time does not change: $vt = 0$. The scheduler sets the maximum termination time to $tt = 100$.
- The scheduler notifies $P1$ that it can update its lists via `P1.passivateFirst()`. $P1$ updates it queue to $Q_{P1} = \{(A2, 100, 150)\}$
- The scheduler removes $P1$ from its queue. But since $P1$ has still another activity scheduled it will be re-inserted after $P2$: $Q_S = \{P2, P1\}$.
- The scheduler executes $A1$ via `A1.perform()`.

Step 6: Execution of the next activity scheduled.

- The scheduler thread takes the first waiting processor from its queue: $P2$ is now *active processor*.
- The next activity to be executed is $A3$, its execution time 50, and its duration $70ms$.
- The scheduler sets the virtual time to the execution time, hence the virtual time is $vt = 50$. The scheduler sets the maximum termination time to $tt = 120(50 + 70)$.
- The scheduler notifies all its listeners that the virtual time has changed.
- The scheduler notifies $P2$ that it can update its lists via `P2.passivateFirst()`. $P2$ updates it queue to $Q_{P2} = \{(A4, 120, 30)\}$.

- The scheduler updates its queue: $Q_S = \{P1, P2\}$.
- The scheduler executes $A3$ via `A3.perform()`.

Step 7: Execution of the next activity scheduled.

- The scheduler thread takes the first processor $P1$ from its queue.
- The next activity to be executed is $A2$, its execution time 100, and its duration $150ms$.
- The scheduler sets the virtual time to the execution time, hence $vt = 100$. The maximum termination time is set to $tt = 250$ (since $100 + 150 = 250$).
- The scheduler notifies all virtual-time listeners about a change of time.
- The scheduler notifies $P1$ that it can update its list: $Q_{P1} = \{\}$.
- The scheduler removes $P1$ from its queue.
- The scheduler executes $A2$.

Step 8: Execution of the next activity scheduled.

- The scheduler thread takes the first processor $P2$ from its queue.
- The next activity to be executed is $A4$, its execution time 120, and its duration $30ms$.
- The scheduler sets the virtual time to the execution time, hence $vt = 120$. The maximum termination time remains $tt = 250$ (since $120 + 30 < 250$).
- The scheduler notifies all virtual-time listeners about a change of time.
- The scheduler notifies $P2$ that it can update its list: $Q_{P2} = \{\}$.
- The scheduler removes $P2$ from its queue: $Q_S = \{\}$.
- The scheduler executes $A4$.
- The scheduler sets the virtual time to the maximum termination time: $vt = 250$, since no processors are waiting, notifies all virtual-time listeners, and brings its thread to sleep.

The internal states of scheduler and processors after each step is depicted in Figure B.2.

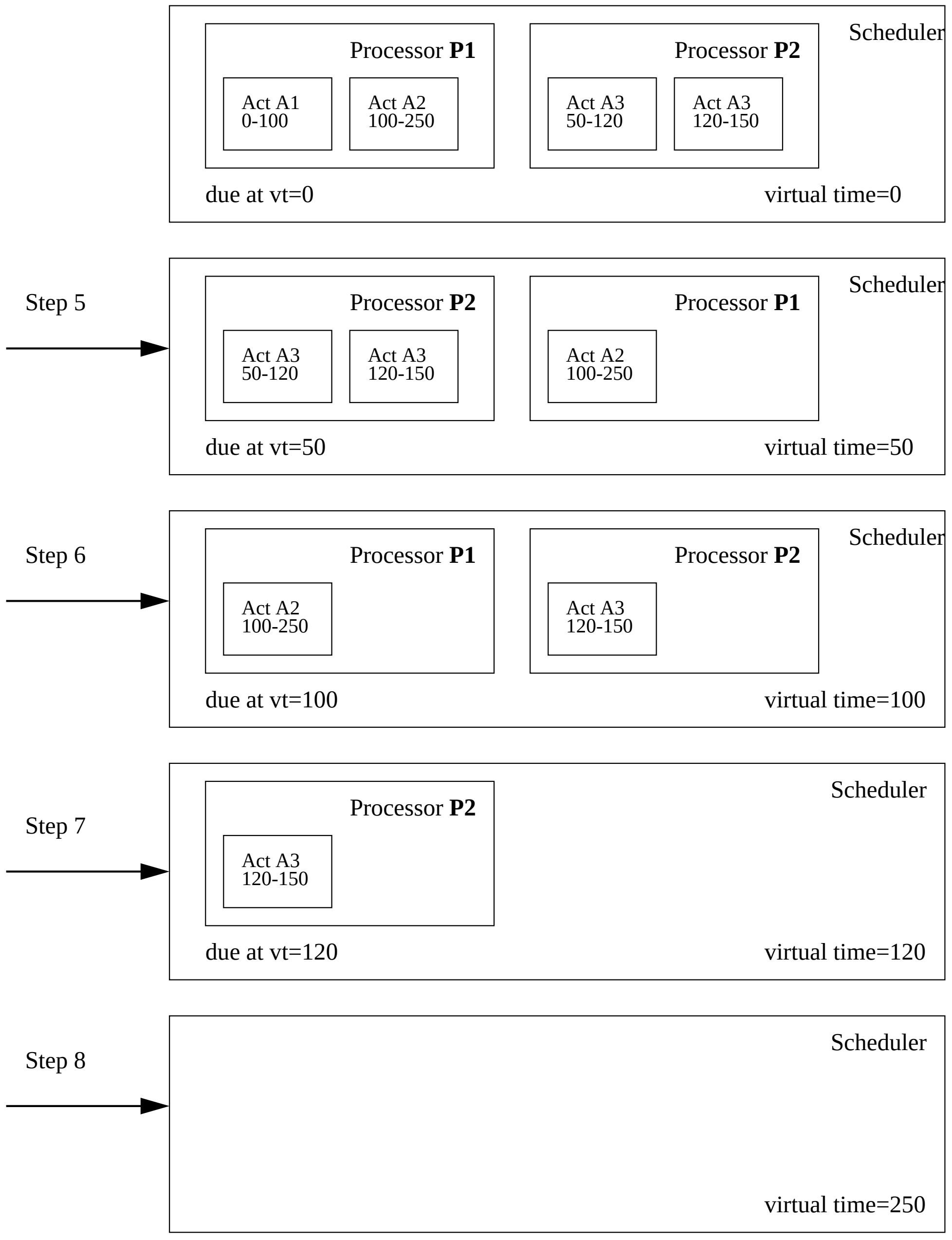


Figure B.2: The Scheduling Scenario during Execution

Appendix C

Integrating Simulation in PITOU

This chapter describes the design that allows us to extend the PITOU framework by a simulation mode.

C.1 Simulating a Network

From Chapter 3, we already know the types that PITOU defines to encapsulate network access: the abstract class `SAP` and the `interface Address`. Both are implemented with regard to simulation support. The class `SimpleAddressS` represents a virtual address that consists simply of a `long` value. The class `SAPvS` works as follows. Besides being of type `SAP`, a `SAPvS` object is also of type `Activity`, i.e. an activity that simulates sending data. It therefore implements the method `setScheduler()` to have access to a scheduler object, and `perform()` to execute sending data.

- The method `void arrive(Packet)` serves to receive data from a virtual network. The incoming data is written into a read queue.
- The methods `setOwnAddress()` and `getOwnAddress()` allow to assign a virtual address to a SAPvs object. This address is used by the virtual network to demultiplex data to the right SAP.
- The method `Packet read()` returns and removes the first element from the read queue (if there is one, otherwise it returns `null`).
- The methods `long getDataRate()` and `void setDataRate()` allow to specify the data rate, the SAPvs is able to send.

- The method `void send(Packet)` puts the data into a write queue, and schedules an activity specifying itself as activity for immediate execution, and a duration depending on data-rate and packet size.
- The method `void perform()` takes and removes the first element from the write queue, and sends it, i.e. gives it to a virtual network for transport. This method is called by the scheduler in response to the scheduling operation done in the `SAPvs.send()` method.

We already refered to a *virtual network* component. What classes are hidden behind this notion of a virtual network? In fact, our solution is extremely simple. We allow `SAPvS` objects to be connected with objects of type `Link`. Each `SAPvS` object provides a method `void setLink(Link)`, to which it hands out the data during the `perform()` method. The class `Link` implements the `Activity interface`, i.e. it acts as an activity that transmits data. It declares the following methods:

- The method `void addTarget(SAPvS)` allows to add `SAPvS` objects to a link.
- The methods `long getOneTripTime()` and `void setOneTripTime()` are used to configure the virtual time a message takes until it arrives at the other end.
- The method `void put(byte[], Address, Address)` is used by a SAPvS object to write data to the link. The other parameters specify source and destination address of the data. The link object puts the information in a queue, and schedules itself as activity for immediate execution and a duration corresponding to the specified one-trip-time.
- The method `void perform()` identifies the right `SAPvS` object using the specified destination address, and delivers the information.

Links may also be configured to simulate any kind of transmission errors (fading channels, linear bit errors, bursty bit errors, etc.). `SAPvS` may also be used to simulate overloaded machines or congested routers.

The following scenario illustrates how two environments E_1 and E_2 communicate via a simulated network L, which connects the simulated SAPs $SAPvS1$ and $SAPvS2$ (see also Figure C.1).

1. E_1 calls the method `SAPvS1.send()` after execution of an emission order.
2. $SAPvS1$ appends the information to its queue and calls the method `Scheduler.scheduleNow(this,duration)`, whereas $duration = packetSize/dataRate$.

3. The information remains in the queue unless the scheduler thread calls `SAPvS1.perform()`. Within the `perform()` method, the information is given to the link L via `L.put()`.

4. In the `put()` method, the information is written to the queue of the link. The link then schedules itself via `Scheduler.scheduleNow(this,oneTripTime)`.

5. The information remains in the link queue until the scheduler thread call `L.perform()`. Here, the destination address is matched with all destination SAPs (in our scenario there is only one), and given to $SAPvS2$ via `SAPvS2.arrive()`.

6. In the `arrive()` method, the information is stored in a queue, where it remains until the `read()` method is called.

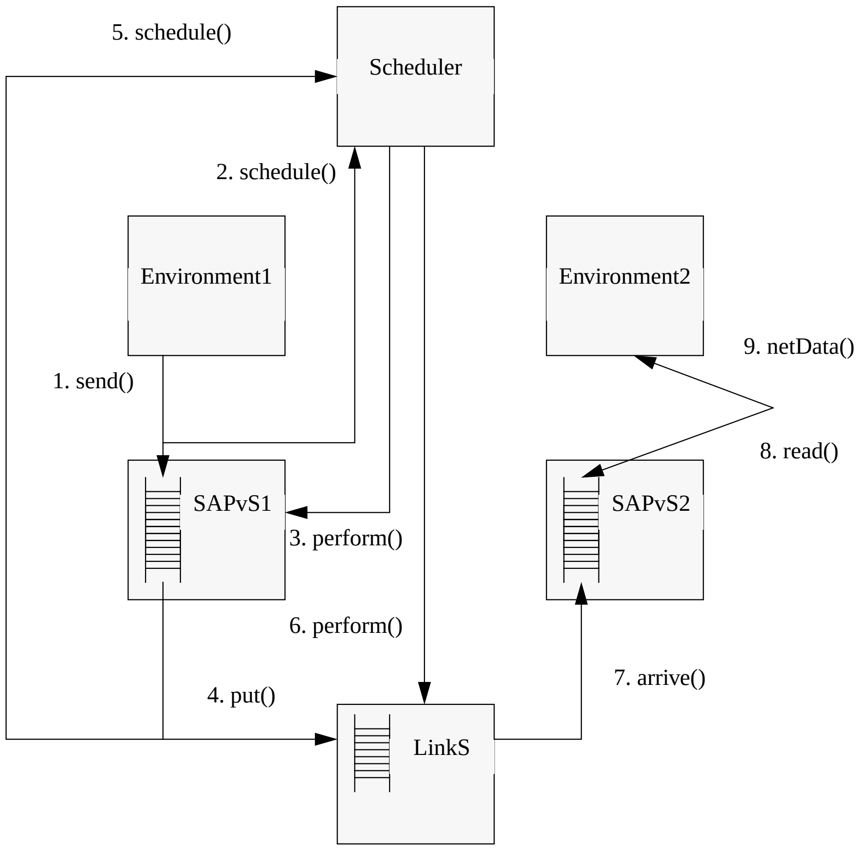


Figure C.1: Simulation of Sending Data

C.2 Simulating Time

The virtual time calculated by the scheduler replaces the real system time during normal operation. Since protocols often use timers, a careful design of how to realize timeouts within a virtual environment is of particular importance. We saw in Chapter 3 that every environment object is initialized with a global timer-pool, which manages all timers used by worker components of the environment. This centralized design facilitates the task of replacing system time by virtual time transparently for the protocol components.

We implement a class `TimerPoolS` – a timer-pool for simulated timers – which extends the class `TimerPool` presented in the last chapter. A simulated timer-pool manages simulated timers implemented as `TimerS`, which extends the class `Timer`.

A simulated timer-pool works very similar to the timer-pool described in the last chapter. But instead of implementing an own thread, which sleeps until the next time-out, a simulated timer-pool implements the `interface VirtualTimeListener` to be informed every time the virtual time changes (via the method `newVirtualTime(long)`). In `newVirtualTime()`, the virtual timer-pool checks if the next timeout is due, and eventually schedules a timeout via `Scheduler.scheduleNow(TimerS,0)`[1].

The class `TimerS` implements the `interface Activity` to be able to be scheduled as an activity. In the method `perform()`, it calls the timer-compatible object that requested the timer via `timerCall()`.

The following scenario illustrates how a simulated timer-pool works.

1. A timer-compatible worker obtains an object of type `TimerPoolS` during initialization time. For the worker, this object is of type `TimerPool`.

2. The worker demands for a timer object via `TimerPool.getNewTimer()`. The worker obtains a simulated timer object – for the worker, however, this object is of type `Timer`.

3. The worker activates a timeout after $100ms$ by calling `Timer.reset(100)`.

4. The simulated timer-pool calculates the timeout by adding actual time and expiry time, and adds the timer-information to his expiry list.

5. The scheduler notifies the timer-pool about a change of time via `newVirtualTime(long)`. The timer-pool realizes that the expiry time of the first timer in the list is higher than the current virtual time.

[1]The duration of a timer call is considered to be 0 and thus does not advance the virtual time. This is not realistic, but extremely simplifies timer-handling.

6. The timer-pool immediately schedules the corresponding timer-object via `Scheduler.scheduleNow(TimerS,0)`.

7. As soon as the timer object is the next to be executed, the scheduler thread calls the method `TimerS.perform()`.

8. In `TimerS.perform()` the worker's `timerCall()` method is called.

The scenario shows that the use of virtual time-outs instead of time-out based on the system time is completely transparent for the worker or any other component that uses timers.

C.3 Simulating Threads

We now treat the most complicated part in transforming PITOU into a simulation environment. How to simulate the threads (implemented by the `Executor` sub-classes) that execute orders? Instead of giving orders to executor objects, orders must be scheduled directly and executed by the scheduler. That is, modifications of at least one of the core framework classes – `InternalOrderType` – are necessary.

One option would be to re-implement the `OrderType interface` in a class `InternalOrderTypeS` with the functionality required for simulation. This approach has a number of disadvantages:

- The modification needed concerns only the way how to execute an order (and not how to manage the order pool). We would therefore need to re-implement most of the functionality already implemented in `InternalOrderType` a second time.

- Moreover, we must re-implement all the sub-classes of `InternalOrderType` to assure that they have the right type – although they do not comprise any method that must be changed.

- Besides the high number of classes to be implemented, there is another problem with this approach: the protocols already built use `InternalOrderType`, `AcceptanceType`, etc. to define its structure. We would thus need to re-build all protocols for the simulation mode.

We solved the problems above by introducing a class called `OrderTypeManager`, which takes on the following responsibilities from the class `InternalOrderType`: managing the order pool, handling order requests, and executing orders. `OrderTypeManager` implements the following methods:

- The methods `assignExecutor()`, `assignEnvironment()`, `assignOrderType()` (and some others) are used to provide the order-type manager with all information he needs to perform his task.
- The method `void requestOrder(Packet)` either fills the order object with data and asks an executor object to execute it, or puts the information into a queue, where it is processed later (when the order object is in execution or other requests are waiting).

The class `InternalOrderType` is modified as follows:

- It obtains a new method `assignOrderTypeManager()`, which allows to assign order-type manager objects.
- The method `requestOrder()` directly forwards the request for a new order to the corresponding method of its manager object.
- The method `init()` provides the order-type manager with all the information it needs.

This design now allows us to introduce simulation support into the `InternalOrderType` class by doing the following steps:

1. Extend the class `OrderTypeManager` by a class `OrderTypeManagerS`, which supports the simulation mode.
2. Implement the `Activity` interface for `OrderTypeManagerS`.
3. During initialization, give an object of type `OrderTypeManagerS` to the order-type objects to replace the original order type manager.

The object of type `OrderTypeManagerS` works as follows. When it obtains the request for a new order from its order-type object via `requestOrder()`, it adds the information to an internal queue and schedules itself as an activity. Each time the `perform()` method is called by the scheduler, it takes an element from the list, initializes the order object with the data, and calls the `Order.execute()` method to execute the order.

Hence, by simply assigning a different manager object, we are able to let objects of type `InternalOrderType` work in both real mode and simulation mode. No other classes need to be modified or re-implemented, and – most important – all protocols built can be reused unmodified. Figure C.2 shows the modifications made to support simulation of protocols.

Our solution is based on a design pattern called *Visitor* [59]. Gamma et al. define the intent of the visitor pattern as follows: *Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates*.

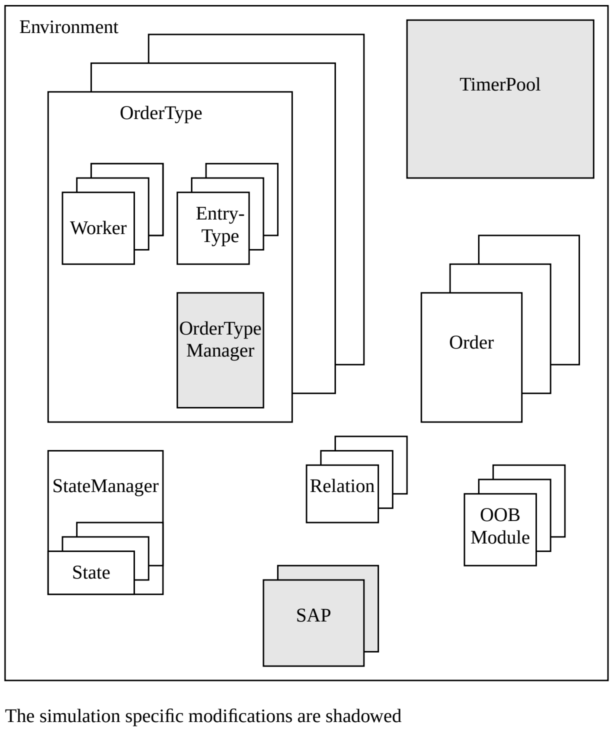


Figure C.2: Simulation specific modifications of PITOU core classes

C.4 Initialization

One thing is still missing: how and when is an order-type manager given to its respective order-type? The most natural thing to do this is during initialization of the environment, i.e. within the method `Environment.init()`. However, how should we tell an environment if it should work in simulation mode or in normal mode?

We decided to choose a solution similar to the use of different order-type managers. We define an `interface InitManagerImpl`, which defines the method `init(Environment,OrderType[],ExecutorRegistrar)` supposed to be implemented to perform initialization of order-types. The class `Environment` modifies its `init()` method by taking a parameter of type `InitManagerImpl`.

The class `InitManager` implements `InitManagerImpl` for the real mode. During `init()`, it assigns `OrderTypeManager` objects and executor objects to all orders, and a `TimerPool` object to the environment. The class `InitManagerS` implements `InitManager` for the simulation mode. During `init()`, it assigns `OrderTypeManagerS` objects to all order-types, and a `TimerPoolS` object to the environment. Additionally, it registers all order-types at the scheduler. It thereby uses the information about the execution-groups to decide, which order-types obtain an own processor `Processor`.

The application can assign a virtual execution time to each order by specifying a `long[]` array, and assigning it to an `EnvInitS` object via `void setOrderVirtualExecTime(long[])`. During initialization each value of the array is used to specify the `OrderTypeManagerS` of the corresponding order-type (the array indices must correspond to the order of registration of the order-type objects).

The assigned virtual times for each order execution should be based on real measurements to reach useful conclusions. Assigning a virtual execution time to each order-type allows to examine the behavior of a protocol with a very fine granularity. Most network simulators assign simply a data-rate to a protocol session, and ignore the processing time and other events than transmitting data. The PITOU simulator allows to count the time for the simulation of the protocol's internal operations, and thus to better identify implementation bottlenecks and design flaws.

It is up to the application now to initialize an environment either with an init-manager for real mode (`InitManager`) or for simulation mode `InitManagerS`).

C.5 Simulating a protocol – what need to be done?

The following scenario illustrates how to build an application that simulates the communication between two protocol sessions. These principal components of this application can be combined and configured visually.

The following beans are put on the screen and configured.

- two `SAPRegistrar` objects to register SAPs,
- two `SAPvS` objects,
- two `ProtocolBuilder` objects that define the desired structure of the environment,
- two `Link` objects to connect the SAPs (one for each direction of communication),
- a `Scheduler` object that controls the simulation

For the `SAPvS` objects, data-rate and address, for the `Link` objects the one-trip time is configured.

Now the beans are connected.

- All `SAPvS` beans are registered with the `SAPRegistrar` bean.
- The `SAPRegistrar` beans are assigned to the `Environment` object accessible from the `ProtocolBuilder` bean.

- The first `Link` bean is registered with the first `SAPvS` bean (and vice versa).
- The second `SAPvS` beans is registered with the first `Link` bean (and vice versa).

Before the application can start, code must be written in the `main` method that starts the scheduler thread, create an `EnvInitS` object to initialize the environment, and to obtain the interfaces of the environments to read and write data.

Appendix D

A Lightweight Distributed Object System

D.1 The Problem of Integrating QoS in Distributed Object Systems

Distributed object systems (DOS) (also refered to as middleware systems) like DCOM [87], CORBA [130], or RMI [138]) allow to simplify the implementation of distributed systems by hiding distribution concerns and allowing the programmer to concentrate on the application logic. However, todays DOS provide almost no support to accommodate the quality of service requirements of distributed applications. Developers thus either must accept service mismatches or make large efforts to integrate possibly complex communications related functions into the application, which clearly contradicts the objectives of middleware systems.

Integrating QoS management into DOS is a challenging research topic that concerns different components of the system. One research direction proposes to integrate QoS management into the middleware system by modifying and extending it. TAO [125] extends a CORBA Object Request Broker by real-time facilities. The Object Management Group (OMG) published specifications for real-time and fault-tolerant services. Other examples for specialized middleware systems are Electra [92], Eternal [101], or DOORS [150]. A second approach proposes to extend the programming model to make QoS concerns explicit. Examples for this approach are QualityObjects (QuO) [153], MAQS [12], and the Squirrel project [86]. Another approach proposed by [112] introduces *interceptor* layers between the middleware infrastructure and the application to avoid changing neither the middleware system nor the application logic.

We believe that one reason for the inflexibility of common DOS is due to the fact that distributed object services are generally built on top of TCP. The stream based nature of TCP does not match very well the request/response character of distributed object calls and does not provide any QoS differentiation. Consider a distributed game in the Internet: multicast, real-time services, and security functions are required in one distributed application, even expected from a single distributed object. None of these services are supported by TCP and are impossible to be realized efficiently on top of TCP.

D.2 Generating Software for Distributed Objects

Our goal is to exploit the flexible structure imposed by the PITOU framework to generate protocol software adapted to the needs of a distributed application (see also [78]).

The architecture we propose consists of two components: 1) a set of tools to support the development of distributed objects, and 2) a distribution infrastructure that allows clients to look-up and access distributed services. The Jini [48],[149] technology would be a good candidate for such an infrastructure. Jini provides a dynamic registration service, a distributed object look-up service, and a set of other useful services to support distributed applications. Amongst others, Jini allows clients to down-load and execute Java code.

Our work focuses on the tools to support automatic generation of software. We first present a tool that allows to map methods including parameters and return-type as protocol environments. We then show how to generate code to allow an application to transparently access these protocol environments via proxy objects.

Figure D.1 gives an overview of all the tools we use and how they are applied.

D.3 Generating Protocol Code

The first tool we provide is a tool to generate protocol code based on the specification of a Java `interface` – one for the client and one for the server. We call these kind of protocols generated **representation protocol**s, since its only goal is to provide a structure that represents all elements of the `interface` specification. Representation protocols do not specify any protocol function.

A Java `interface` specification consists of a set of method declarations. A method declaration in Java basically consists of a method name, a set of parameter types and names, and a return value. They look like this: `<return type> <method name>(<parameter type 1> <parameter name 1>,<parameter type 2> <parameter name 2>, ...)`. An example for a method declaration is `int add(int a, int b)`.

For the client representation protocol, a Java `interface` is mapped to a PITOU protocol configuration by applying the following rules:

- each method declared is represented by one output-order-type object and one input-order-

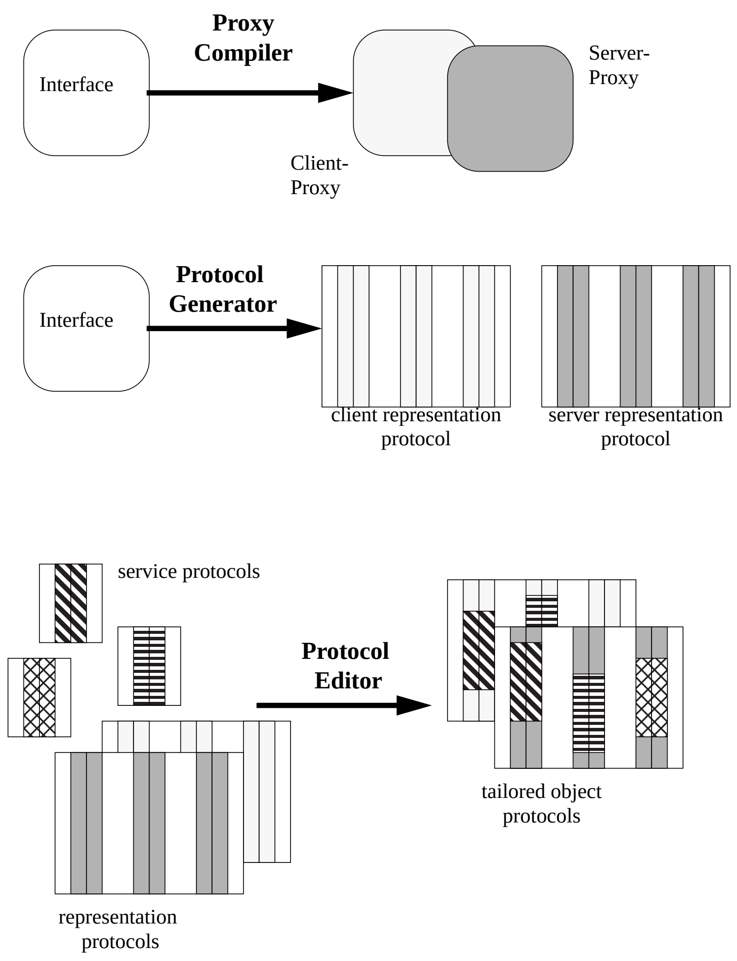


Figure D.1: Generation tools

type object,

- the order-type object gets the name of the method via `OrderType.setName()`,
- the order-type objects are numbered consecutively (`setID()`); this numbering is important since we use the `DefaultMappingStrategy`, which uses the ID for mapping to order-type objects,

• each parameter type of a method is represented by an entry-type object of the corresponding type (e.g. `int` is mapped to `IntegerEntryType`), which is registered with the respective output-order-type object,

- all entry-types are configured visible (since the information is sent) and initializable (since the information is obtained from the application),
- all entry-types get the name of the parameter type via `EntryType.setName()`,

  - the byte length (`setSize()`) corresponds to the type of the parameter (i.e. for integer types `int`, the size in bytes is set to 4),

- the return type of each method is represented by an entry-type object, which is registered with the respective input-order-type object.

Figure D.2 shows the client protocol code generated based on a Java `interface`. The figure depicts the code for a method declared as `int add(int a, int b).`

For the server representation protocol, the rules above are applied in a reciprocal manner, i.e.

- the parameter types are mapped to entry-types registered with an *input*-order-type,
- the return parameter type is mapped to an entry-type registered with an *output*-order-type,
- the ID of an output-order-protocol corresponds to the ID of the respective input-order-type of the client protocol.

Once the representation protocol is generated, the PITOU protocol editor can be used to combine the representation protocol with specific service protocols. Since each method has its own data-path independent from the other methods it is possible to associate each method with workers that assure different QoS characteristics. One method could be based on reliable transfer, another one based on real-time semantics, a third one based on multicast transmission.

The application developer is responsible to choose the appropriate workers to be added for each method, and to connect them manually. However, it is also imaginable to define QoS categories (reliability, latency, security constraints), and allow to specify the needed QoS category for each method by a language extension. Each QoS category would then be represented by a predefined set of workers and relations that are plugged with the representation protocol by a special tool.

D.4 Generating Proxies

Now that the representation protocol is generated and integrated with the service protocol components, the access to the protocol must be done in a transparent manner for the application

On the client side, the application should invoke methods on an object without being concerned if this object resides on the local or on a remote host. On the server side, a remote object should be able to operate as a local object as well, and thus should not be involved in distribution aspects. On both sides, so called **proxy** objects act as place-holders for the object, which is remotely accessible. According to wide-spread terminology in middleware systems, we call the client proxy a **stub** and the server proxy a **skeleton**. In contrary to stub and skeletons e.g. in CORBA, our stub and skeleton do not perform parameter marshalling (this is already done by

```
   public void init() throws ProtocolConstructionException {
               Environment env=getEnvironment();
               //ORDER DEFINITIONS
               OrderRegistrar orderReg=new OrderRegistrar();
               env.setOrderRegistrar(orderReg);
               .....
               // Definition of the order-type add_callOut
               OutputOrderType O7_addc=new OutputOrderType();
               orderReg.addOrder(O7_addc);
               StructureRegistrar r_O7_addc=new StructureRegistrar();
               O7_addc.setStructureRegistrar(r_O7_addc);
               O7_addc.setName("add_callOut");
               O7_addc.setPriority(5);
               O7_addc.setNrOutputSAP(0);
               O7_addc.setOrderID(4);
               //Define all ENTRYs
               probeans.entries.IntegerEntryType O7_addc_E6_____=new probeans.entries.IntegerEntryType();
               r_O7_addc.addEntry(O7_addc_E6_____);
               O7_addc_E6_____.setVisibleFlag(true);
               O7_addc_E6_____.setInitFlag(true);
               O7_addc_E6_____.setSize(4);
               probeans.entries.IntegerEntryType O7_addc_E7_____=new probeans.entries.IntegerEntryType();
               r_O7_addc.addEntry(O7_addc_E7_____);
               O7_addc_E7_____.setVisibleFlag(true);
               O7_addc_E7_____.setInitFlag(true);
               O7_addc_E7_____.setSize(4);
               //Define all PARAMETER-relations

               // Definition of the order-type add_respIn
               InputOrderType O8_addr=new InputOrderType();
               orderReg.addOrder(O8_addr);
               StructureRegistrar r_O8_addr=new StructureRegistrar();
               O8_addr.setStructureRegistrar(r_O8_addr);
               O8_addr.setName("add_respIn");
               O8_addr.setPriority(5);
               O8_addr.setNrInputSAP(0);
               O8_addr.setOrderID(11);
               //Define all ENTRYs
               probeans.entries.IntegerEntryType O8_addr_E8_____=new probeans.entries.IntegerEntryType();
               r_O8_addr.addEntry(O8_addr_E8_____);
               O8_addr_E8_____.setVisibleFlag(true);
               O8_addr_E8_____.setInitFlag(true);
               O8_addr_E8_____.setSize(4);
               .....
               return env;
   }
```

Figure D.2: Generated Client Protocol Code

the representation protocol), but only manage access to the protocol environment and method invocation.

The client stub must implement the Java `interface` that defines the type of the remotely accessible object. But instead of implementing functionality, the stub obtains a reference to an environment and uses the `WriteAPI` interface of the corresponding output-order-type to give the parameter information to the environment, which finally sends it to the server. The stub method also blocks until it receives an answer from the server, i.e. until the input-order with the corresponding return type has been executed and delivered the return type value. The client stub implements the `ReadAPI` interface to be notified about the delivery of return type values. In the `ReadAPI.delivery()` method the according method is unlocked and provided with the return type value. The method then casts the return type value from `Object` representation into a concrete type and returns it to the client application.

```
CLIENT-STUB:
public int add(int p1, int p2) {
int methodNr=4;
try {
Object[] param={new Integer(p1), new Integer(p2)};
allWriteAPIs[methodNr-1].accept(param);
try {
synchronized(blocking[methodNr-1]) {
blocking[methodNr-1].wait();
}
}
catch(Exception e) {
throw new RuntimeException("remote error calling add(int p1, int p2)");
}
return ((Integer)results[methodNr-1]).intValue();
}
catch(Exception e) {
throw new RuntimeException("local system error in add(int p1, int p2)");
}
   }

public void deliver(Object[] o, Deliverable d) {
try {
int i=((InputOrder)d).getOrderID()-numberOfMethods;
if (o.length!=0)
   results[i-1]=o[0]; //set return value
synchronized(blocking[i-1]) {
blocking[i-1].notify();
}
}
catch(Exception e) {
throw new RuntimeException("incoming information is unusable");
}
   }
```

Figure D.3: Generated Code of Client Stub

The server skeleton also implements the `ReadAPI` interface to be notified about the arrival of method invocation requests and their parameters. A skeleton is instantiated with a reference to the remotely accessible object. In the `ReadAPI.delivery()` method, it identifies the method to be called, casts the parameters from `Object` representation into the concrete type, and invokes the method. It then transforms the return type value of the method call into an `Object` and gives it to the environment, which sends it back to the client.

Client stub and server skeleton are generated automatically using the PITOU proxy generator. Instead of explaining in detail how the proxy generator works, we show the code generated based on the example interface given above in Figure D.3 for the client and in Figure D.4 for the server.

```
public void deliver(Object[] o, Deliverable d) {
                try {
                        int i=((InputOrder)d).getOrderID();
                        Address a=((InputOrder)d).getAddress();
                        switch (i) {
                                case 1: {
                                   callMethod1(o,a);
                                   break;
                                }
                                case 2: {
                                   callMethod2(o,a);
                                   break;
                                }
                                case 3: {
                                   callMethod3(o,a);
                                   break;
                                }
                                case 4: {
                                   callMethod4(o,a);
                                   break;
                                }
                        }
                }
                catch(Exception e) {
                        throw new RuntimeException("incoming information is unusable");
                }
   }

private void callMethod4(Object[] o, Address a) {
                //code for add
                int methodNr=4;
                try {
                        int p1=((Integer)o[0]).intValue();
                        int p2=((Integer)o[1]).intValue();
                        Object[] ret={ new Integer(myObj.add(p1, p2)) };
                        allWriteAPIs[methodNr-1].accept(ret,a);
                }
                catch(Exception e) {
                        throw new RuntimeException("local system error in add(p1, p2)");
                }
   }
```

Figure D.4: Generated Code of Server Skeleton

D.5 Scenario

Based on the example of Figure D.3, we show how a client application can call the `add()` method of a remote object. We assume that the client is already connected with the client (e.g. via UDP sockets), and that no congestion or corruption disturb the communication.

1. the client invokes the `add()` method on the stub
2. within `add(int,int)` the stub transforms the parameters from integer type into `Object` type, activates a blocking metaphor, and gives the two parameters as an array of two `Objects` to the `WriteAPI` object that stands for the order-type object representing the `add()` method
3. the data is processed within the environment and sent to the server
4. the server environment processes the incoming data and delivers the information (i.e. the parameter values) via `deliver()` to the server skeleton object
5. the skeleton identifies the method `add()` to be called using the order-ID of the delivery, casts the `Objects` to values of type `integer`, and calls the method `add()` of the server object
6. the server object performs `add()`, i.e. he adds the two parameter values and returns the result
7. the skeleton object casts the result into an `Object` object and gives it to the `WriteAPI` object responsible to create an output-order of the type supposed to transport return values for the `add()` method
8. the data is processed within the environment and send to the client
9. the client environment processes the incoming data and delivers the information (i.e. the return type value) via `deliver()` to the client stub
10. the client stub casts the result and writes it into a special variable; it then unlocks the metaphor, which allows the stub's `add()` method to return
11. the client application obtains the result without knowing the entity that calculated it

An illustration of how the various parts interact is given in Figure D.5.

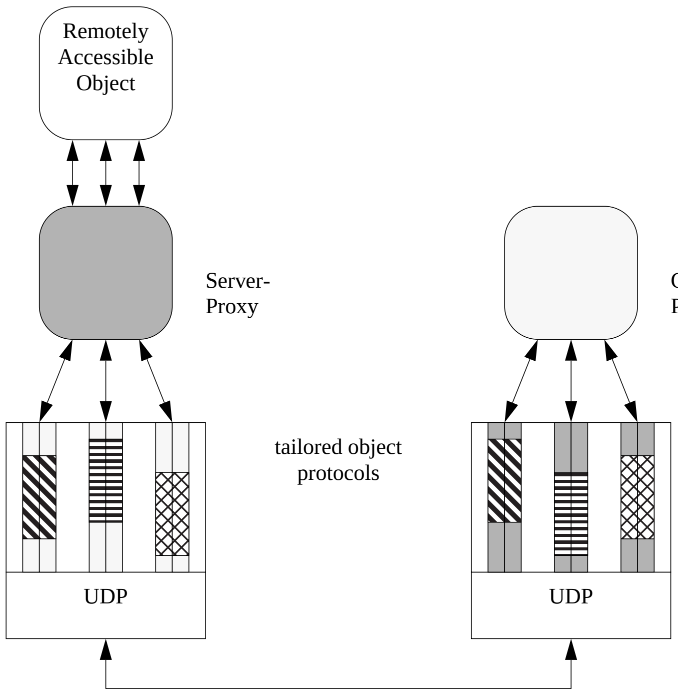


Figure D.5: System Integration

D.6 How to build a distributed application

Now we can put all pieces together. Building a distributed application comprises the following steps:

1. the application developer writes a Java `interface`, which declares all methods that a distributed object is supposed to implement

2. the application developer implements a class considered to be remotely accessible (based on the defined `interface`)

3. a tool called **representation-protocol-generator** uses a Java `interface` as input to produce code that constructs a protocol environment for the PITOU framework, i.e. all declared methods are represented as order-type objects, and all parameter types of a method are represented as entry-type objects.

4. a tool called **proxy generator** generates code that serves as proxy between application (server or client) and the protocol code

5. the application developer uses the PITOU protocol-editor to combine the produced representation protocol with protocol components required by the application (e.g. real-time, multicast-management, admission control)

6. by using the generated proxies, a distributed object can easily and transparently be integrated in the client application

D.7 Conclusions

This section demonstrates the flexibility of our structuring approach and its implementation in the PITOU framework. We make it possible to generate protocol software based on Java `interface` specifications. Our approach – QoS for middleware by generating tailored protocols [78] – contributes to an important research area by offering a new solution. The tools we provide are the first step into the direction of a QoS aware middleware systems.

Appendix E

Curriculum Vitae

Personal Details

Matthias Jung
27, Chemin des Parettes
06130 Grasse-Plascassier, France
E-Mail: jung@eurecom.fr
Telephone: +33 6 14 89 13 89

Birthday:	May 2, 1969
Place of Birth:	Frankfurt (Germany)
Nationality:	German
Marital Status:	Single

Education

Nov 1997 –	PhD student at Institut Eurécom (Corporate Communications Department) under the supervision of Prof. Ernst W. Biersack and supported by a Siemens Doctoral Scholarship.
Oct 1990 – Sep 1997	*Diplom-Wirtschaftsinformatiker* (Master Degree on Business Oriented Computer Science) from the University of Mannheim (Germany).
Sep 1996 – June 1997	*Diplôme des Etudes Approfondies – D.E.A* (Master Degree) on Computer Networks and Distributed Systems from the Ecole Supérieure en Sciences Informatiques (E.S.S.I.), Sophia Antipolis (France).

Training

Nov 1997	– Mar 1999	Project ”Network Protocols in Java” with Siemens München, Germany
Jan 1994	– Apr 1995	Internship at IBM, ENC Heidelberg, Germany
Jul 1990	– Aug 1990	Placement at Bosch Telekom, Frankfurt, Germany
Mar 1990	– Apr 1990	Placement at Dusseldorf Trade Shows, New York, USA
Aug 1988	– Feb 1990	Civil Service at the German Red Cross Organization, Bad Nauheim, Germany

Additional Information

Languages:	German (native), English (fluent), French (fluent), Spanish (basics)
Hobbies:	Literature, Playing Piano and Saxophone, Playing Inline Hockey and Table Tennis

Selected Publications

- M. Jung and E.W. Biersack. **Order-Worker-Entry: A System of Patterns to Structure Communication Protocol Software**. In *Proc. of EuroPLoP'00, Irsee, Germany*, July 2000.

- M. Jung and E.W. Biersack. **How Layering Protocol Software violates Separation of Concerns**. In *ECOOP Workshop on Aspects and Dimensions of Concerns*, Cannes, France, June 2000.

- M. Jung and E.W. Biersack. **QoS for Distributed Objects by Generating Tailored Protocols**. In *ECOOP Workshop on QoS in Distributed Object Systems, Cannes*, France, June 2000.

- M. Jung and E.W. Biersack. **A Component-Based Architecture for Software Communication Systems**. In *Proceedings of IEEE ECBS'00*, Edinburgh, Scotland, April 2000.

- M. Jung, E.W. Biersack, and A. Pilger. **Implementing Network Protocols in Java - A Framework for Rapid Prototyping**. In *Proceedings of ICEIS'99*, Setubal, Portugal, March 1999.

Bibliography

[1] “The Java Media Framework (JMF)”, http://java.sun.com/products/java-media/jmf/.

[2] “Microsoft Foundation Classes (MFC)”, Microsoft Homepage: http://www.microsoft.com.

[3] “RealPlayer Homepage”, www.real.com.

[4] P. Ackermann, *Developing Object-Oriented Multimedia Software - Based on the MET++ Application Framework*, dpunkt Verlag, Heidelberg, 1996.

[5] C. Alexander, *A Pattern Language*, New York, Oxford Univ. Press, 1977.

[6] P. Amer and D. New, “Protocol Visualization in Estelle”, *Computer Networks and ISDN Systems*, 25(7):741–760, February 1993.

[7] P. Amer, T. Connolly, P. Conrad, C. Chassot, and M. Diaz, “Partial Order Transport Service for Multimedia and Other Applications”, *IEEE/ACM Transactions on Networking*, 5(2):440–456, October 1994.

[8] L. Apvrille, L. Dairane, L. Rojas, P. Senac, and M. Diaz, “Implementing a User Level Multimedia Transport Protocol in Java”, LAAS Report 99494, Laboratoire d’Analyse et d’Architecture des Systèmes, December 1999.

[9] J. Arjona and G. Roberts, “Architectural Patterns for Parallel Programming”, In *Proceedings of EuroPLoP’98*, Irsee, Germany, July.

[10] M. Atkins, “Experiments in SR with Different Upcall Program Structures”, *ACM Transactions on Computer Systems*, 6(4):365–392, 1988.

[11] W. Atwood, “Concurrency in Operating Systems”, *Transaction on Computer Systems*, 9(10):18–26, 1976.

[12] C. Becker and K. Geihs, “MAQS - Management for Adaptive QoS-enabled Services”, In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Service*, San Francisco, USA, 1997.

[13] M. Ben-Ari, *Principles of Concurrent Programming,* Englewood Cliffs, 1982.

[14] N. Bhatti, *A System for Constructing Configurable High-Level Protocols*, Ph.D. Thesis, University of Arizona, 1996.

[15] N. Bhatti and R. Schlichting, "A System for Constructing High-Level Protocols", In *ACM SIGCOMM Symposium*, pp. 138–150, August 1995.

[16] A. Bhushan, B. Braden, W. Crowther, E. Harslem, J. Heafner, A. McKenzie, J. Melvin, B. Sundberg, D. Watson, and J. White, "File Transfer Protocol", RFC 172, June 1971.

[17] E. W. Biersack, "Network Simulation using NetSim", In *Proceedings Bellcore Symposium on Performance Modeling,* pp. 159–168, Livingston, NJ, May 1990.

[18] E. W. Biersack, E. Rütsche, and T. Unterschütz, "Demultiplexing on the Adapter, Experiments with Internet Protocols over ATM", In *First International Workshop on High Performance Protocol Architectures,* INRIA, Sophia Antipolis, December 1994.

[19] K. Birman, *Building Secure and Reliable Network Applications,* Prentice Hall, January 1997.

[20] A. Birrel, "An Introduction to Programming with Threads", Technical Report, Digital Systems Research Center, 1989.

[21] E. Birrer, "Frameworks in the Financial Engineering Domain", In *Proceedings of ECOOP*, Springer-Verlag, 1993.

[22] S. Boecking, V. Seidel, and P. Vindeby, "CHANNELS. A Run-Time System for Multimedia Protocols", 1995.

[23] B. Boehm, "A Spiral Model of Software Development and Enhancement", In *ACM Sigsoft, Software Engineering Notes,* volume 11, pp. 22–42.

[24] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS", In C. A. V. Peter H.J. van Eijk and E. Michel Diaz, editors, *The formal description language LOTOS*, pp. 23–73, 1989.

[25] G. Booch, *Object-Oriented Analysis and Design*, Benjamin/Cummings Publishing, 1994.

[26] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide,* Object Technology Series, Addison-Wesley, 1999.

[27] F. Booussinot and R. DeSimone, "The ESTEREL Language", *Proceedings of the IEEE,* 79(9):1293–1304, 1991.

[28] R. Braden, “Computing the Internet Checksum”, Request for Comments RFC 1071, ISI, September 1988.

[29] T. Braun and C. Diot, “Protocol Implementation Using ILP”, In *Proceedings of ACM SIGCOMM’95*, pp. 151–161, 1995.

[30] T. Braun, C. Diot, A. Hoglander, and V. Roca, “An Experimental User Level Implementation of TCP”, Technical Report, INRIA, 1995.

[31] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A Systems of Patterns*, John Wiley & Sons, 1996.

[32] C. Castelluccia, I. Chrisment, W. Dabbous, C. Diot, C. Huitema, E. Siegel, and R. de Simone, “Tailored Protocol Development Using ESTEREL”, Technical Report TR 2374, INRIA, October 1994.

[33] V. G. Cerf and R. E. Kahn, “A protocol for packet network intercommunication”, *IEEE Transactions on Communications*, COM-22(5):637–648, May 1974.

[34] D. Cheriton, “The V Kernel: A Software Base for a Distributed System”, *IEEE Software*, pp. 19–42, 1984.

[35] G. Chesson et al., “XTP Protocol Definition”, Technical Report PEI-90-120, Protocol Engines Inc., Santa Barbara, CA, September 1990.

[36] I. Chrisment, D. Kaplan, and C. Diot, “An ALF Communication Architecture: Design and Automated Implementation”, *IEEE JSAC, Special Issue on Protocol Architectures for the 21st century Applications*, 16(3), April 1998.

[37] P. Ciarfella, L. Moser, P. Melliar-Smith, and D. Agarwal, “The Totem Protocol Development Environment”, In *Proceedings of Infocom’94*, pp. 168–177, Toronto, Ontario, Canada, 1994, IEEE.

[38] D. D. Clark and D. L. Tennenhouse, “Architectural Considerations for a New Generation of Protocols”, In *Proc. ACM SIGCOMM 90*, pp. 200–208, Phildelphia, PA, September 1990.

[39] D. D. Clark, “The Structuring of Systems Using Upcalls”, In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pp. 171–180, Oakland, CA, December 1985.

[40] D. Cohen and J. B. Postel, “On Protocol Multiplexing”, In *Proc. 6th Data Communication Symposium*, pp. 75–81, Pacific Grove, CA, November 1979.

[41] D. Connolly, *XML – Principles, Tools, and Techniques*, O’Reilly, 1997.

[42] G. H. Cooper, “An Argument for Soft Layering of Protocols”, Technical Report MIT/LCS/TR-300, MIT, May 1983.

[43] J. Crowcroft, I. Wakeman, and Z. Wang, “Is Layering Harmful?”, *IEEE Network*, 6(1), January 1992.

[44] A. Delarue and E. Fernandez, “Extension and Java Implementation of the Reactor-Acceptor-Connector Pattern Combination”, In *Proceedings of PLoP’99*, Monticello, Illinois, USA.

[45] E. Dijkstra, “The Structure of the THE-Multiprogramming System”, *Communications of the ACM*, 11(5), May 1968.

[46] D’Souza, D. Francis, Wills, and A. Cameron, *Objects, Components and Frameworks with UML : The Catalysis Approach*, Addison-Wesley, 1998.

[47] A. Edwards et al., “User-Space Protocols Deliver High Performance to Applications on Low Cost Gb/s LANs”, In *Proc. SIGCOMM 94*, pp. 14–23, London, England, September 1994.

[48] W. Edwards, *Core JINI*, Sun Microsystems Press, 1999.

[49] A. El Saddik, C. Seeberg, A. Steinacker, K. Reichenberger, S. Fischer, and R. Steinmetz, “Component-based Construction Kit for Algorithmic Visualizations”, In *Proceedings of IDPT, Kusadasi, Turkey*, June 1999.

[50] R. Englander, *Developing Java Beans*, O’Reilly & Associates, June 1997.

[51] H. Eriksson, “MBONE: The Multicast Backbone”, *Communications of the ACM*, 37(8):54–60, August 1994.

[52] M. E. Fayad and D. C. Schmidt, “Object–Oriented Application Frameworks”, *Communications of the ACM*, 40(10):32–38, October 1997.

[53] D. C. Feldmeier, “Multiplexing Issues in Communication System Design”, In *Proc. ACM SIGCOMM 90*, pp. 209–219, Philadelphia, PA, September 1990.

[54] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, et al., “RFC 2068: Hypertext Transfer Protocol — HTTP/1.1”, January 1997.

[55] D. Flanagan, *Java in a Nutshell*, O’Reilly & Associates, 2. edition, May 1997.

[56] S. Fosse-Parisis, “FreePhone”, http://www-sop.inria.fr/rodeo/fphone/index.html.

[57] M. Fournier, C. Chassot, M. Diaz, and A. Lozes, "Performance Evaluation of Partial Order Connections", LAAS Report 97053, Laboratoire d'Analyse et d'Architecture des Systèmes, February 1997.

[58] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[59] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing, 1994.

[60] B. Garbinato and R. Guerraoui, "Flexible Protocol Composition in Bast", In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pp. 22–29, Amsterdam, The Netherlands, May 1998, IEEE Computer Society Press.

[61] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification", White Paper, Sun Microsystems, 1996, Version 1.0.

[62] S. Haslbeck, "Structuring and Implementing a TCP/IP Protocol Stack with JChannels", M.S. Thesis, University of Munich/Institut Eurécom, Sophia Antipolis, France, November 1998.

[63] M. G. Hayden, *The ENSEMBLE System*, Ph.D. Thesis, Cornell University, 1998.

[64] H. Hüni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software", In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*, ACM Press, 1995.

[65] D. Hoscher and R. Hodges, "SEMATECH'S Experiences with the CIM Framework", *Communications of the ACM*, 40(10), October 1997.

[66] X. Huang, R. Sharma, and S. Keshav, "The ENTRAPID Protocol Development Environment", In *Proceedings of INFOCOM*, New York, USA, 1999.

[67] J. Hummes, A. Kohrs, and B. Merialdo, "Questionnaires: A Framework using Mobile Code for Component-Based Tele-Exams", In *Proceedings of IEEE 7th Intl. Workshops on Enabling Technologies: Infrastructure for Collaborating Enterprises (WET ICE)*, Stanford, CA, USA, June 1998.

[68] N. Hutchinson and L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[69] IBM, "Application Framework for e-business", http://www-4.ibm.com/software/ebusiness/mobile.html.

[70] IBM, "Programming with VisualAge for Java Version 2", IBM Redbook SG24-5264-00, 1998.

[71] International Organization for Standardization, "Information Processing Systems — Open Systems Interconnection — Basic Reference Model", *ISO, ISO 7498*, 1984.

[72] V. Jacobson, "Congestion Avoidance and Control", In *Proc. of ACM SIGCOMM'88*, pp. 314–329, Stanford, CA, August 1988.

[73] V. Jacobson, "4BSD TCP Header Prediction", *Computer Communication Review*, 20(2):13–15, April 1990.

[74] V. Jacobson, "Modified TCP Congestion Avoidance Algorithm", end2end-interest mailing list, April 30 1990.

[75] JavaSoft, *Java Beans 1.0 API specification*, October 1996.

[76] R. Johnson and B. Woolf, "The Type Object Pattern", http://www.ksccary.com/Articles/TypeObjectPattern.html, 1997.

[77] R. E. Johnson, "Frameworks = (Components + Patterns)", *Communications of the ACM*, 40(10):39–42, October 1997.

[78] M. Jung and E. W. Biersack, "QoS for Distributed Objects by Generating Tailored Protocols", In *ECOOP Workshop on QoS in Distributed Object Systems*, Cannes, France, June 2000.

[79] M. Jung, E. W. Biersack, and A. Pilger, "Implementing Network Protocols in Java - A Framework for Rapid Prototyping", In *Proceedings of ICEIS*, pp. 649–656, Setubal, Portugal, March 1999.

[80] M. Jung, M.-N. Sauvayre, and E. W. Biersack, "ProJava - A Protocol Developers' Environment for Java", Internal Report, Institut Eurecom, April 1998.

[81] D. Kaplan and E. W. Biersack, "Benchmark of JChannels based Protocol Implementations", Internal Report, Institut Eurecom, April 1999.

[82] P. Karn and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", In *Proc. ACM SIGCOMM 87*, pp. 2–7, Stowe, VT, August 1987.

[83] S. Keshav, *An Engineering Approach to Computer Networking*, Prentice Hall, 1st edition, 1997.

[84] D. Kiely, "Are Components the Future of Software?", *IEEE Computer*, pp. 10–11, February 1998.

[85] F. Kon, A. Singhai, R. H. Campbell, D. Carvalho, R. Moore, and F. J. Ballesteros, "2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments", In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[86] R. Koster and T. Kramp, "Structuring QoS-Supporting Services with Smart Proxies", In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, Hudson River Valley, USA, April 2000.

[87] D. Krieger and R. M. Adler, "The Emergence of Distributed Component Platforms", *IEEE Computer*, pp. 43–53, March 1998.

[88] B. Krupczak, K. Calvert, and M. Ammar, "Implementing Protocols in Java: The Price of Portability", In *IEEE Infocom '98*, San Francisco, USA, April 1998.

[89] G. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, 1998.

[90] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publ., Reading, MA, 1989.

[91] C. Maeda and B. Bershad, "Networking Performance for Microkernels", In *In Proceedings of th Third Workshop on Workstation Operating Systems*, pp. 154–159, April 1992.

[92] S. Maffeis, "Adding Group Communication and Fault Tolerance to CORBA", In *Proceedings of USENIX Conference on OO Technologies*, Monterey, CA, June 1995.

[93] S. Maffeis, "iBus - The Java Intranet Software Bus", White Paper, SoftWired AG (www.softwired.ch), Zurich, Switzerland, February 1997.

[94] R. Maralsli, P. D. Amer, and P. T. Conrad, "Retransmission-Based Partially Reliable Transport Services: An Analytic Model", In *Proc. IEEE INFOCOM'96*, pp. 621–629, San Francisco, CA, April 1996.

[95] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", In *1993 Winter USENIX*, San Diego, CA, January 1993.

[96] R. Mecklenburg, C. Burger, and K. Rothermel, "Visualized Interactive Protocols: A Way to Teach and Test Rules of Computer Communication", In *Fourth World Conference on Integrated Design and Process Technology*, Kusadasi, Turkey, 1999.

[97] I. Mitrani, *Simulation Techniques for Discrete Event Systems*, Cambridge University Press, 1982.

[98] J. Mogul, “The Case for Persistent-Connection HTTP”, In *Proceedings of SIGCOMM’95*, pp. 299–314, September 1995.

[99] J. Mogul, R. Rashid, and M. Accetta, “The Packet Filter: An efficient mechanism for User-Level Network Code”, In *In Proceedings of the 11th ACM Symposium on Operating System Principles*, pp. 39–51, November 1987.

[100] D. Mosberger and L. Peterson, “Making Paths Explicit in the Scout Operating System”, In *Proceedings of OSDI*, pp. 153–168, October 1996.

[101] L. Moser, P. Melliar-Smith, P. Narasimham, L. Tewksbury, and V. Kalogeraki, “The Eternal System: An Architecture for Enterprise Applications”, In *3rd International Enterprise Distributed Object Computing Conference (EDOC)*, University of Mannheim, Germany, September 1999.

[102] S. C. Murphy, P. Gunningberg, and J. P. J. Kelly, “Experiences with Estelle, LOTOS and SDL: A Protocol Implementation Experiment”, *Computer Networks and ISDN Systems*, 22:51–59, 1991.

[103] J. Nagle, “Congestion control in TCP/IP internetworks”, *Computer Communication Review*, 14(4):11–17, October 1984.

[104] NS, UCB/LBNL/VINT Network Simulator - ns (version 2), http://www.isi.edu/nsnam/ns.

[105] ObjectSpace Inc., Dallas, Texas, *ObjectSpace Voyager. Version 2.0.0 User Guide*, 1998, http://www.objectspace.com/developers/voyager/white/index.html.

[106] S. W. O’Malley and L. L. Peterson, “A Dynamic Network Architecture”, *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[107] R. J. Pooley, *An Introduction to Programming in SIMULA*, Blackwell, 1987.

[108] J. Postel, “NCP/TCP Transition Plan”, Request for Comments (Informational) RFC 801, Internet Engineering Task Force, November 1981.

[109] J. Postel, “Transmission Control Protocol – Protocol Specification”, Request for Comments (Standard) RFC 793, Information Sciences Institute, USC, September 1981.

[110] J. Postel, “User Datagram Protocol – Protocol Specification”, Request for Comments (Standard) RFC 768, Information Sciences Institute, USC, August 1981.

[111] W. Pree, “Meta Patterns—A Means for Capturing the Essentials of Reusable Object–Oriented Design”, In *Proceedings of ECOOP’94*, Bologna, Italy, September 1994.

[112] J. Pruyne, “Enabling QoS via Interception in Middleware”, Submitted, Hewlett-Packard Labatories, 2000.

[113] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, “Machine-Independent Virtual Memory Management for Pages Uniprocessor and Multiprocessor Architecture”, Technical Report, Carnegie-Mellon University, 1987.

[114] L. Rising, “Patterns: A Way to Reuse Expertise”, *IEEE Communications Magazine*, 37(4):34–36, April 1999.

[115] D. Roberts and R. Johnson, “Evolving Frameworks – A Pattern Language for Developing Object-Oriented Frameworks”, In *Proceedings of PLoP*, Monticello, Illinois, September 1998.

[116] V. Roca, T. Braun, and C. Diot, “Demultiplexed Architectures: A Solution for Efficient STREAMS based Communication Stacks”, *IEEE Networks Magazine*, June 1997.

[117] D. Ross, J. Goodenough, and C. Irvine, “Software Engineering: Process, Principles, and Goals”, *IEEE Computer*, 18(5), May 1975.

[118] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[119] A. Sane and R. Campbell, “Composite Messages: A Structural Pattern For Communication between Components”, In *OOPSLA Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, 1995.

[120] H. A. Schmid, “Creating the Architecture of a Manufactoring Framework by Design Patterns”, In *Proceedings of OOPSLA’95*, NY, 1995, ACM.

[121] D. Schmidt, “Patterns for Concurrent, Parallel, and Distributed Systems”, Website: http://www.cs.wustl.edu/ schmidt/patterns-ace.html.

[122] D. Schmidt and F. Buschmann, “Reactor – An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching”, In *Proceedings of PLoP*, Monticello, Illinois, USA, August 1994.

[123] D. Schmidt and F. Buschmann, “Active Object – An Object Behavioral Pattern for Concurrent Programming”, In *Proceedings of PLoP*, Monticello, Illinois, USA, September 1995.

[124] D. Schmidt and F. Buschmann, "Acceptor-Connector – An Object Creational Pattern for Connecting and Initializing Communication Services", In *Proceedings of EuroPLoP*, Kloster Irsee, Germany, July 1996.

[125] D. Schmidt, D. Levine, and T. Harrison, "The Design and Performance of a Real-time CORBA Object Event Service", In *Proceedings of OOPSLA*, Atlanta, Georgia, April 1997.

[126] D. C. Schmidt, "Applying Patterns and Frameworks to develop OO Communications SW", In P. Salus, editor, *Handbook of Proramming Languages*, volume 1, MacMillan, 1997.

[127] D. Schmidt, "An Architectural Overview of the ACE Framework", In P. Salus, editor, *Special Issue of USENIX Login*, 1998.

[128] H. Schulzrinne, S. Casner, R. Frederic, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", Request for Comments RFC 1889, Internet Engineering Task Force, January 1996.

[129] G. Sidhu, R. Andrews, and A. Oppenheimer, *Inside AppleTalk*, Addison-Wesley, 1989.

[130] J. Siegel, *CORBA – Fundmentals and Programming*, John Wiley and Sons, Inc., 1996.

[131] A. Singhai, *Quarterware: A Middleware Toolkit of Software Risc Components*, Ph.D. Thesis, University of Illinois, 1999.

[132] P. Sommerlad, "Configurability", In *Proceedings of EuroPLoP*, Kloster Irsee, Germany, July 1999.

[133] P. Sommerlad and F. Buschmann, "Manager", In *Proceedings of PLoP*, Monticello, Illinois, USA, September 1996.

[134] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", Request for Comments RFC 2001, Internet Engineering Task Force, January 1997.

[135] W. R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, Reading, MA, 1995.

[136] Sun Microsystems, "The Java Virtual Machine Specification", White Paper, 1995.

[137] Sun Microsystems, "Java Homepage", 1999, http://java.sun.com.

[138] Sun Microsystems, "The Java Remote Method Invocation Specification", 1999, http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-spec.ps.

[139] D. L. Tennenhouse, "Layered Multiplexing Considered Harmful", In H. Rudin and R. Williamson, editors, *Proc. IFIP Workshop on Protocols for High-Speed Networks*, pp. 143–148, Zurich, Switzerland, May 1989, North-Holland Publ., Amsterdam, The Netherlands.

[140] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture", *Computer Communication Review*, 26(2):5–18, April 1996.

[141] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska, "Implementing Network Protocol at User Level", *IEEE/ACM Transaction on Networking*, 1(5):554–565, October 1993.

[142] J. Touch, J. Heidemann, and K. Obraczka, "Analysis of HTTP Performance", Technical Report 98-463, USC/ISI, December 1998.

[143] W. Tracz, "RMISE Workshop on Software Reuse Meeting Summary", In I. C. S. Press, editor, *Software Reuse: Emerging Technology*, pp. 41–53, Los Alamitos, CA, 1988.

[144] C. Tschudin, "Flexible Protocol Stacks", In *Proceedings of ACM SIGCOMM*, Zurich, Switzerland, October 1991.

[145] Unix, "STREAMS Programmer's Guide", *Unix System V Release 4*, 1990.

[146] R. Urtasun, "Implementation of a Tool to Visualize Protocol Design and Processing", M.S. Thesis, University of Pamplona/Institut Eurécom, July 2000.

[147] R. van Renesse, "Masking the Overhead of Protocol Layering", In *Proceedings of ACM Sigcomm*, Stanford, CA, USA, September 1996.

[148] R. van Renesse, K. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System", *Communications of the ACM*, 39(4):76–83, April 1996.

[149] J. Waldo, "Jini Technology Architectural Overview", White Paper, SUN, 1998.

[150] S. Yajnik, "DOORS: Fault Tolerance for CORBA Applications", In *FIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, England, September 1998.

[151] K. Yasumoto, T. Higashino, T. Matsuura, and K. Taniguchi, "Protocol Visualization using LOTOS Multi-Rendezvous Mechanism", In *Proceedings of ICNP*, pp. 118–125, November 1995.

[152] H. Zimmerman, "OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection", *IEEE Transactions on Communications*, COM-28(4), April 1980.

[153] J. Zinky, R. Schantz, J. Loyall, K. Anderson, and J. Megquier, “The Quality Objects (QuO) Middleware Framework”, In *RM – Workshop on Reflective Middleware*, New York, USA, April 2000.

Index

Abstract

Résumé

Développer des applications distribuées implique souvent l'implémentation de nouveaux protocoles qui concernent les sémantiques de l'application ainsi que de la couche transport. Cependant, l'implémentation de nouveaux protocoles à partir de zéro est une tâche lourde et coûteuse. L'objectif de notre thèse est le développement des techniques et des outils qui nous aident à minimiser les coûts d'implémentation et de maintenance des protocoles de communication dans les systèmes finaux. Nous nous concentrons sur des techniques modernes du génie logiciel – *charpentes orienté objet*, *motifs de conception* et *développement par assemblage de composants* – qui ont montré récemment leur applicabilité dans différents domaines.

La fondation de notre travail est un ensemble d'abstractions et de principes de structuration qui promeuvent la réutilisation et la flexibilité des logiciels de protocole. L'idée principale derrière ces principes suit une structure verticale au lieu d'une structure en couches. Notre approche intègre tous les services dont l'application a besoin dans une seule entité alors que tout démultiplexage est concentré en dehors de cette entité dans une couche plus basse. Par ailleurs, nous proposons de structurer les protocoles par *chemin de données* et de diviser chaque chemin de données identifié en modules à grain fin, réutilisables et configurables. Basée sur notre approche de structuration, nous présentons une charpente en Java dénommée PITOU qui permet de construire des nouveaux protocoles en assemblant et configurant des composants existants. De plus, nous avons mis en oeuvre quelques outils de support, tels que des outil de simulation et de visualisation, ainsi qu'un générateur de code. L'applicabilité et la flexibilité de notre approche sont démontrées par l'implémentation des services de transport de TCP et par une application qui analyse des données d'une séance de RTP/RTCP.

Abstract

Distributed application developers are often forced to implement with application-specific protocols also transport layer semantics that are incorporated in and delivered with the application code. However, implementing new protocols from scratch is tedious and expensive. The goal of our thesis is the development of techniques and tools that minimize the cost to implement and maintain protocols in end-systems. Our focus is on modern software engineering techniques – like OO frameworks, design-patterns, and component-based development – that have recently shown their applicability in various application domains.

The foundation of our work is a set of structuring principles and abstractions that promote reusability and flexibility of protocol software. The main idea behind these principles is to follow a vertical instead of a horizontal structure (such as layering). Our approach suggests to integrate all application-specific services into one entity and to decouple demultiplexing functionality from protocol processing. We further propose to structure protocol software along all its types of data paths and to partition the identified data paths into fine-grained, reusable, and configurable modules. Based on our structuring approach, we present a Java framework called PITOU that allows to construct new protocols by assembling existing components and configuring them to the specific application requirements. We further implement a number of tools to assist protocol development, such as a tool to simulate protocol sessions, a tool to animate protocol software, and a tool to generate protocol code based on specified structures and configurations. The applicability and expressiveness of our structuring approach is demonstrated by a modular implementation of the TCP data transfer services and a RTP/RTCP snoop application.