Protocol Engineering Concepts and Patterns using UML

Juha Pärssinen
Markku Turunen
Jukka Heinonen
Niklas von Knorring
Arto Kvist
Pekka Jäppinen

25 November 1999
108 pages

Contents

List of Figures

Preface

XXX

Goals

XXX

Audience

XXX

Outline of The Book

XXX

Acknowledgements

Many people helped us with ideas, providing critics, and corrections. Especially we would like to thank Ari Ahtiainen from Nokia Research Center, Juha Koivisto and Timo Kyntäjä from Technical Research Centre of Finland, and professor Olli Martikainen from Helsinki University of Technology.

Part 1 An Introduction to Protocol Engineering

Chapter 1 Introduction

XXX

Chapter 2 Basic Protocol Constructs and Functions

For two entities to successfully communicate, they must “speak the same language”. What is communicated, how it is communicated, and when it is communicated must conform to some mutually acceptable set of conventions between the entities involved. An entity is anything capable of sending or receiving information, and a system is a physically distinct object that contains one or more entities. The set of conventions is referred to as a *protocol*, which may be defined as a set of rules governing the exchange of data between entities [3].

Sections “Protocol Hierarchies” on page 5, and “Protocol Functions” on page 11 are written in accordance with the Open Systems Interconnection (OSI) Reference Model [5].

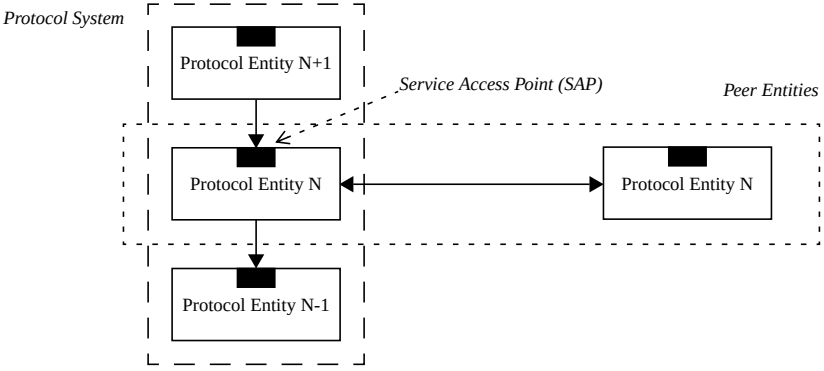

Fig. 1. Protocol Entities as Layers

2.1 Protocol Hierarchies

To reduce communication systems design complexity, most systems are organized as a series of layers or levels, each one built upon its predecessor. The number of layers, the name of each layer, the contents of each layer and the function of each layer differ from system to system. The purpose of each layer is to offer certain services to the higher lay-

ers, shielding those layers from the details of how the offered services are actually implemented.

A layer n entity on one system carries on a conversation with a layer *n peer entity* on another system. The rules and conventions used in this conversation are collectively known as the *layer n protocol* [2].

2.1.1 The Open Systems Interconnection (OSI) Reference Model

The International organization for Standardization (ISO) established in 1977 a subcommittee to develop an architecture to define communications tasks. The result is the *Open Systems Interconnection (OSI) Reference Model*, adopted in 1983. It is a framework for defining standards for linking heterogeneous computers [3].

The logical structure of OSI is made up of seven protocol layers. The three lowest layers (1-3) are network dependent and concerned with the protocols associated with the data communication network being used to link two communicating systems. The three upper layers (5-7) are application oriented and are concerned with the protocols that allow two end-user application processes to interact each other. The intermediate transport layer (4) hides the detailed operation of the lower network-depend layers from the upper application oriented layers [5].

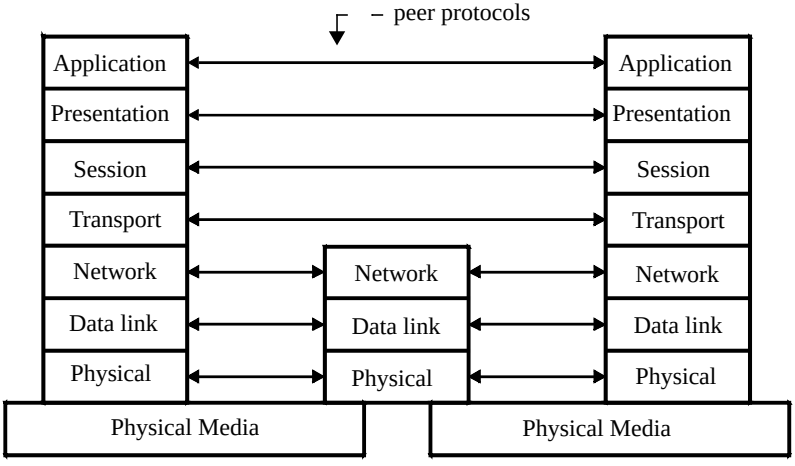

Fig. 2. OSI Reference Model

Application Layer (7). As the highest layer in the Reference Model of Open Systems Interconnection, the Application Layer provides the sole means for the application process to access the OSI environment. Hence the Application Layer has no boundary with a

higher layer. This layer contains management functions and generally useful mechanisms to support distributed applications.

Presentation Layer (6). The Presentation Layer provides for the representation of information that application-entities either communicate or refer to in their communication.

Session Layer (5). The purpose of the Session Layer is to provide the means necessary for cooperating presentation-entities to organize and to synchronize their dialogue and to manage their data exchange. To do this, the Session Layer provides services to establish a session-connection between two presentation-entities, to support orderly data exchange interactions, and to release the connection in an orderly manner.

Transport Layer (4). The transport-service provides transparent transfer of data between session-entities and relieves them from any concern with the detailed way in which reliable and cost effective transfer of data is achieved.

Network Layer (3). The Network Layer provides the functional and procedural means for connectionless-mode or connection-mode transmission among transportentities and, therefore, provides to the transport-entities independence of routing and relay considerations.

The Network Layer provides the means to establish, maintain, and terminate network-connections between open systems containing communicating application-entities and the functional and procedural means to exchange network-service-data-units between transport-entities over network-connections.

Data Link Layer (2). The Data Link Layer provides functional and procedural means for connectionlessmode among network entities, and for connection-mode for the establishment, maintenance, and release data-link-connections among network entities and for the transfer of data-link-service-data-units. A data-link-connection is built upon one or several physical connections. The Data Link Layer detects and possibly corrects errors which may occur in the Physical Layer.

Physical Layer (1). The Physical Layer provides the mechanical, electrical, functional and procedural means to activate, maintain, and de-activate physical-connections for bit transmission between data-link-entities. A physical-connection may involve intermediate open systems, each relaying bit transmission within the Physical Layer. Physical Layer entities are interconnected by means of a physical medium.

2.2 Modeling Protocol Entities

The active elements in each layer are called entities. An entity is anything capable of sending or receiving information between systems, which are physically distinct objects that contains one or more entities. Entities in the same layer on different systems are called peer entities.

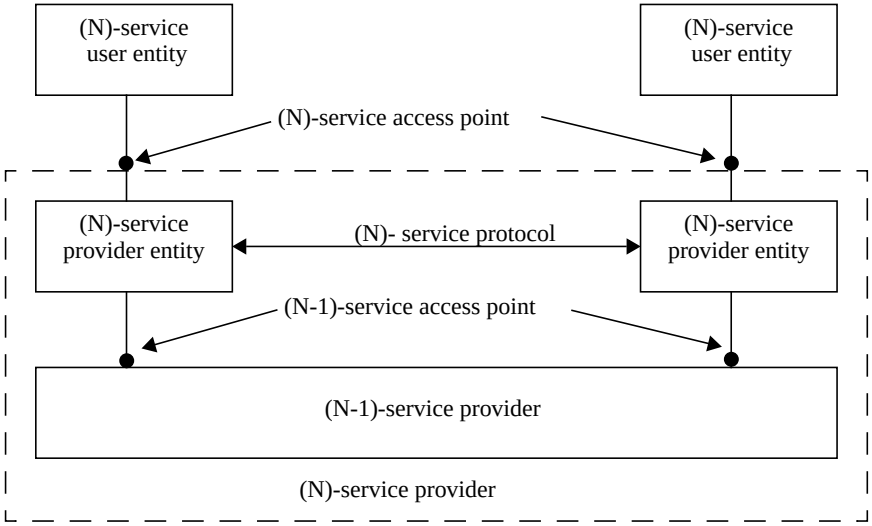

Fig. 3. Modelling the Protocol Layer [1]

2.2.1 Services

The entities in layer (N) implement a service used by layer (N+1). In this case layer (N) is called service provider and layer (N+1) is called service user. Layer (N) may use the services of layer (N-1) in order to provide its service [3] (Figure. 3).

Services are available at service access points (SAP). The layer (N) SAPs are the places where layer (N+1) can access to services (Figure. 3). Each SAP has an address that uniquely identifies it.

2.2.2 Service Primitives

Each service can be characterized by describing the interactions between the service provider and user. A service can be *confirmed*, *unconfirmed*, or *provider initiated* [1].

The Confirmed Service. The confirmed service involves a handshake between the user that requests the service, and the user that is informed that the service has been requested (Figure 4 on page 9). All un-confirmed services consist of four primitives types:

1. .request (.req)
2. .indication (.ind)
3. .response (.rsp)
4. .confirmation (.cnf)

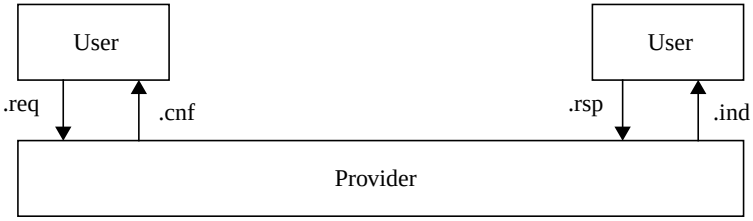

Fig. 4. The Confirmed Service

The Unconfirmed Service. The unconfirmed service involves no handshake between communicating parties (Figure 5 on page 9). Unconfirmed services consist of two primitive types:
1. .request (.req)
2. .indication (.ind)

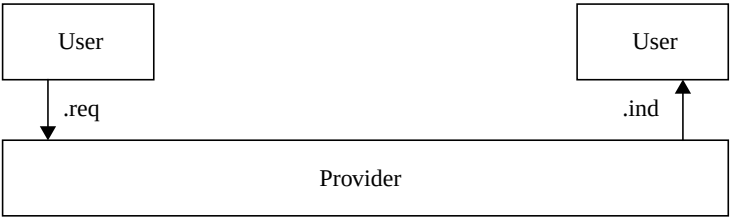

Fig. 5. The Unconfirmed Service

The Provide Initiated Service. The provider initiated service is generated by the service provider in the response to some internal conditions (Figure 6 on page 10). Provider initiated services consists of only one primitive:

1. .indication (.ind)

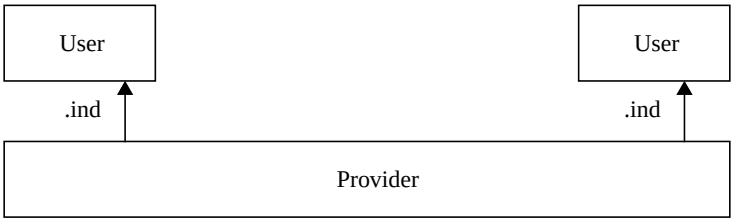

Fig. 6. The Provide Initiated Service

2.2.2.1 Service Interfaces

In order for two layers to exchange information, there has to be an agreement upon set of rules about the interface. At an interface, the layer (N) entity passes an interface data unit (IDU) to the layer (N-1) via (N-1)-service access point (SAP) (Figure. 7). The IDU consists of a data, termed as a protocol data unit (PDU), and a control information, termed as a interface control information (ICI).

The PDU is the unit of information that is exchanged by peer entities. The PDU consists of user-data, termed as a service data unit (SDU), and control information (a header), termed as protocol control information (PCI) (Figure. 7). The PCI identifies the data that is to be transferred [1].

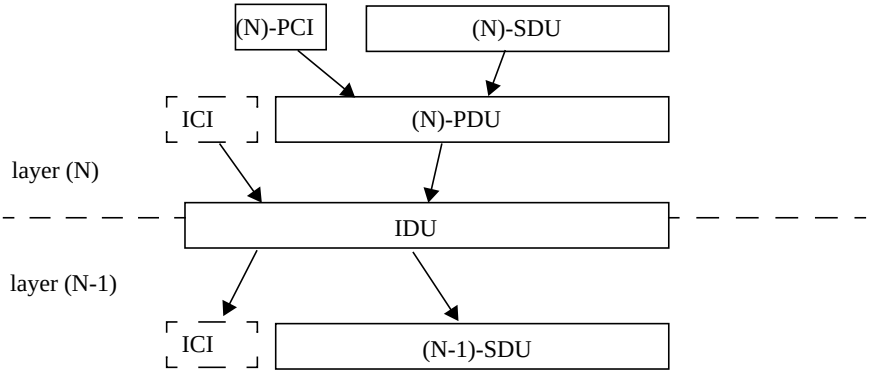

Fig. 7. Interface units

The layer (N-1) service provider receives the IDU and breaks it apart into the ICI and an (N-1)-SDU. It then invokes the desired primitives based on the ICI. Note that from the perspective of the (N-1)-service provider, an (N)-PDU is usually an (N-1)-SDU [1].

In more familiar terms from other protocol suites, the SDU is the data, the PCI is the header, and the PDU is the packet [1].

2.3 Protocol Functions

Protocol functions can be grouped to segmentation and reassembly, encapsulation, connection control, ordered delivery, flow control, error control, synchronization, addressing, multiplexing, and transmission services [3]. Not all protocols have all these functions. Each protocol function may be present in several layers of the system.

Segmentation and reassembly. An application entity sends data in messages or in a continuous stream. Lower level protocols may need to break the data up into blocks of smaller size. This process is called segmentation or fragmentation. The counterpart of segmentation is reassembly or defragmentation. The fragmented data must be reassembled into higher level messages. If fragmented data blocks arrive out of order, the task is complicated. These data blocks are referred to as a protocol data unit (PDU) (“Service Interfaces” on page 10).

Encapsulation. Each PDU contains data and control information. The control information is used for addressing, error detection and protocol control. The addition of control information to data is referred to as encapsulation (Figure 7 on page 10).

Connection control. An entity may transmit data to another entity without planning or co-ordination. This is known as connectionless data transfer. In the connection-oriented data transfer a logical association, or connection, is established between entities. A connection has three phases:
1. Connection establishment
2. Data transfer
3. Connection termination

Ordered delivery. In connection-oriented protocols, it is required that PDU order is maintained. A PDU has a finite sequence number field, and sequence numbers repeat. The maximum number of sequence numbers must be greater than the maximum numbers of PDUs that could be outstanding at any time. This function is usually in the transport layer.

Flow control. Flow control is a function performed by a receiving entity to limit the amount or rate of data that is sent by a transmitting entity. This function is often needed in several protocol levels in the same system. This function is usually in lower layers.

Error control. Error control is needed to guard against loss or damage of data and control information. Most techniques involve error detection and retransmission. Retransmission (usually in lower layers) is often activated by a timer.

Synchronization. Two communicating protocol entities must be simultaneously in a welldefined state, synchronized. The state of an entity is the collection of the values of its variables, e.g. connection phase, timer value.

Addressing. For two entities to communicate they must be able to identify each other. A distinction is made between names, addresses and routes. A name specifies what an object is, an address specifies where it is, and a route tells how to get there.

Multiplexing. Multiplexing can be upward multiplexing, which occurs when multiple higher-level connections are multiplexed on a single lower-level connection, or downward multiplexing, which occurs when a single higher-level is built on top of multiple lower-level connections.

Transmission services. A protocol may provide additional services to the entities that use it, e.g. priority, security.

2.4 Protocol System Structure

The traditional way to illustrate protocol system structure is presented in the Figure 8 on page 12 on page 29. A system, or stack, is built using adjacent entities or layers. These form a protocol stack (Figure. 8.a.), which can have branches (Figure. 8.b.). Layers use interfaces to communicate with each other, and the whole stack has interfaces to the system's environment.

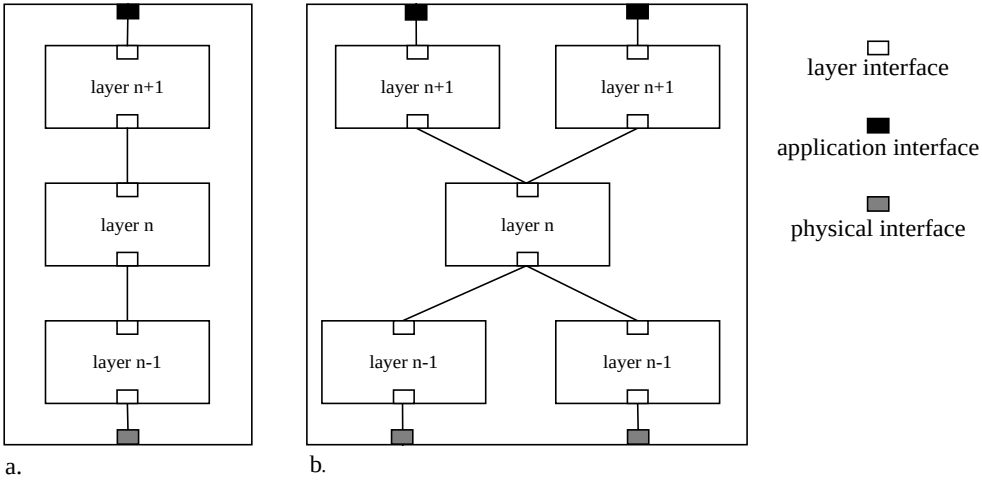

Fig. 8. Traditional picture of protocol structure

2.4.1 Protocol Entity

Active communicating elements in protocol systems are called protocol entities. An entity is capable of sending or receiving information between systems. An entity has a logical connection to its peer entity, and real connection to its adjacent entity or environment. In Figure 12 there are four entities which are grouped in two stacks. These stacks are connected via a physical connection.

An entity has to have capability to communicate with other entities in the same system and with peer entities. It has to have storage for inner states and other information, and in some Entities, it has to have the capability to manage multiple concurrent communication sessions.

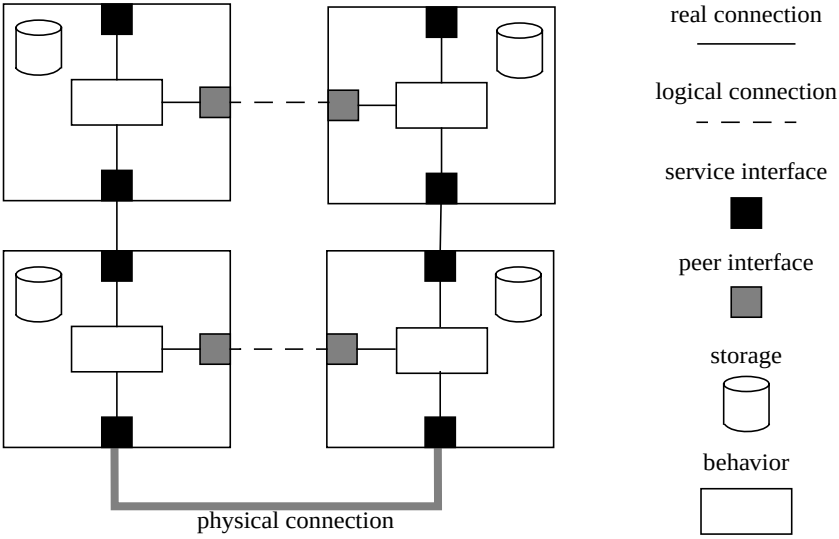

Fig. 9. Traditional Picture of Protocol Entities

2.4.2 Data Flow

A communicating system can be composed of several entities which have to be able to communicate with each other. Communicating entities can be in the same or in a peer system. The conventional way to illustrate data flow within a Protocol Entity is shown in Figure 10 on page 14. In Figure. 10 there are four entities which are grouped in two stacks. These stacks are connected via a physical connection.

A Protocol Entity receives a Entity Message (e.g. a data request) via a Entity Interface. Message parameters are stored into Storage for future use. Protocol Behavior deter-

mines that reception of a Entity Message causes sending of a Peer Message (e.g. a data PDU). Peer Interface produces a Peer Message and sends it to a peer Protocol Entity. Because there is only a logical connection between peer entities the Peer Message must be transmitted within a lower layer Service Message.

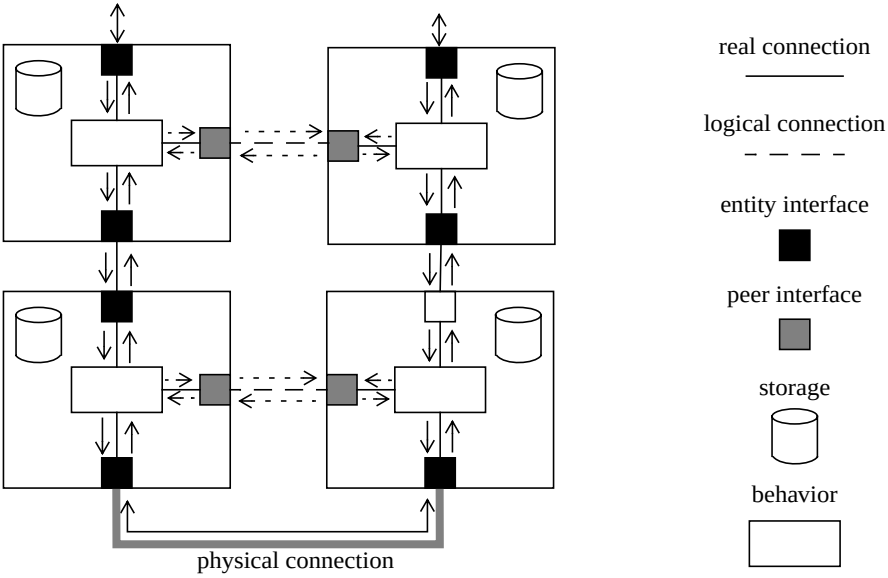

Fig. 10. Data Flow in a protocol system

Chapter 3 Protocol Specifications

XXX

Chapter 4 Protocol Engineering Process

XXX

Chapter 5 Tools for Protocol Implementation

XXX

5.1 Protocol Frameworks in General

A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way of their interaction [11]. In protocol engineering there are certain components that almost always are needed. Implementation of statemachines of protcol entities, communication between protocol entities within the same system (messages, primitives) and between peer entities (PDUs) are common tasks that everyone working with protocols has encountered.

Frameworks offer efficient and simple way to create and configure functionality of these common tasks. Implementation details are hidden behind stable interfaces thus letting the protocol designer to concentrate on analyzing and developing the protocol itself [9].

It would be good if the framework would provide different validation tools that help analyzing the protocol as well as finding the problems in the protocol itself. The framework should also provide testing tools for finding possible bugs in the implementation.

5.1.1 What frameworks offer to the implementation

When we are talking about frameworks in communication protocols it is useful to make a presumption that some basic features are implemented in each framework. Each framework has to offer some prevalent basic mechanisms regardless of its own implementation. In this chapter we'll take a look at those common features thought as necessary and how those are utilized in protocol development. The implementation of these features must be comprehensive enough to satisfy certain demands before they truly can be used in an efficient way. For example the need to write additional code must be minimal in order to the framework to bring real benefit to the development.

The basic features that are seen as necessary for protocol development are implementations of EFSA (Extended Finite State Machine), scheduler, message passing, PDU (Protocol Data Unit) en/decoding, timer and I/O interface. More detailed examination of each of these components is located in this chapter in paragraphs "Message Passing" on page 20 through "PDU en/decoding" on page 21.

Even if we assume that the features presented here are, indeed, implemented comprehensively these basic features won't enable creation of complex protocols without a lot of own coding if no additional framework components are used. The more complete framework feature set is discussed in chapter XXX and in chapter XXX1.

5.1.1.1 Message Passing

The communicating protocol stack consists of several communicating protocol-objects (CPO) which need to communicate somehow with each other. Communication is handled by moving messages from one CPO to another. Frameworks provide many alternatives for message passing. Some frameworks provide internal message queues for passing of message structures. In others messages are passed from one CPO to another by simply moving a pointer, while frameworks relying on design patterns, such as conduit based frameworks, use the Visitor pattern [8].

5.1.1.2 Scheduler

In communicating system several events can happen almost simultaneously, and thus not all the events can be processed right away. The framework has to offer some kind of scheduling in order to resolve this problem. Because we are not dealing with hard real-time systems a non-pre-emptive scheduler is enough for our purposes.

5.1.1.3 Input/Output Interface

A communicating system must be able to communicate with the outside world. Communication can be anything from simple user to user communication in single machine to complex multi machine parallel computing system. Therefore there is a need for an interface to external devices like pipes and sockets. The protocol might also want to use other services, like simply reading or writing a file, provided by the machine. Interface for at least to the filesystem, should also be supported by the framework.

5.1.1.4 State Machine

The behavior of a protocol entity is usually most easily understood when it is presented as a state machine. Frameworks provide varying support for state machines ranging from special EFSA languages, interpreters and translators to class libraries providing base classes for EFSA implementations. EFSA takes care of the changes in the inner state of CPO (Communicating Protocol Object).

5.1.1.5 Timer

Timers are needed to prevent endless waiting of answers when communicating through non-reliable channel. Many protocols rely on the sureness that if the expected event

doesn't occur, timer will expire and inform the protocol so that it can react properly. This brings reliability and flexibility to the protocol as well as to the whole system. Most commonly timers are used when creating connection to peer entity.

5.1.1.6 PDU en/decoding

In peer to peer communication the information is moved in messages called PDU's. These messages need to be encoded to the lower layer primitive message since the lower layer has no information about the upper layers PDU's. On the other end the peer entity should know how to decode the PDU from lower layer primitive in order to react correctly. Encoding and decoding are essential for the layered architecture peer to peer communication and thus should be easily implemented in framework.

5.1.1.7 Summary

Although one might state that the features discussed here are self-evident, the situation isn't that unequivocal if we examine common protocol frameworks more closely. PDU en/decoding and EFSA definition are good examples of features not implemented properly in many frameworks. Often these features are in large part left to be implemented by the creator of the protocol albeit these features are commonly thought as part of the framework core. If these features demand lots of additional coding to implement them it can be said that the framework is not complete enough to be called a protocol implementation framework.

5.1.2 Commonly requested features

While there are certain requirements for framework there are also many useful features in different frameworks that help in creating usable protocol stack. These features have been requested by several experienced protocol designers. Some of these features are implemented in different frameworks but none of the frameworks include all of them. The implementations of these features are not comprehensive enough on most frameworks.

5.1.2.1 Protocol operations

Notation for defining the different layers of protocol stack and their communication conduits is needed to help communication between developer and implementor.

Different CPO's and messages might have different priorities or need Timeouts; thus scheduler should offer support to deal with this kind of situation.

Automatic generation of state machines is needed. These automatically generated EFSAs should contain the top-level decisions. Low level data handling should be encapsulated in separate functions. Graphical notation/language for defining EFSA is also needed. Notation should show the functionality of CPO, but should not be too detailed.

- It must be possible to specify reception and sending of both primitives and PDUs in an EFSA definition
- It must be possible to specify properties for a set of states in one place. This can be done e.g. using hierarchical states like in Harel's notation
- It must be possible to specify decisions and branches in the notation.
- The notation must be compact, so that a statechart can be drawn in one paper
- Visual protocol layer structure specification (~ CVOPS assoc) [23]
- Visual protocol entity instance structure specification (~ CVOPS struct.str) [23]

The UML statechart notation with additional semantics is a good candidate for EFSA notation.

Automatic generation of codecs would help development by removing manual code writing and therefore making sure that there are no bugs in any one of the codec's functionality. Generated code should be debuggable with the hand-written code.

In the protocol stack there is often a lot of different kind of data handling. Efficient implementation for common operations like copying and moving values between generic data structures increases the efficiency of the final product.

While implementing protocols there is often a need for commonly used algorithms such as sliding window. Support for this kind of general algorithms would greatly reduce the time of implementing the protocol.

Knowledge base or a cook book about implementation related solutions and facts would be useful because some old errors could be avoided and solutions which have presented their excellence could be utilized without needing to test several different approaches to the problem in vain.

Up-to-date documentation about the framework is a crucial demand. Proper documentation helps the developer to concentrate in creating the protocol instead of studying the frameworks special behavior.

5.1.2.2 External tools

There are many commonly used tools for designing and creating protocols. Support for using such tools would ease the design and creating process. Also the implementor should have to use as little "glue" code as possible when integrating the external tools. Some kind of flexible interface to outside would be useful in achieving this. Specification of interfaces should be done in detail because that way it would be easier to integrate external components to the system.

For testing purposes MSC diagrams of the running protocol would help the tracing of the protocol behavior. Also the framework should be scalable so that it is possible to test only one layer of protocol or even just one CPO. Possibility of changing CPO state and giving different input messages to it (for example to simulate corrupted messages) would greatly help debugging. When changes are made in the code to fix bugs the possibility of seeing different versions and the differences between them would be beneficial.

5.1.2.3 Summary

These features would greatly benefit the development of complex protocols by allowing the developer to focus on the real problems of protocol implementation.

Some of these features may be very difficult to implement properly in frameworks. For example the interface of external tools is very difficult to define so that it would be usable with large variety of tools. The benefits of these features are so significant that implementing them to the frameworks would be extremely advantageous.

5.2 Conduits

The Conduits framework was originally designed by Jonathan Zweig for his Master's Thesis in the University of Illinois [16]. It was used to implement the TCP/IP stack. The framework was later improved by Hüni et al. and it was then named Conduits+ [15].

There are several other Conduits+ based frameworks, such as Jacob framework by the Helsinki University of Technology [17], OVOPS++ by the TOVE project [19] and JVOPS of the Necsom Ltd [18]. Both Jacob and JVOPS are written in Java and the source code is available at their WWW sites. OVOPS++ is implemented in C++ and is the property of the funding partners until July 1999. All these implementations differ in many ways but share the same basic elements.

On this section the graphical symbols and the roles of different conduits conform to the Conduits+ framework.

5.2.1 Modeling Concepts

A Conduits framework consists of two basic elements, conduits and information chunks or messages. The conduits can be connected to each other creating a conduit graph that represents the actual protocol stack. Messages represent the information flowing through the stack. They traverse the conduit graph and can trigger actions in the conduits.

There are four different conduits - Protocols, Muxes, ConduitFactories and Adaptors (Figure 11 on page 24). Each of them are derived from the Conduit base class and therefore possess the same capabilities to both interconnect with other conduits and handle incoming messages.

The Protocol, Mux and ConduitFactory Conduits have two distinct sides, sideA and sideB, that can be connected to their neighbor Conduits. The Adaptor has only sideA since the other side of it is an interface to the environment.

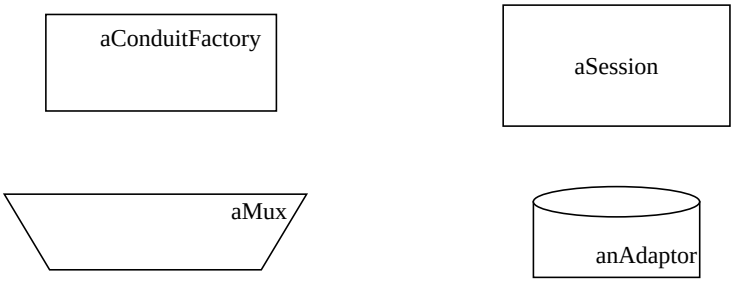

Fig. 11. Four kinds of conduits

The Protocol conduit provides the basic functionality of a communication protocol. It can be described as a finite state automaton that remembers the current state of communication and that receives, creates and responds to the control messages or the messages sent by its peer protocol entity. It also supplies additional services, such as counters, timers and storage, e.g. for partially received PDUs.

The Mux is a conduit that is connected to one sideA neighbor and to any number of sideB neighbors. The Mux multiplexes messages from several conduits to one single conduit and vice versa. Usually when the system is initialized there already is one conduit on the sideB, the ConduitFactory. A Mux is used to handle multiple simultaneous sessions on a single protocol layer. Protocol conduits can be dynamically added to and removed from the sideB of the Mux.

When a Mux is unable to determine the receiver of a message, the message is routed to the ConduitFactory. Usually the ConduitFactory installs a new Protocol conduit to the Mux sideB. Then the message that caused the creation of a new Protocol is routed to the new Protocol conduit through the Mux and is handled there. The ConduitFactory can also be used to take care of unexpected or corrupted messages.

An Adapter does not have a neighbor on its sideB. It is used as an interface to some other software or hardware. Thus the Adapters are the endpoints in the conduit graph.

5.2.2 Framework Structure and Services

In order to implement the Protocol state machine the different States must be defined and implemented. When a Messenger arrives in a Protocol, the Protocol conduit calls its apply method with a pointer to the Protocol conduit's current State object. Then the Messenger invokes its specific apply method in the State object that for example triggers the Protocol to change State. So the implementor must define separate methods for all Messengers that can reach that State.

In order to convey a message from Mux sideA to its destination, the correct sideB neighbor must be determined. A separate Accessor is usually used for this task. It is also used for storing the session key in message when messages are multiplexed from sideB to sideA. The use of separate Accessor means that there is no need to subclass the different Muxes from the base class. The implementor can use the general Mux and just attach the proper Accessor to the different instances.

The Conduits+ framework uses widely design patterns from [8]. Actually the Messenger introduced earlier is an example of Command pattern that lets any conduit handle any Messenger by passing it to its neighbor conduit. A Protocol conduit handles Messenger differently by applying it to its current State. Another example of design patterns is the Visitor pattern which is used to decouple Messengers from Conduits. Visitors are responsible for the traversal on the conduit graph and the Messengers only trigger actions in

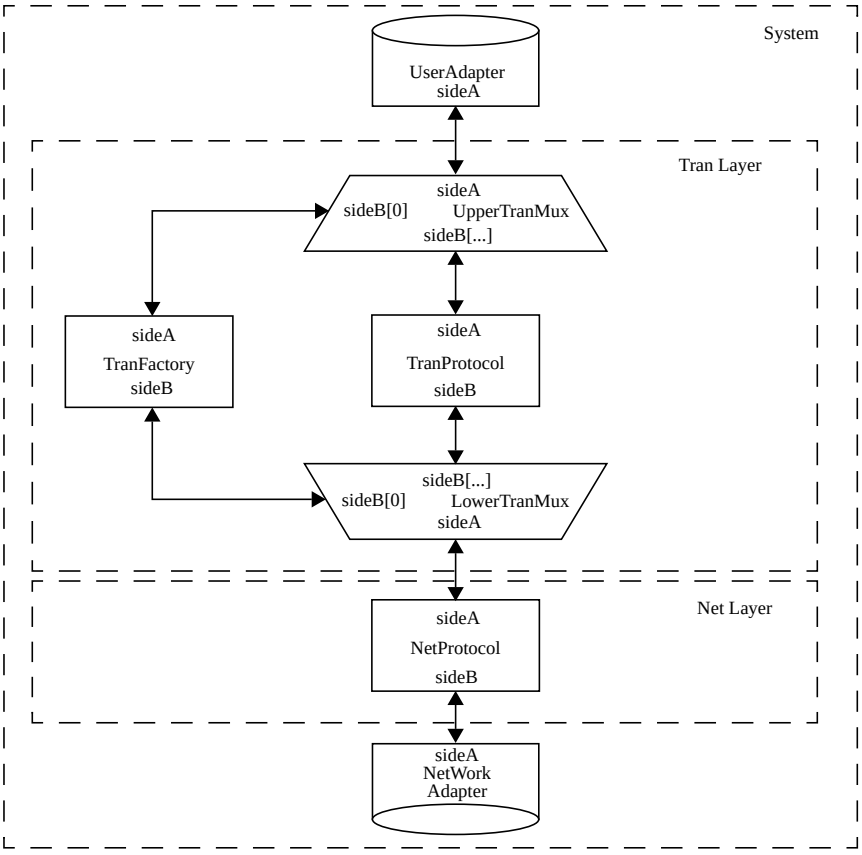

Fig. 12. Tran and Net protocol stack in Conduits framework

State objects. For more details on the used design patterns please see the original Conduits+ paper [15]

There are various scheduling schemes for the conduits frameworks. The original conduits model does not fix the system scheduling. Usually the frameworks schedule conduits but some other implementations exist. For example the Jacob framework provides the possibility to schedule the Visitors travelling the graph.

The Conduits+ lack proper Test and Management Interfaces. These can be implemented using Adapters with Muxes but it increases the complexity of the model. Also the absence of a peer interface is a serious limitation.

5.2.3 Support for Different Phases of Development

The Conduits+ framework lacks the support for the design phase. The conduits graph can be designed conceptually but there is no tools available for refining this design. However, the other frameworks are currently being developed and for example the JVOPS framework is going to have a set of graphical design tools.

The implementation of a simulator is done by using the already implemented conduits elements and other services provided by the framework and the coding the rest, e.g. in C++ or Java. Since there is no tools supporting the design phase a lot of the work must be done manually.

Usually in the Conduits frameworks there is only very primitive tracing available and no support for e.g. TTCN. Therefore the testing can be a tedious phase.

5.2.4 Summary

The Conduits+ is a relatively light black-box framework. Its basic ideology is very easy to understand and the roles of the different conduits are quite intuitive.

With the Conduits+ framework it is quite easy to quickly implement small protocol systems but managing more complicated ones can be very difficult. Most of the work must be done manually. The used design patterns make the system more usable and quite elegant but increase the complexity of the whole system.

The other referenced Conduits+ based frameworks are being developed further and it is very interesting to see how they use the potential that the conduits possess.

5.3 CVOPS

CVOPS is a tool and run-time environment for implementing communications protocols. CVOPS was originally developed by the Technical Research Centre of Finland (VTT). The development started in the year 1985 and at the time being the latest version number is 6.2. It is written entirely in C, apart from the latest version, CVOPS 6.2, which is partially written in C++. The development of CVOPS has been going on for over a decade now, and the framework clearly shows many signs of a high age. In spite of that, many feel that CVOPS still is a very useful tool when developing communications protocols. Today, CVOPS is used by the finnish telecommunications industry, and by several universities in Finland and in the rest of europe [24].

CVOPS is available for many different operating systems, e.g. UNIX, VAX, MS-DOS and Windows NT/95 [26]. The porting of the framework is relatively easy, because it needs only a small set of services from the OS. This means that it can be also be ported to embedded systems with limited resources.

5.3.1 Modeling Concepts

The main component of CVOPS is the virtual task (vTask). VTasks connect to other vTasks through its interfaces. A number of messages is associated with each interface. A vtask can communicate with another connected vtask by sending messages through it's interface. Multiplexing is performed by the protocol entity. It directs the received message to the correct connection, either using a default algorithm or one defined by the implementer. The entity can also function as a connectionless layer.

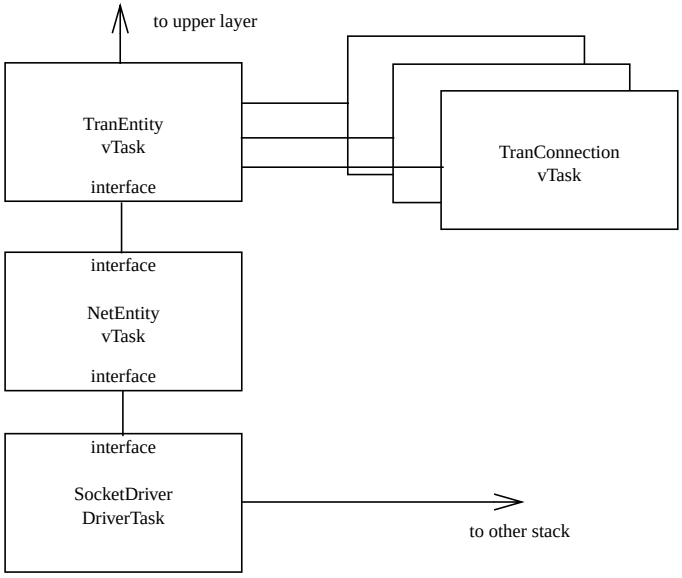

Fig. 13. A simple protocol stack in CVOPS, the functionality of the entity and connection vtasks

5.3.2 Framework Structure and Services

CVOPS offers all the basic services required of a protocol framework - an EFSA interpreter, a scheduler, message passing, PDU handling (including frame handling, encoding and decoding), timers, and interfaces to devices. These services are embedded into the framework itself, and they are not separate tools. The services are mainly provided as a library of functions. For example, the tools for handling frames in PDUs are basically a set of functions, used for inserting and extracting bytes from a byte buffer that is contained in the message.

The state machine is defined using a simple C-like language, that is designed especially for protocol software use. The programmer creates a text file containing the description of the state machine using this EFSA-language. The EFSA interpreter, which is

provided by the CVOPS core, then creates an internal representation of the description at run-time. The EFSA can also be implemented in plain C, but this is of course a much more tedious process.

Message passing is handled by the CVOPS core to a certain extent, but there is still a lot of work left to the programmer. For example, the implementor has to define for every interface of every layer a number of parameter functions just to get, put or trace the message parameters. Message passing is asynchronous. CVOPS can produce a textual trace of the events occurring in the system, but this is only partially automated, and the programmer often has to do some additional programming.

5.3.3 Support for Different Phases of Development

The design phase is not strongly supported in CVOPS. There are some tools, e.g. ASN.1 compilers available for CVOPS, but often the design has to be done in another environment than CVOPS.

The implementation phase is strongly supported in CVOPS. It offers the majority of the features needed to build communications protocols - scheduling, message passing, a layer structure, device drivers and peer message handling.

The testing phase is somewhat supported. There is no GUI or MSC generator available, but there is still a textual user interface and CVOPS can produce a textual trace of the events.

5.3.4 Summary

Even though CVOPS contains services and tools covering many different aspects of protocol programming, there is a lot of manual work left for the programmer. The limitations of a non-object oriented implementation language are visible in many parts of CVOPS. Reusability of components is poor, and the system not flexible.

5.4 OVOPS

Ovops was originally developed in co-operation between Telecom Finland (nowadays called Sonera) and Lappeenranta University of technology. The development started in the mid 90's. In later versions there have been other participating partners, for example Nokia Research Center. Ovops uses the same basic idea as does CVOPS - a protocol layer entity or connection is represented as a task, and this task uses the services of the virtual operating system, not the underlying operating system. The difference between these two frameworks lies in the implementation - Ovops uses object-oriented techniques extensively and it is modular in its design.

The Ovops core offers many of the features needed to create communicating systems. The protocol toolbox (PTB) is an extension to the core that facilitates the design and im-

plementation of communications protocols (see Figure 14 on page 30). In this chapter, we describe framework Ovops when used together with PTB.

5.4.1 Modeling Concepts

The main building blocks of Ovops are Ovops tasks (oTasks) and ports. When using the PTB, protocol tasks (pTasks) are used instead of oTasks. A pTask is a subclass of oTask containing additional communications protocol specific functionality. Interconnected ports between two tasks form channels. Tasks exchange information by sending messages through channels. The tasks know only the interface of the other tasks. The interface consists of the port and the messages that can be sent through the channel. In all other senses, all tasks are totally independent of each other.

5.4.2 Framework Structure and Services

The Ovops-framework consists of two separate main components: the application component and the tool pool. These components rely on operating system services, and they provide services to the application objects (tasks) [28].

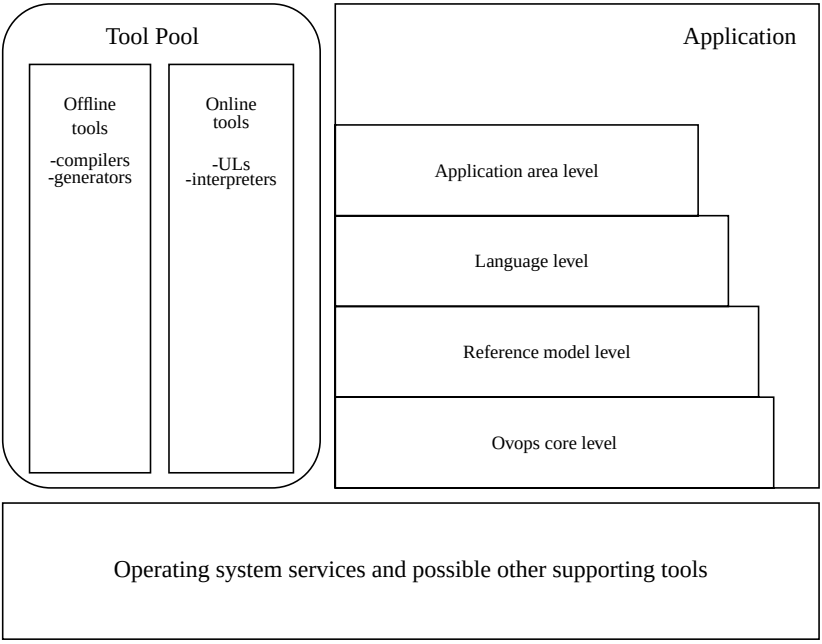

Fig. 14. Application, toolPool and OS

The application component is divided into four layers: the Ovops core level, the reference model level, the language level and the application area level.

The core classes offer only basic operations needed to build event based applications, such as multitasking, scheduling, message based communication, memory management and timer functions. All the communications protocol specific services are offered separately in the other levels of the application component and in the tool pool.

The reference model level offers more advanced tools for the selected reference model. In this chapter we have used an implementation of a OSI oriented reference model level, the protocol toolbox (PTB). PTB offers both off-line tools and additional classes that aid the development of protocol software. The PTB classes define the structure of e.g service access points (SAPs), protocol data units (PDUs) and the state automaton. For these three components there is also a set of off-line generators to ease manual work. These generators generate understandable and editable code, but the implementor is still expected to define a large part of the functionality. The PTB classes also offer support for multiplexing and a peer port abstraction.

The language level could contain mappings from a higher level language (e.g. SDL) to Ovops-code, but there are no implementations for this layer as of yet. The application area includes frequently used elements for a specific application area, for example tools for developing intelligent networks.

The tool pool is divided into two categories, on-line and off-line tools (see Figure 14 on page 30). The on-line tools include tools for tracing and debugging the execution, and the off-line tools include e.g. the three code generators found in the protocol toolbox.

5.4.2.1 Offered services and inner structure

In Ovops, a communicating protocol object is modeled as a protocol task (pTask). Basically a pTask is a C++ class that contains all the basic functionality of a protocol object, such as schedulability and message passing. The developer then instantiates this class with the needed parameters to suit system specific needs. Protocol storage is implemented as class variables. Every pTask can have an arbitrary number of variables of any type supported by C++. The EFSA is implemented as a collection of state-input methods, i.e. for every incoming message of every state of the state machine, there has to be a method that defines the functionality of the state-input pair. These methods are declared by the sapg, but the functionality has to be implemented by the programmer. Communication between protocol layers is performed by exchanging messages. Peer to peer communication is performed by exchanging PDUs. The messages and the PDUs are defined with the help of code generators (sapg and pdug) that are included in the PTB. Message passing is automated, but encoding and decoding functions have to be defined for each of the PDUs. In order to help this process, the PTB offers a rich set of functions for handling the data frame of the PDU. The run time system includes a scheduler (which can be defined by the implementor) that schedules the different tasks. Scheduling is asynchronous [29].

5.4.3 Support for Different Phases of Development

The design phase is supported by various tools found in the protocol toolbox, in particular pdug and sapg. With the help of these tools, the user can design and create PDUs, messages and interfaces easily. There are some other tools available as well, e.g. ASN.1 compiler for Ovops (ACO), which produces BER encode and decode functions for each ASN.1 type [28].

Implementation Ovops supports the implementation phase the most. Especially when using the PTB, implementation of a communications protocol can be straightforward. There is still a lot of C++ coding left for the implementor, but the work is mainly mechanical.

The testing capabilities of Ovops are somewhat limited, but nevertheless it offers some basic capabilities that are useful. The support for testing includes: o Automatic generation of textual trace o Automatic generation of MSC trace o System structure and object inspection o A textual user interface With the help of the textual user interface, the user can e.g. send messages, trace the flow of operation and read the contents of the messages.

5.4.4 Summary

Ovops follows the same basic principles as CVOPS, but it provides more flexibility, more tools and automation and more modularity. This does not come without a cost - Ovops is a large system, demanding a lot of resources from the underlying system and it takes time to learn the numerous details

5.5 SDL

Specification and Description Language (SDL) is a standard language to specify and describe communicating systems. It is currently developed by ITU-T and is defined in the Z.100 recommendation [31].

There are several SDL versions. The first recommendation was released in 1976, followed by further releases once in every four years. In the 1988 version the language was given a formal basis and reached a mature status as a Formal Description Technique. The 1992 version was a major improvement introducing among other things object orientation. The latest version, currently being, the 1996 version was only a fix correcting the errors found during the few years of use [30].

There are several public domain and commercial applications that support SDL. A list of some available ones can be seen at the SDL Forum Society WWW site [32].

5.5.1 Modeling Concepts

An SDL system is a sum of its parts. In SDL the basic building blocks are processes which communicate with each other and the environment surrounding them using signals.

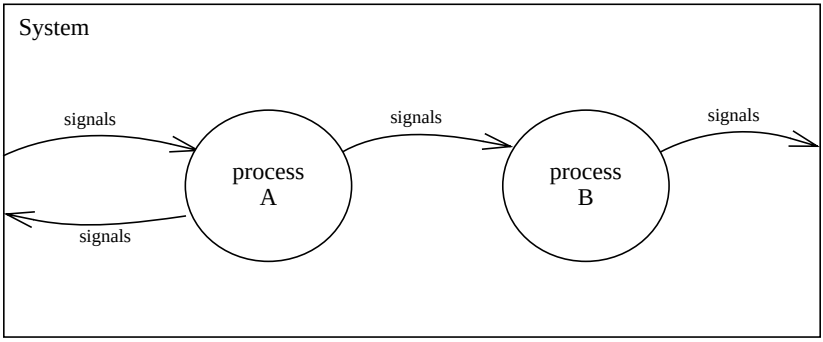

Fig. 15. Communicating processes in SDL

5.5.2 Framework Structure and Services

When an SDL system is initialized there exists a number of processes. In addition to that process instances can be dynamically created. These instances are specified by types that define the common properties of a process set. Note that the signal routes are connected to sets, not to specific instances. The manager process must be aware of all its managed processes and send signals to them using their distinct process identifications (PIds).

SDL process instances are Extended Finite State Automatons (EFSAs) that are concurrent and run in parallel [30]. The protocol state machines can be defined with these processes.

Timers are objects that belong to a process instance. A timer can be set, examined and resetted and it can send a signal to a process. These signals are handled like other signals in the system.

The processes can also contain data stored in attributes. The variable definition notation resembles the Pascal notation: “dcl variable VariableType;”. Declared variables are

visible in all sub-blocks and sub-processes. Thus global variables can be defined by declaring them in the system level diagram.

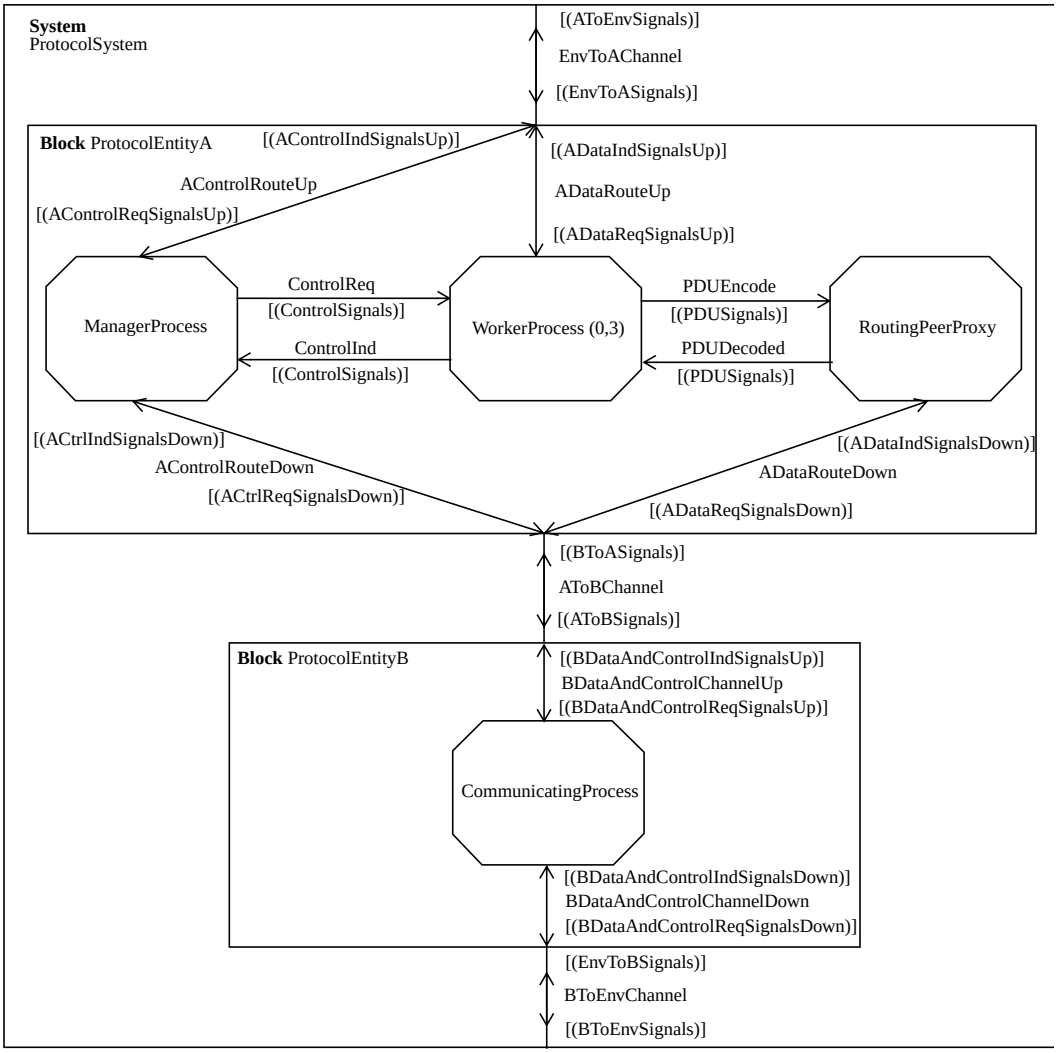

Fig. 16. Example Tran and Net protocol stack in SDL

All data in SDL is defined using abstract data types (ADTs). ADTs are used for defining data in an implementation-independent way. Data is only defined by its properties and is not limited by the hardware or software, i.e. the specification of objects and the opera-

tions on those objects is separated from the implementation. An ADT specification has the following properties: values, literals or names of values, operations on the values and equations. SDL has a set of predefined types (Integer, Character, Boolean, PId, Time and Duration) that can be used when constructing structured types.

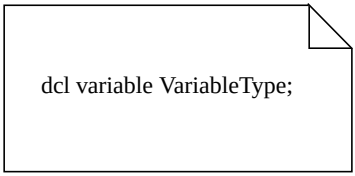

Fig. 17. Variable declaration

The data structures and signal parameters can be designed using ASN.1. The ITU-T recommendation Z.105 defines the mapping from ASN.1 types to SDL abstract data types. However, this mapping has some serious limitations. For example SDL has no support for data transparency (the use of ASN.1 ANY type) and Z.105 does not provide any encoding or decoding functions.

SDL systems can be defined using either graphical or textual representation. The graphical representation makes SDL more user-friendly and easier to understand while the textual representation was originally planned to be the interchange format. The two representations are partially intersecting. This is because of that for example data is defined more naturally in text format.

Fig. 18. An example of a process EFSA definition with its graphical and textual representation.

5.5.3 Support for Different Phases of Development

SDL supports the design phase very well. A simple simulator can be designed easily using the graphical notation. The possibility to use ASN.1 with the SDL is an advantage and the MSCs and validation services provided by many SDL applications makes SDL a powerful design tool.

Usually the software that supports SDL has a built-in compiler that can be used to generate a standalone executable. SDL is not a real programming language but a design and specification language so the generated code may use a lot of resources.

SDL has poor support for PDU encoding and decoding. In most applications these functions and other demanding calculations can be implemented externally using a real programming language such as C, and then imported in the SDL system.

Note that SDL does not provide any means for modeling the outside world. The channels are used to connect the designed system to the environment but this only defines the signals that can be transmitted. The generated SDL system must be e.g. integrated into some other framework to create a fully functional simulator.

MSCs are supported by most of the SDL applications and they can be used in the implementation debugging phase to verify the correct signalling. TTCN can be used to create the test cases for the implementation. SDL is a formal language and therefore makes exhaustive testing possible. The complete testing may be a very time consuming operation.

5.5.4 Summary

SDL is an excellent design tool but it lacks the facilities required for the implementation of complex protocol simulators. However, the protocol behavior designed with SDL can fairly easily be mapped to be used with other frameworks.

Part 2 Protocol Concepts and Patterns

Chapter 6 Introduction

XXX

This part of the book defines Conceptual Protocol Classes, Communication Patterns, and improves process presented in “Protocol Engineering Process” on page 17.

6.1 Conceptual Protocol Classes

Communication protocols and their entities can be structured using UML class diagrams. These diagrams take a conceptual perspective, they represent the main concepts in the domain under study is drawn. They provide some views on protocol structure and behavior.

The understanding of the conceptual model behind of a system architecture facilitates the creation of a clear, consistent, and more understandable system architecture which is also efficient to implement and maintain. It facilitates integration of components produced using different frameworks.

In this chapter one possible conceptual class diagram for communication protocols will be presented. Note that these diagrams provide some views on protocol structure and behavior. The chosen perspective gives enough concepts for modeling of protocol system without going into framework-specific level. Parts of the conceptual classes are explained using Design Patterns [8].

XXX

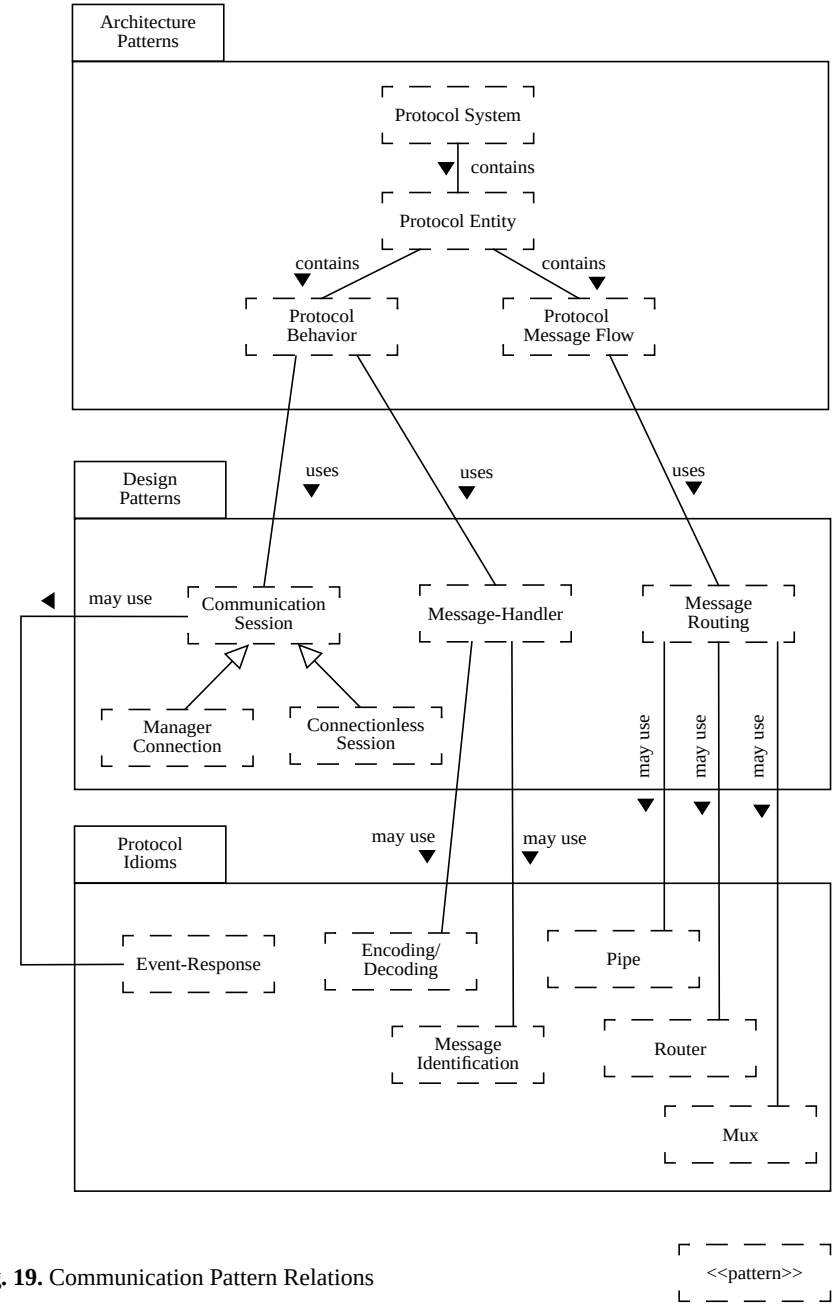

Fig. 19. Communication Pattern Relations

6.2 Communication Patterns

Parts of the conceptual classes are explained using Design Patterns [8].

XXX

Communication Patterns are grouped into three categories: architectural protocol patterns, protocol design patterns, and protocol implementation idioms and algorithms. This grouping follows [13].

6.2.1 Architectural Protocol Patterns

An architectural pattern express a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [13].

6.2.2 Protocol Design Patterns

A design pattern provides a scheme for refining the susbsystems or components of a software system, or the relationships between them. It descibes a commonly-recurring structure of communicating components that solves a general design problem within a particular context [8].

6.2.3 Protocol Imlementation Idioms and Algorithms

An idiom is a low-level patern specific to a programming language. An idion describes how to implement particuar aspects of components or relationships between them using the features of the given language [13].

Chapter 7 Conceptual Protocol Classes

One approach to structure communication protocols and their entities is to use UML's conceptual class diagram. In conceptual perspective, a diagram representing the concepts in the domain under study is drawn. These concepts will naturally relate to classes that implement them, but there is often no direct mapping to implementation [35].

The understanding of the conceptual classes behind of a system architecture facilitates the creation of a clear, consistent, and more understandable system architecture which is also efficient to implement and maintain. It facilitates integration of components produced using different frameworks. The conceptual model of the system is also needed when implementing tools for code generation. The UML is used to present the classes in this chapter, because it is a well known and standardized modeling language. The UML is standardized by the OMG [36].

In this chapter one possible conceptual class diagram for communication protocols will be presented. Note that these diagrams provide some views on protocol structure and behavior. The chosen perspective gives enough concepts for modeling of protocol system without going into framework-specific level. Parts of the conceptual classes are explained using Design Patterns [8].

These conceptual classes and related Design Patterns have been derived during studies of several existing communication protocols which are implemented using several different frameworks [15][23][27][32]. They are explained in the sections XXX, XXX, and XXX. Concrete examples how these classes are used is given in the chapter XXX.

7.1 Conceptual Protocol Classes

Different protocols seem to have common general parts and relations between them. These parts can be identified and they are independent from a system implementation. In this chapter one possible conceptual class diagram will be presented.

Main conceptual classes and their associations are presented in Figure 20 on page 44. These Conceptual Protocol Classes are divided into the Protocol System Structure (section XXX) and the Protocol System Functionality (section XXX) classes.

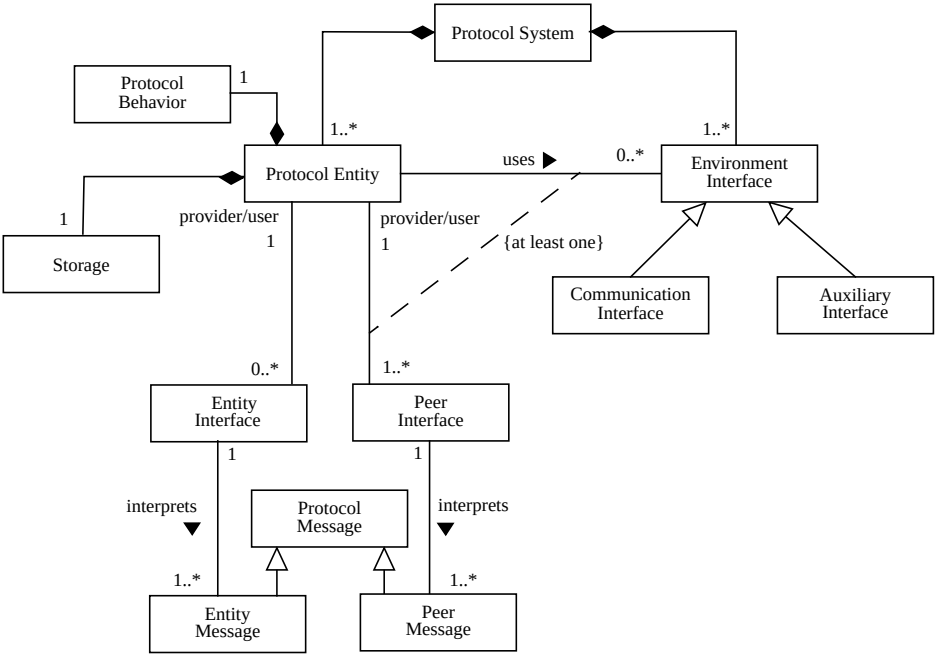

Fig. 20. Main Protocol Classes

The *Protocol Structure* contains concepts that model static parts of a protocol system, i.e. what are the components that a system is composed of, what is the structure of the components and how the system components are interconnected. System component related concepts include Protocol System, Environment Inteface, Communication interface, and Auxiliar Interface. They are presented in section XXX. Concepts that are related to structure of the components include Protocol Entity, Storage, Entity Interface, and Peer Interface. They are described in XXX.

The *Protocol System Functionality* includes concepts that model dynamic parts of a system like responses to different events (arriving messages, errors etc.). These concepts included Protocol Behavior, a Storage, Entity Interfaces and Peer Interfaces. These concepts are explained in XXX.

Protocol Entities communicate using Protocol Messages. Protocol Messages are used to deliver messages between entities in the same system and between entities in different systems through Entity and Peer Interfaces. These classes are explained in XXX.

7.2 Protocol Structure Concepts

7.2.1 Protocol System

A *Protocol System* is a high-level class which encapsulates the whole protocol system. A Protocol System class describes a protocol system structure by specifying
1. what are the components that a system is composed of,
2. how the components are interconnected, and
3. how a system communicates with its environment.

Related pattern, Protocol System, is presented in XXX.

The componets that form a protocol system are one or more Environment Interfaces and one or more Protocol Entities (Figure 21). A Protocol Entity represents a protocol layer or sublayer. It communicates with other Protcol Entities in the same system by exchanging messages. A Protocol Entity provides and uses Entity Interfaces and a Peer Interface. Entities in the same system communicate using Protocol Messages through Entity Interfaces. It is these binding between Entity Interfaces and their user and provider Protocol Entities that specify how system components are interconnected. Specific Protocol Messages are used in communication between the peer systems through Peer Interfaces. A Protocol Entity is presented in detail in section XXX , and Protocol Messages in section XXX.

The Protocol System has conceptual classes (from Figure 21):

Environment Interface. An Environment Interface models interfaces to system's environment. From a protocol system's point of view an Environment Interface acts as a mesage source for incoming external messages and as a message sink for outgoing messages.

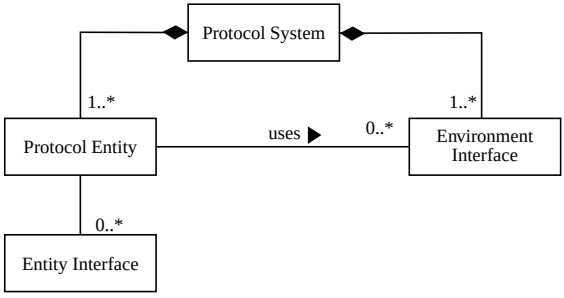

Fig. 21. Protocol System

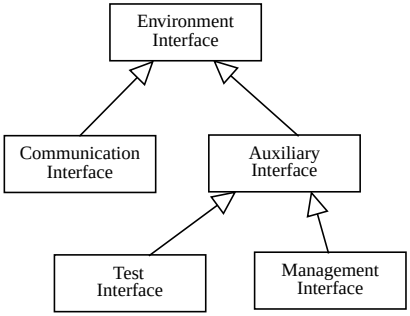

Fig. 22. Enviroment Interface Classes

An Environment Interface may have different roles. The subclasses, Communication Interface and Auxiliary Interface, represent those roles. The Enviroment Interface subclassed are presented in Figure 22 on page 46:

Communication Interface. A Communication Interface handles communication with a low level service (i.e. hardware) and with system users (i.e. applications). It handles Protocol System's normal communication.

Auxiliary Interface. An Auxiliary Interface handles communication that is not directly related to Protocol System's normal communication. For example test and management messages fall into this category and they may have their own interfaces as presented in Figure 22. An Auxiliary Interface is subclassed as Test and Management Interfaces.

7.2.2 Protocol Entity

A Protocol Entity (see Figure 23 on page 46) contains a Protocol Behavior and a Storage, and it uses and provides Entity Interfaces and a Peer Interfaces. One Protocol Entity can be used as a layer in the Layers Architecture pattern [13]. Classes of the Protocol Entity (from Figure 23):

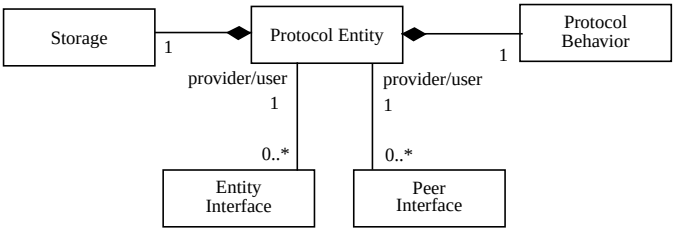

Fig. 23. Protocol Entity

Protocol Behavior. A Protocol Behavior handles protocol functionality. The Protocol Behavior is explained in detail in section "Protocol Behavior" on page 47.

Storage. A Storage contains all volatile and non-volatile information of a Protocol Entity. Information collected to Storage can be visible for the whole Entity or it can be split to dedicated parts. An example of this is connection specific information.

Entity Interface. An Entity Interface handles communication between two Entities in the same protocol system. It interprets a Entity Message which is received from another Protocol Entity. It also produces a Entity Message which is sent to another Protocol Entity in the same system.

Peer Interface. A Peer Interface handles communication between entities located in the peer protocol system. It interprets a Peer Message which is received from a peer Entity. It also produces a Peer Message which is sent to another Entity in a peer system.

The Entity Message and the Peer Message are explained in "Protocol Message Flow" on page 48.

7.3 Protocol System Functionality Concepts

The Protocol System Functionality includes the dynamic parts of a system. These are the Protocol Behavior and the Protocol Message Flow. The Protocol Behavior defines protocol's response to events. The Protocol Message Flow defines Protocol Messages and information related to them. Static parts of a system are presented in "Protocol Structure Concepts" on page 45.

7.3.1 Protocol Behavior

The Protocol Behavior contains needed protocol functions for the system in concern. A Protocol Behavior is typically specified as a state machine. A protocol behavior can be divided into two main modes: connectionless and connection-oriented behavior. In connectionless behavior a communication session is a simple Request-Response or just Request) kind of message exchange. The connectionless Protocol Behavior contains one Connection Manager which handles all communication events.

In connection-oriented behavior a communication session consists of connection establishment, message exchange, and finally disconnection phase. There can be multible concurrent communications which can be in different phases. A special case of connection-oriented behavior is a case when a connection may have sub-connections.

The connection oriented Protocol Behavior contains *Router*, *Connection Manage*r, and zero or more *Connections*. A Manager-Worker pattern is used to manage multiple concurrent communication sessions (defined in section XXX). The used Protocol Behavior pattern is presented in Figure 24. Both Connection Manager and Connection can use Event Response pattern (defined in section XXX).

The classes of the Protocol Behavior (from Figure 24):

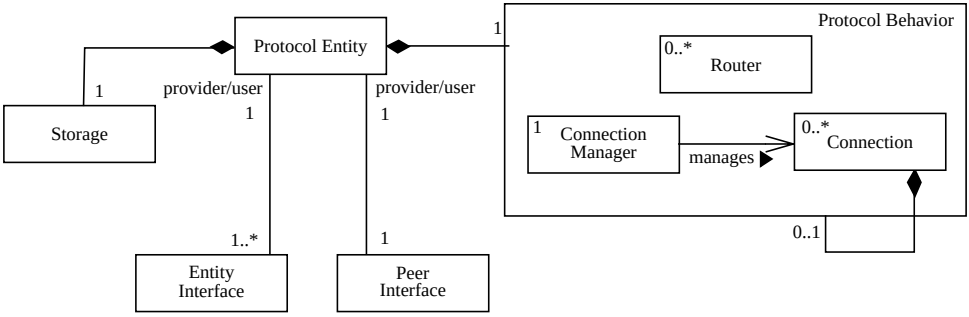

Fig. 24. Protocol behavior

Protocol Behavior. A Protocol Behavior contains zero or more Routers, one Connection Manager, and zero or more Connections.

Router. A Router routes incoming messages to correct receiver i.e. Connection Manager or one of Connectins.

Connection Manager. A Connection Manager creates, controls, and closes connections as needed in connection-oriented protocols.

Connection. A Connection handles one communication session between two communicating peers in connection-oriented.

The Protocol Entity and its classes are explained in the section “Protocol Entity” on page 46.

7.3.2 Protocol Message Flow

Protocol Entity’s classes which are related to data flow, are presented in Figure 25 on page 49. These are Storage, Protocol Behavior, *Protocol Message*, *Entity Message*, *Peer Message*, *Service-only Parameter*, *Peer-only Parameter*, *Shared Message*, *Local Syntax*, and *Transfer Syntax*. One Message type and one Interface type form a Message-Handler pattern (XXX). The Protocol Message Flow Pattern is presented in XXX.

The classes of the Protocol Message Flow (from Figure 22):

Router. A Router determines proper receiver for a Protocol Message in case of connection-oriented communication.

Protocol Message. A Protocol Message is subclassed to a Service Message and a Peer Message.

Entity Message. A Entity Message is used between two entities in the same system. It contains data and control information.

Peer Message. A Peer Message (aka PDU) is used between two entities in the different systems.

Message Parameter. A Message Parameter contains all message parameters.

Classes Storage, Protocol Behavior, Entity Interface, and Peer Interface are explained in “Protocol Entity” on page 46, on page 30 and “Protocol Behavior” on page 47.

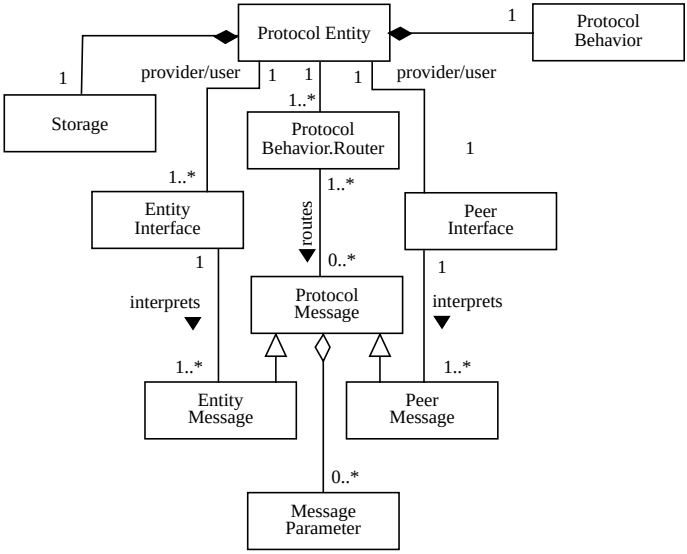

Fig. 25. Protocol Message Flow of a protocol system

7.4 Protocol Message Concepts

XXX

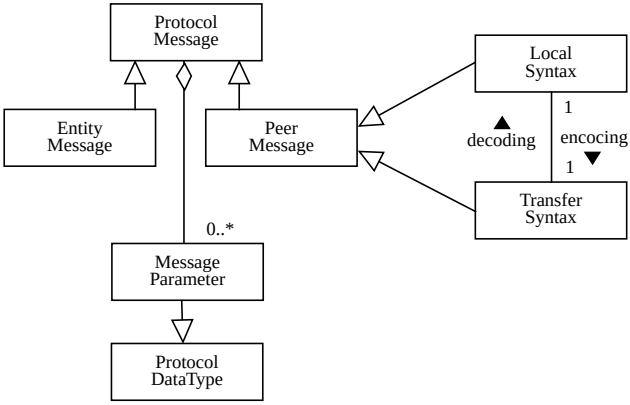

Fig. 26. Protocol Message Classes

Chapter 8 Architectural Protocol Patterns

XXX

8.1 Protocol System Pattern

Name Protocol System

Context

Problem

Solution The Protocol System pattern encapsulates the whole protocol system. A Protocol System pattern is used to implement a protocol system by specifying:
1. what are the components that a system is composed of,
2. how the components are interconnected, and
3. how a system communicates with its environment.

The roles that form the Protocol System pattern are Protocol System, Protocol Entity, Entity Interface, and Environment Interface, as shown in Figure 27.

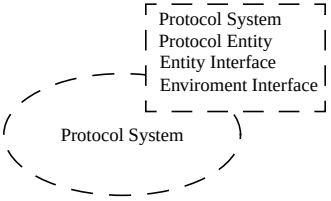

Fig. 27. Protocol System Pattern

Structure The Protocol System Pattern has Protocol System, one or more Protocol Entities, one or more Entity Interfaces, and one or more Environment Interfaces as shown in Figure 28 on page 52.

The Protocol System represents object which manages the whole structure of system in concern. A Protocol Entity represents a protocol layer or sublayer. It communicates with other Protocol Entities in the same system by exchanging messages. A Protocol Entity provides and uses Entity Interfaces. Entities in the same system communicate using

Protocol Messages through Entity Interfaces. It is these binding between Entity Interfaces and their user and provider Protocol Entities that specify how system components are interconnected.

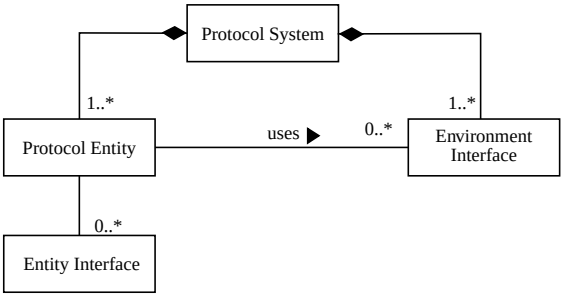

Fig. 28. Protocol System Pattern Structure

*An Environment Interfac*e models interfaces to system's environment. From a protocol system's point of view an Environment Interface acts as a message source for incoming external messages and as a message sink for outgoing messages.

An Environment Interface may have different roles. The subclasses, Communication Interface and Auxiliary Interface, represent those roles. The Enviroment Interface subclassed are presented in Figure 29.

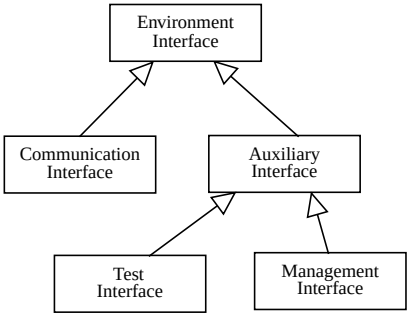

Fig. 29. Enviroment Interface Classes

A Communication Interface handles communication with a low level service (i.e. hardware) and with system users (i.e. applications). It handles Protocol System's normal communication.

An Auxiliary Interface handles communication that is not directly related to Protocol System's normal communication. For example test and management messages fall into

this category and they may have their own interfaces as presented in Figure 29. An Auxiliary Interface is subclassed as Test and Management Interfaces.

Implementation Following section contains implementation examples of Protocol System pattern in four differnt implementation framework. These frameworks are explained in detail in chapter “Tools for Protocol Implementation” on page 19.

Conduits

There is no distinct System entity in the Conduits+ framework. The conduits are binded together for example in the main function. Also the Layer entity is conceptual. A Protocol Entity consists of at least one Protocol conduit and any number of Muxes and Conduit-Factories.

The Adapter conduit provides all the different Environment Interfaces and is the only way to attach the conduit graph to the environment.

The sides of conduits are used to connect conduits and thus are the Entity Interfaces in the Conduits+ framework. No separate Peer Interface can be found. The Protocol conduit must encode the peer messages and insert the PDUs manually in the lower lever service primitives and vice versa.

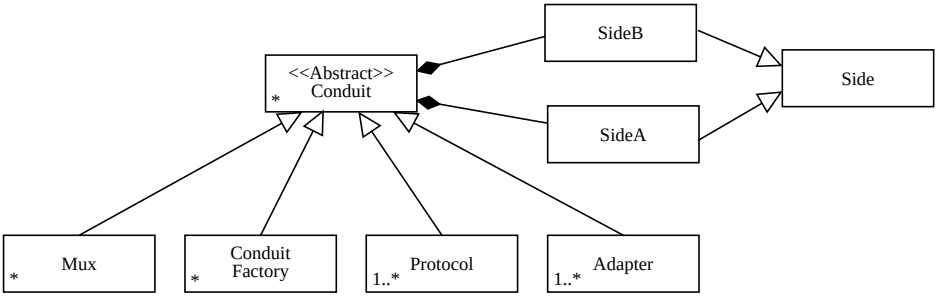

Fig. 30. Conceptual Conduits protocol system diagram

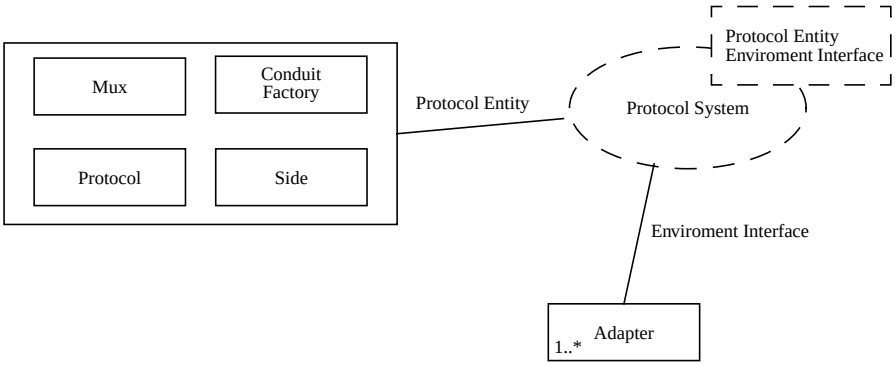

Fig. 31. Conduits Protocol System as pattern

CVOPS

In CVOPS, a protocol system consists of virtual tasks (vTasks) and interfaces between them. The system communicates with the environment using driver tasks. Driver tasks are normal vTasks, but they use OS services to perform a specific function, such as communication through sockets or the provider of a user interface. A file called struct.str binds together the different layers by defining which interfaces of a vTask connect to which other vTasks. This defines the statical layer structure of the protocol system.

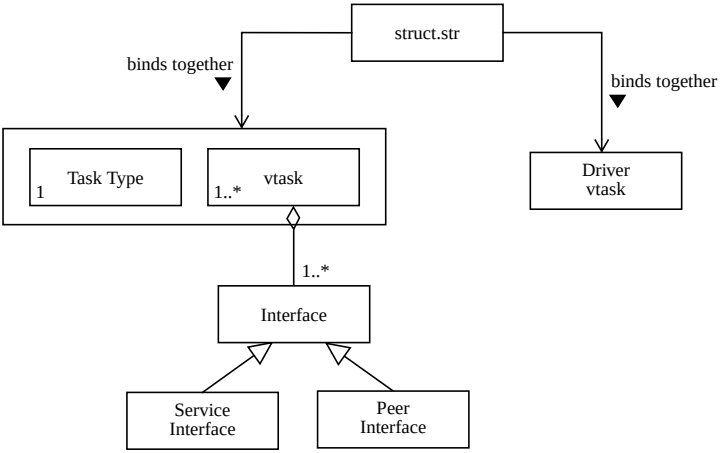

Fig. 32. Conceptual CVOPS protocol system diagram

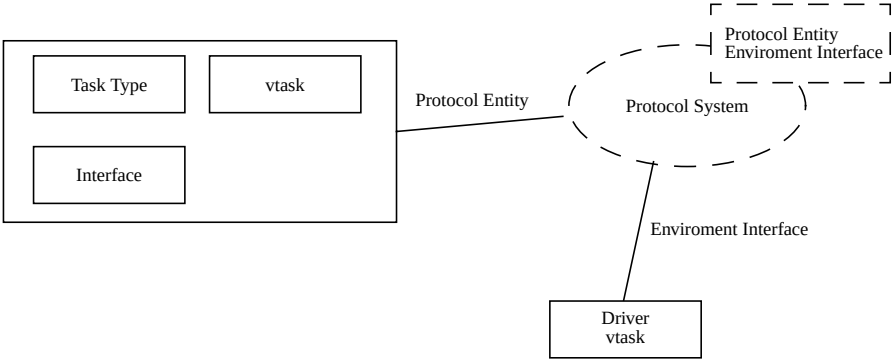

Fig. 33. CVOPS Protocol System as a pattern

OVOPS

In Ovops, the protocol system is constructed by connecting the ports of the separate protocol layers together, and in this way forming channels between them. A protocol layer

can consist of one or more pTasks. IoHandlers are used to communicate with the environment. A port can act as an interface between two layers or between peer objects.

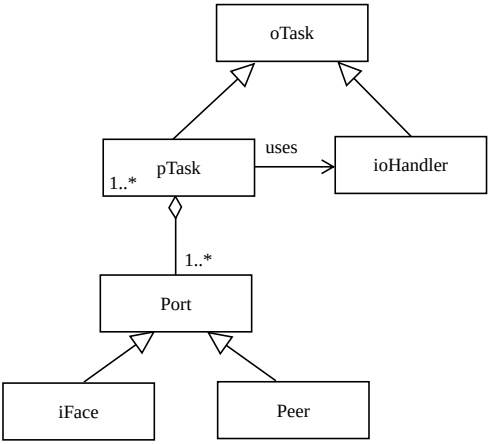

Fig. 34. Conceptual Ovops protocol system diagram

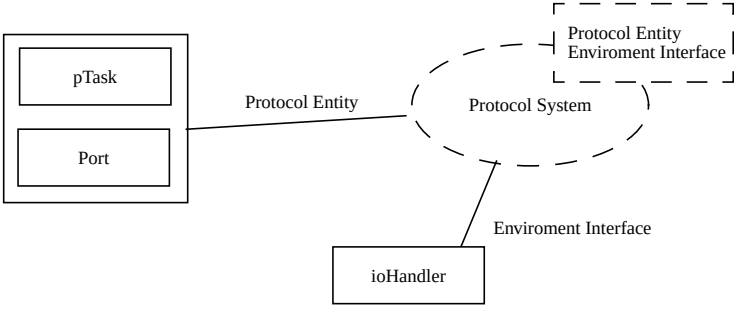

Fig. 35. OVOPS Protocol System pattern

SDL

A large number of processes communicating with each other can be difficult to comprehend. SDL provides mechanisms for coping with complex systems. The processes can be organized in a hierarchy of blocks hiding the fine details when inspected from a higher abstraction level.

In SDL a block can represent a single Layer. The different layers are interconnected with channels which convey signals. Signal lists can be used to define the signals that can be transmitted through a channel and thus define the Service Inteface between the two layers. There is no specific Peer Interface structure in SDL. The PDUs must be manually encoded and inserted in the lower level service primitives.

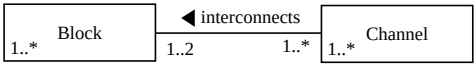

Fig. 36. Conceptual SDL Protocol System diagram

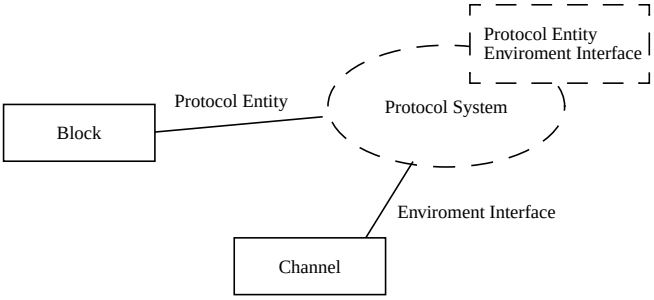

Fig. 37. SDL Protocol System pattern

Consequences The Protocol System pattern has several advantages, but it also contains a few disadvantage. These are quite similar as in the Layers Architecture pattern [13].

Benefits

- Reuse of Protocol Entities. If an individual entity embodies a well-defined abstraction and has a well-defined and documented interface, the entity can be reused in multible context.
- Support for standardization. Cleary-defined and commonly-accepted levels of abtraction enable the development of standardized tasks and entity interfaces. Different implementations of the same entity interface can be interchanged.
- Dependencies are kept local. Standardized entity interfaces between layers usually confine the effect of code changes to the Protocol Entity that is changed. Changes in the environment affect only Environment Interface in concern.
- Exchangeability. Individual Protocol Entity implementations can be replaced by semantically equivalent implementations without too great effort.

Liabilities

- Cascades of cahnging behavior. Entities can often be shielded from changes in other entities. However sometimes a change ripples from one entity to all others. In this case the Protocol System pattern becomes a disadvantage if a substantial amount of rework has to be done on many entities to incorporate an apparently local change.
- Lower efficienty. An architecture, which is implemented using the Protocol System pattern, is usually less effiecient than a monolithic implementation. This is result from the large number of operations and transformations the Protocol Message Flow experiences when it passes throught the entities.
- Unnecessary work. Some services performed by previous enties may not needed by adjacent entities. This has of cource a negative impact on performance.

See also

The Layers Architecture pattern [13] is closely related to this pattern. However, in the Protocol System Pattern, Protocol Entites don't have to be adjacent.

8.2 Protocol Entity Pattern

Name Protocol Entity

Context The Protocol Entity pattern can be used as a compositional part of the Protocol System pattern . It can behave as a layer in the Layers Architecture pattern [13].

Problem t is complex to design, implement, and test a Protocol Entity. It has to have at least the following functionality:

- Manage possible multiple concurrent communication sessions.
- Store internal states and other information.
- Communication with other entities in the same system.
- Communication with entities in peer systems.

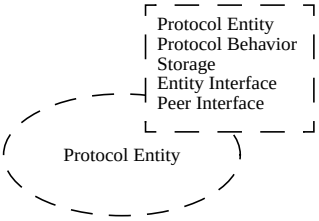

Fig. 38. Protocol Entity Pattern

Solution The functionality of an entity is devided to the following parts (see Figure 38 on page 58). A Protocol Entity Pattern contains a Protocol Entity, a Protocol Behavior and a

Storage, and it uses and provides Entity Interfaces and a Peer Interfaces. One Protocol Entity can be used as a layer in the Layers Architecture pattern [13].

Structure A structure of the Protocol Entity pattern is presented in Figure 39. *The Protocol Entity* represents object which manages the whole structure of system in concern.

*A Protocol Behavio*r handles protocol functionality. The Protocol Behavior is explained in detail in section "Protocol Behavior Pattern" on page 65.

A Storage contains all volatile and non-volatile information of a Protocol Entity. Information collected to Storage can be visible for the whole Entity or it can be split to dedicated parts. An example of this is connection specific information.

An Entity Interface handles communication between two Entities in the same protocol system. It interprets a Entity Message which is received from another Protocol Entity. It also produces a Entity Message which is sent to another Protocol Entity in the same system.

A Peer Interface handles communication between entities located in the peer protocol system. It interprets a Peer Message which is received from a peer Entity. It also produces a Peer Message which is sent to another Entity in a peer system.

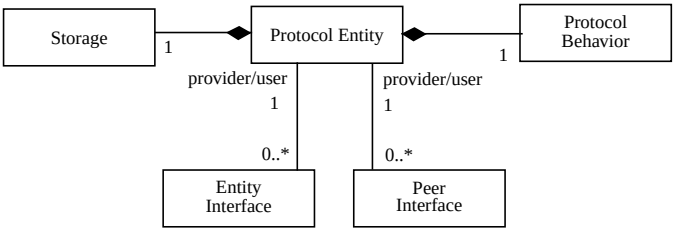

Fig. 39. Protocol Entity Pattern Structure

The Entity Message and *the Peer Message* are parts of "Protocol Message Flow pattern" on page 72.

Implementation Following section contains implementation examples of Protocol Entity pattern in four differnt implementation framework. These frameworks are explained in detail in chapter "Tools for Protocol Implementation" on page 19.

Conduits

A Protocol Entity can be modeled by one or more Protocol conduits and any number of Muxes and ConduitFactories. These conduits define the Protocol Behavior of a Entity. All of the conduits can contain ordinary variables which can be used to store protocol specific information.

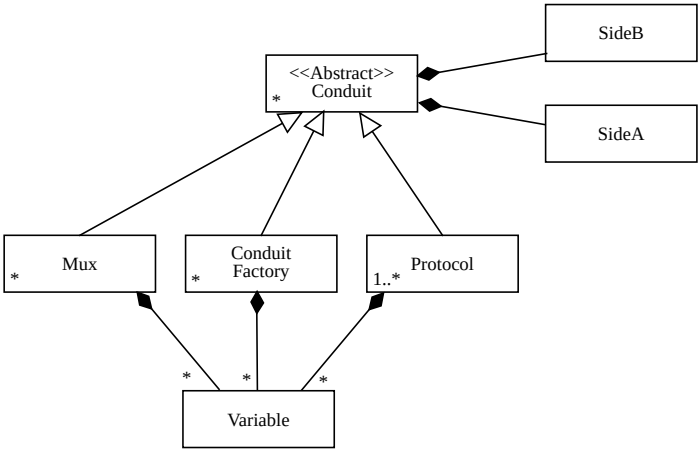

Fig. 40. Conceptual Protocol Entity diagram

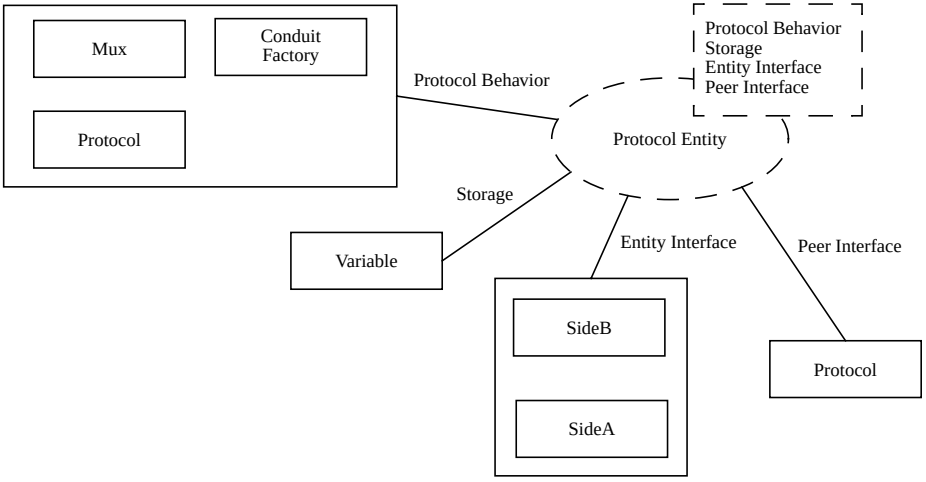

Fig. 41. Conduits Protocol Entity aspattern

CVOPS

A layer consists of one or more vTasks. A single vTask acts either as a connection or a connection manager, or, in case of a connectionless protocol, as the whole layer. The roles of connection and manager are predefined, and the implementor has to conform to certain rules when defining the vTasks. The attributes of a layer that are common for both the manager and the connections of a layer are stored in a structure called task type. Any vTask specific information can be stored in its internal variable structure. A vTask can handle both service messages (called primitives in CVOPS) and PDUs.

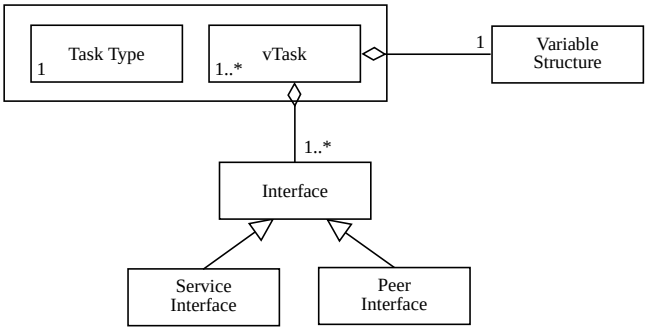

Fig. 42. Conceptual CVOPS Protocol Entity diagram

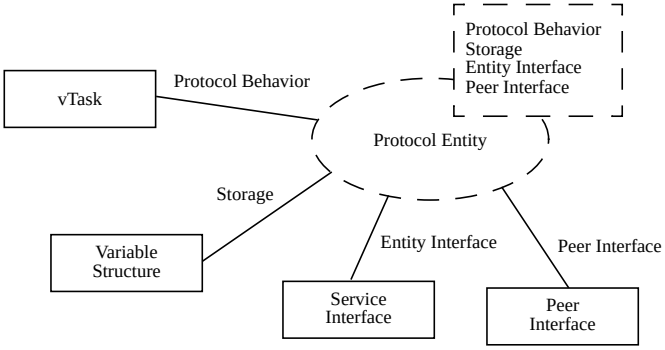

Fig. 43. CVOPS Protocol Entity pattern

OVOPS

A protocol entity consists of pTasks, its variables and its behavior, and the ports by which it is connected to adjacent layers. Depending on the protocol, a layer consists of one or more pTasks. Each pTask can have a arbitrary amount of variables that may act as protocol storage. PTasks can share ports. This means that two or more pTasks may have, for example, the same up port for communication with the upper layer. This can be used in connection- oriented protocols where the connections of a layer may have the same up and down ports as the manager.

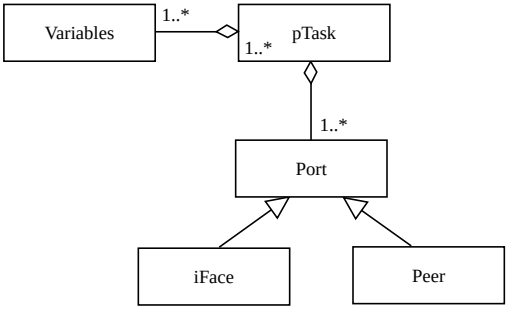

Fig. 44. Conceptual Ovops protocol entity diagram

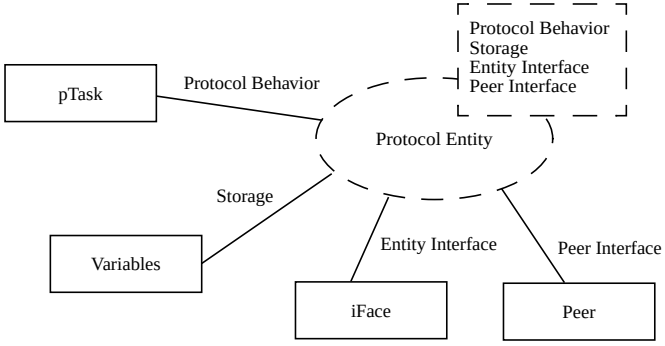

Fig. 45. Ovops protocol entity pattern

SDL

A block can contain several processes or process sets. The behavior of a block is not explicitly defined but it can be derived from the behavior of its processes.

The processes model the Protocol Behavior. Also the manual Peer Interface functions are implemented in them. The processes may contain attributes which are used to store information.

The processes or process sets are connected with each other using signal routes. The signals carrying information are conveyed by these routes from the sender process to the receiver. Processes inside a block communicate with the block environment, i.e. the system outside them, also using the signal routes. The routes linking processes to the block environment are attached to channels which are on the upper abstraction level connected to other blocks or the system environment.

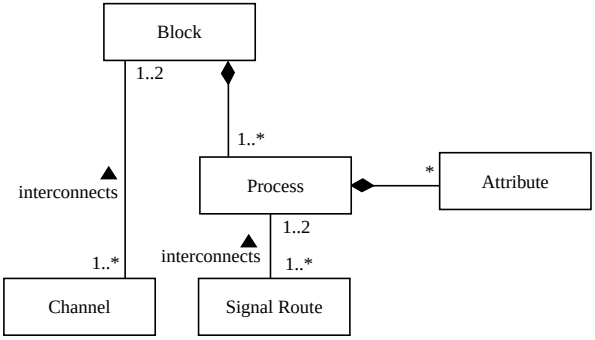

Fig. 46. Conceptual SDL Protocol Entity diagram

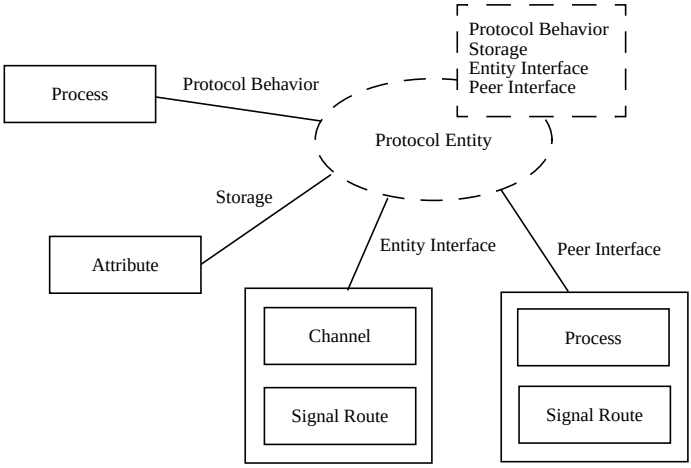

Fig. 47. SDL Protocol Entity pattern

Consequences The Protocol Entity pattern has several advantages, but it also contains a few disadvantage. These are quite similar as one layer in the Layers Architecture pattern [13].

Benefits

- Reuse of Protocol Entities. If an individual entity embodies a well-defined abstraction and has a well-defined and documented interface, the entity can be reused in multible context.
- Support for standardization. Cleary-defined and commonly-accepted levels of abtraction enable the development of standardized tasks and entity interfaces. Different implementations of the same entity interface can be interchanged.
- Dependencies are kept local. Standardized entity interfaces between layers usually confine the effect of code changes to the Protocol Entity that is changed. Changes in the environment affect only Environment Interface in concern.
- Exchangeability. Individual Protocol Entity implementations can be replaced by semantically equivalent implementations without too great effort.

Liabilities

- Cascades of changing behavior. Entities can often be shielded from changes in other entities. However sometimes a change ripples from one entity to all others. In this

case the Protocol System pattern becomes a disadvantage if a substantial amount of rework has to be done on many entities to incorporate an apparently local change.
- Lower efficienty. An architecture, which is implemented using the Protocol System pattern, is usually less effiecient than a monolithic implementation. This is result from the large number of operations and transformations the Protocol Message Flow experiences when it passes throught the entities.
- Unnecessary work. Some services performed by previous enties may not needed by adjacent entities. This has of cource a negative impact on performance.

See also The Protocol Entity pattern uses the Protocol Behavior pattern (presented in "Protocol Behavior Pattern" on page 65) and the Protocol Message Flow pattern (presented in "Protocol Message Flow pattern" on page 72).

8.3 Protocol Behavior Pattern

Name Protocol Behavior

Context

Problem

Solution The Protocol Behavior pattern is used to implement needed protocol functions for the system in concern. A Protocol Behavior is typically specified as a state machine. A protocol behavior can be divided into two main modes: connectionless and connection-oriented behavior. In connectionless behavior a communication session is a simple Request-Response or just Request) kind of message exchange.

In connection-oriented behavior a communication session consists of connection establishment, message exchange, and finally disconnection phase. There can be multible concurrent communications which can be in different phases. A special case of connection-oriented behavior is a case when a connection may have sub-connections.

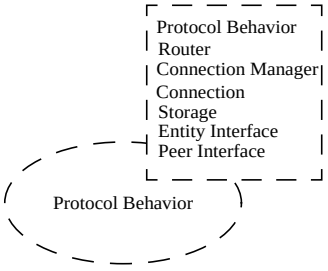

Fig. 48. Protocol Behavior Pattern

Structure The connectionless Protocol Behavior contains one Connection Manager which handles all communication events.

The connection oriented Protocol Behavior contains *Router*, *Connection Manage*r, and zero or more *Connections*. A Connection Manager-Connection pattern is used to manage multiple concurrent communication sessions (defined in section "Connection Manager-Connection Pattern" on page 81). The structure of the Protocol Behavior pattern is presented in Fig. 49. Both Connection Manager and Connection can use Event Response pattern (defined in section "Event-Response Implementation Idiom" on page 93).

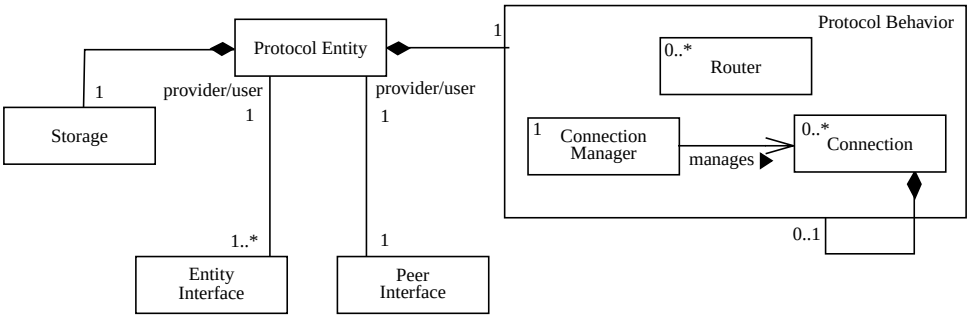

Fig. 49. Protocol Behavior Pattern Structure

Implementation Following section contains implementation examples of Protocol Behavior pattern in four differnt implementation framework. These frameworks are explained in detail in chapter "Tools for Protocol Implementation" on page 19.

Conduits

The Connectionless Protocol Behavior is modeled by a single Protocol conduit. There is no need for a ConduitFactory on this kind of layer. However, if the service of this layer is wanted to be available for multiple parallel upper level protocols an additional Mux, sideB upwards, is needed between the upper protocols and the Protocol conduit of this layer.

A connection-oriented protocol requires a combination of Protocols, Muxes and ConduitFactories. A Protocol represents a single Connection and a Mux multiplexes messages from several connections on sideB to the neighbor conduit on sideA. When messages arrive from the sideA neighbor the Mux must find the correct receiver and direct the message to it. This procedure is called demultiplexing. A ConduitFactory creates new Protocol conduits and attaches them to the conduit graph, usually to the sideB of a Mux.

The Protocol EFSA is modeled by a pointer in a Protocol conduit to a single State entity. These State entities can be shared by multiple Protocol conduits since they do not contain any session specific storage but provide the required functionality.

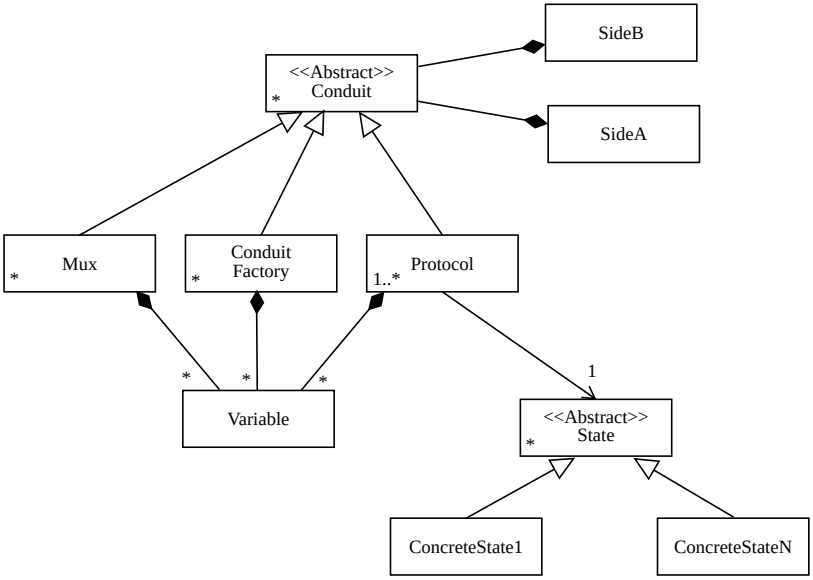

Fig. 50. Conceptual Conduits Protocol Behavior diagram

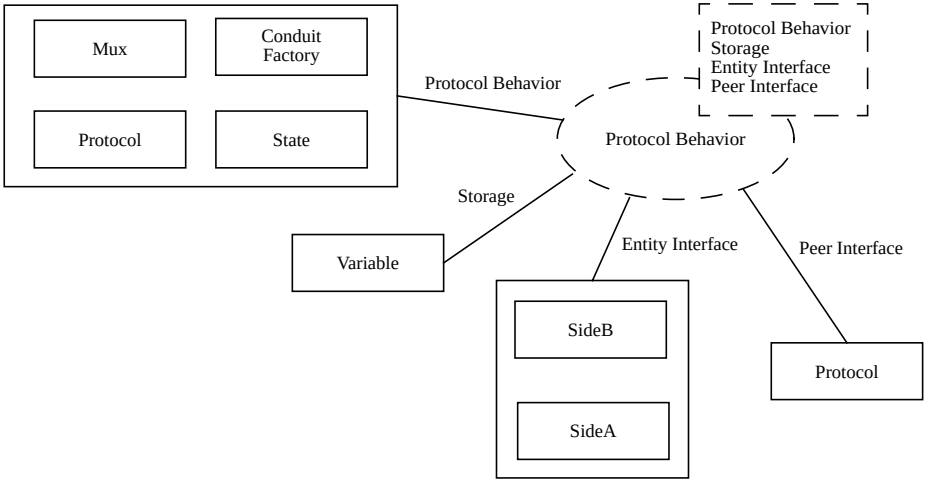

Fig. 51. Conduits Protocol Behavior as pattern

CVOPS

A connection oriented protocol is implemented using a connection manager (called entity in CVOPS) and connections. The connections manager creates and destroys the connections and routes the messages to the appropriate connection. The routing function has to be defined by the implementor. No multiplexer port or object is explicitly needed, because the roles manager and connection are predefined and the manager already contains such a structure. A connetionless protocol layer is implemented using a single manager (entity) for the entire layer [25]. An EFSA is associated with each vTask, so that every manager and every connection has their own. The EFSA is usually implemented using a EFSA language which the CVOPS core is able interpret. The EFSA language reminds somewhat of C, but it is more limited. The EFSA can also be implemented in plain C.

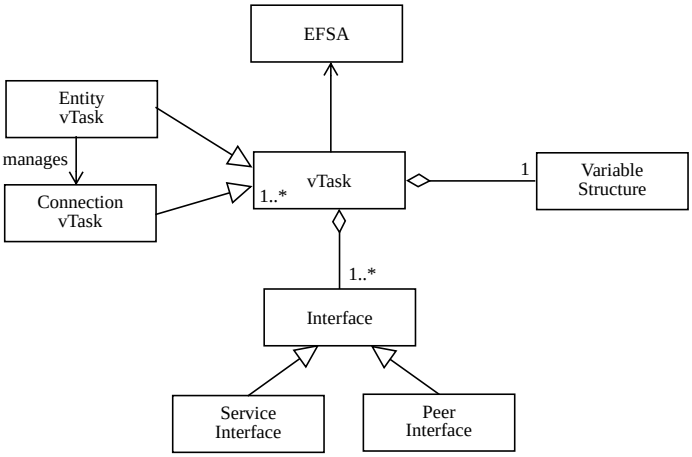

Fig. 52. Conceptual CVOPS protocol behavior diagram

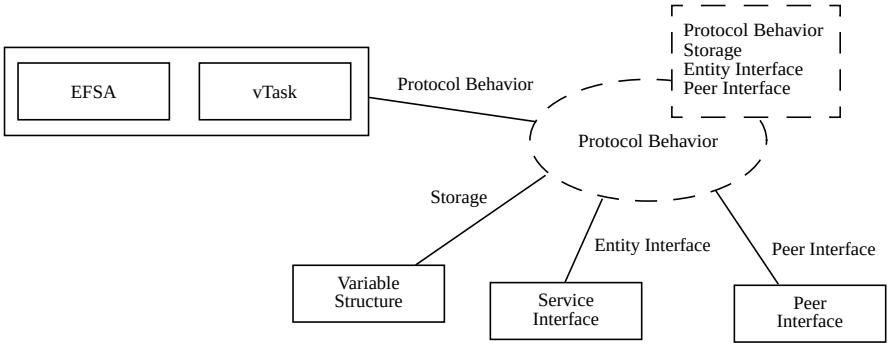

Fig. 53. CVOPS protocol behavior as pattern

OVOPS

A connection oriented protocol is usually implemented using managers and connections. In Ovops, the manager usually creates and destroys the connections, and routes the messages with the help of a multiplexer to the wanted connection. Alternatively, the routing can be made by connecting the ports of adjacent layers directly together, allowing the messages to pass from connection to connection without (de)multiplexing them in between. There is no predefined distinction made between managers and connections. The role is defined solely by the implementor. A connectionless protocol is usually implemented using a single pTask for the entire protocol layer. The pTask contains a set of state-input functions, and information about the current state. The statemachine identifies the incoming message and port, and invokes the correct function of the pTask. This means that one method is required for each incoming message or PDU for every state of the state machine.

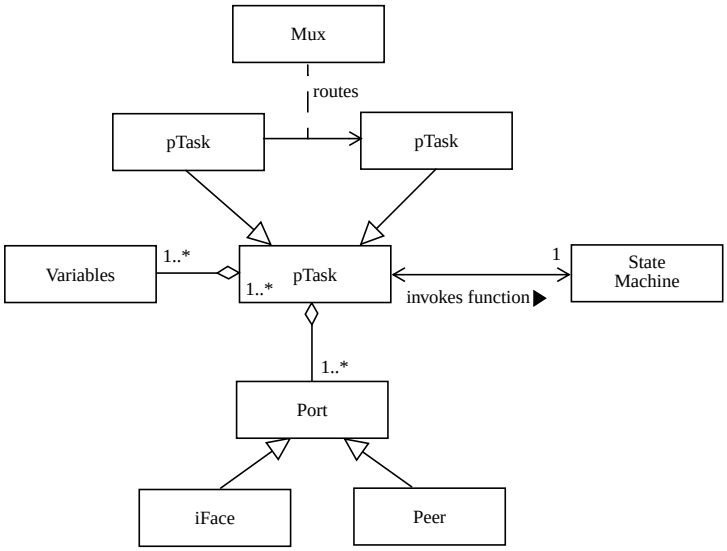

Fig. 54. Conceptual OVOPS Protocol Behavior diagram

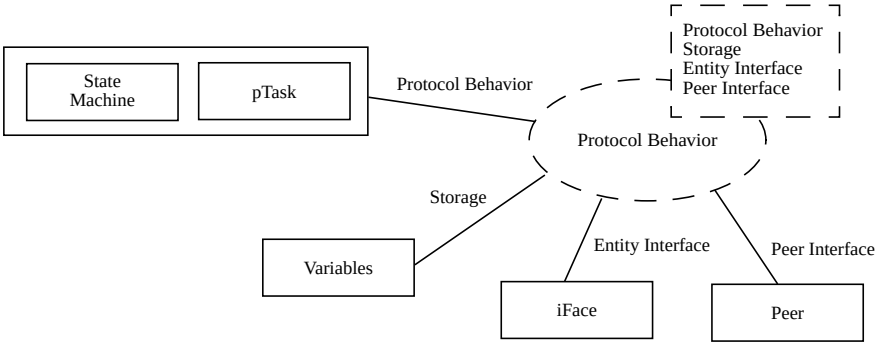

Fig. 55. OVOPS Protocol Behavior as pattern

SDL

A single process can be used to model the behavior of a connectionless protocol. A connection-oriented protocol requires a separate manager process that manages the sessions and acts as a mux, if necessary.

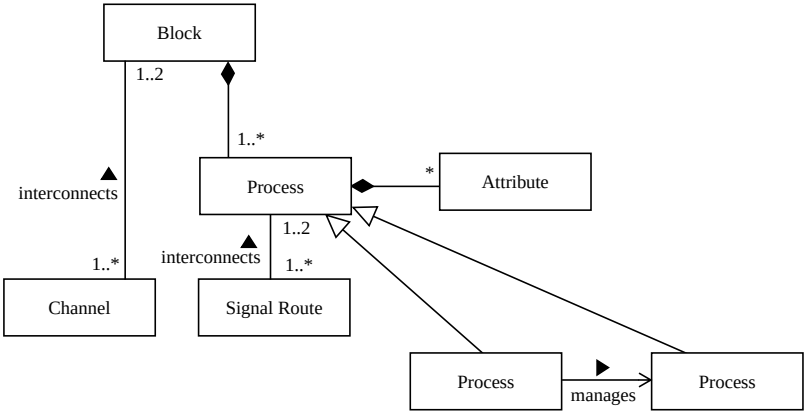

Fig. 56. Conceptual SDL protocol behavior diagram

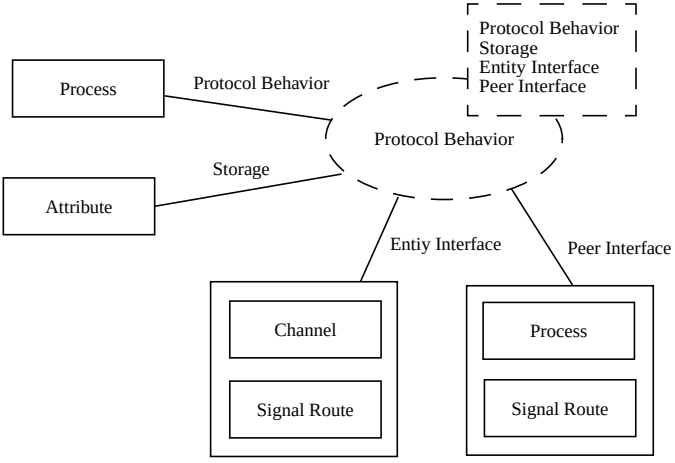

Fig. 57. SDL protocol behavior as pattern

Consequences

See also

8.4 Protocol Message Flow pattern

Name Protocol Message Flow

Context

Problem

Solution XXX

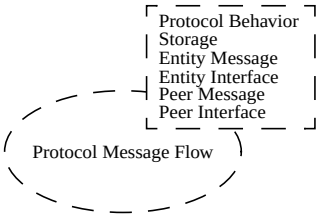

Fig. 58. Protocol Message Flow Pattern

Structure Classes which are related to data flow, are presented in Figure 59. These are Storage, Protocol Behavior, *Protocol Message*, *Entity Message*, *Peer Message*, *Service-only Parameter*, *Peer-only Parameter*, *Shared Message*, *Local Syntax*, and *Transfer Syntax*. One Message type and one Interface type form a Message-Handler pattern (defined in “Message-Handler Pattern” on page 86).

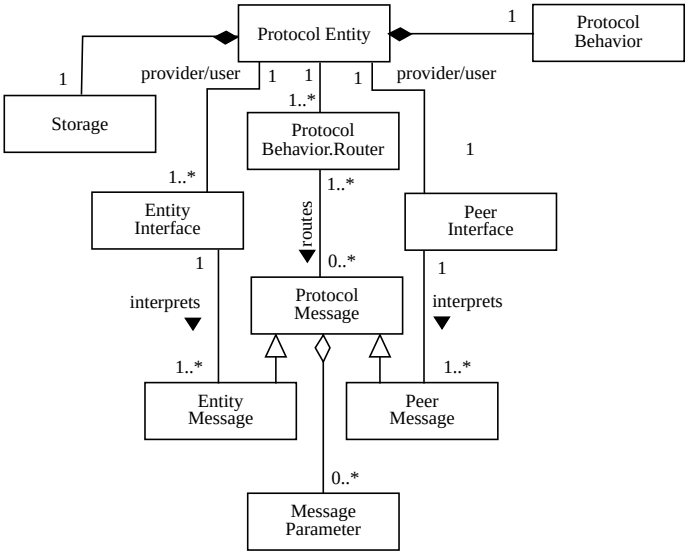

Fig. 59. Protocol Message Flow of a protocol system

The classes of the Protocol Message Flow (from Figure 22):

A Router determines proper receiver for a Protocol Message in case of connection-oriented communication.

A Protocol Message is subclassed to a Service Message and a Peer Message.

A Entity Message is used between two entities in the same system. It contains data and control information.

A Peer Message (aka PDU) is used between two entities in the different systems.

A Message Parameter contains all message parameters.

Implementation Following section contains implementation examples of Protocol Behavior pattern in four differnt implementation framework. These frameworks are explained in detail in chapter “Tools for Protocol Implementation” on page 19.

Conduits

Raw Information Chunks are just arrays of bytes and cannot possess any methods the framework needs. Therefore in the Conduits+ framework the Information Chunks are encapsulated in a Messenger that contains the required functionality and has the ability to

apply itself when it arrives in a conduit. The Messenger is also used to identify different messages.

Messages, that is Messengers with Information Chunks, are created in the Adapters as a result of the interaction with the environment and in the Protocols as a result of for example some input.

The Peer Messages come to a Layer inside an Information Chunk of a Service Message. When a entity Messenger carrying a PDU applies itself to a Protocol conduit its Information Chunk is examined, the corresponding peer Messenger is created and then the Protocol conduit sends the new peer Messenger to itself. The actions that a Peer Message triggers are defined in the current State entity of a Protocol.

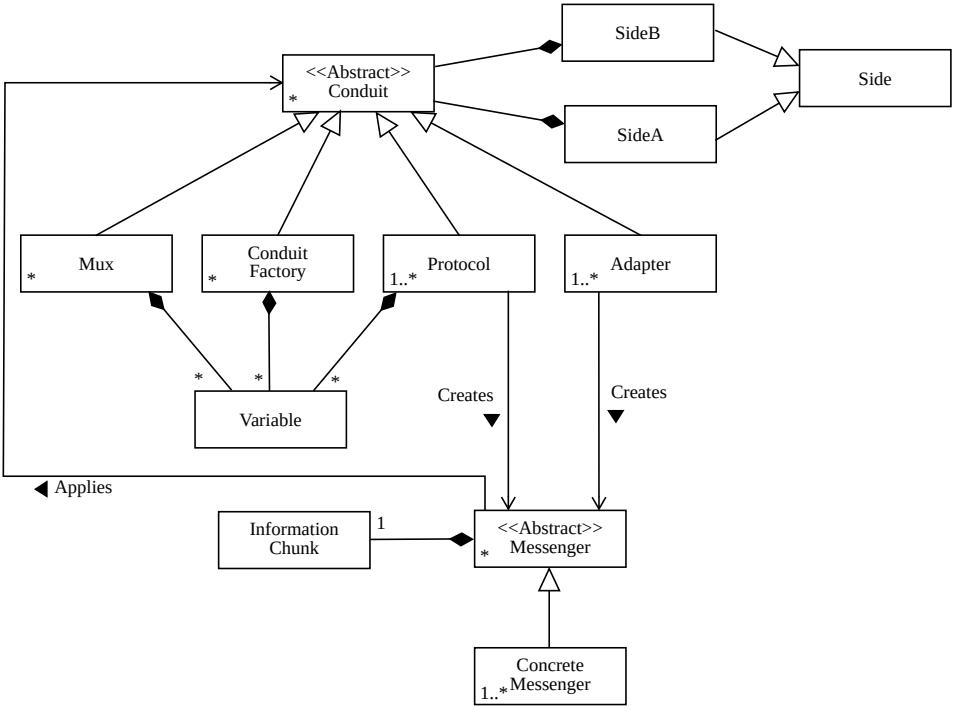

Fig. 60. Conceptual Conduits protocol data flow diagram

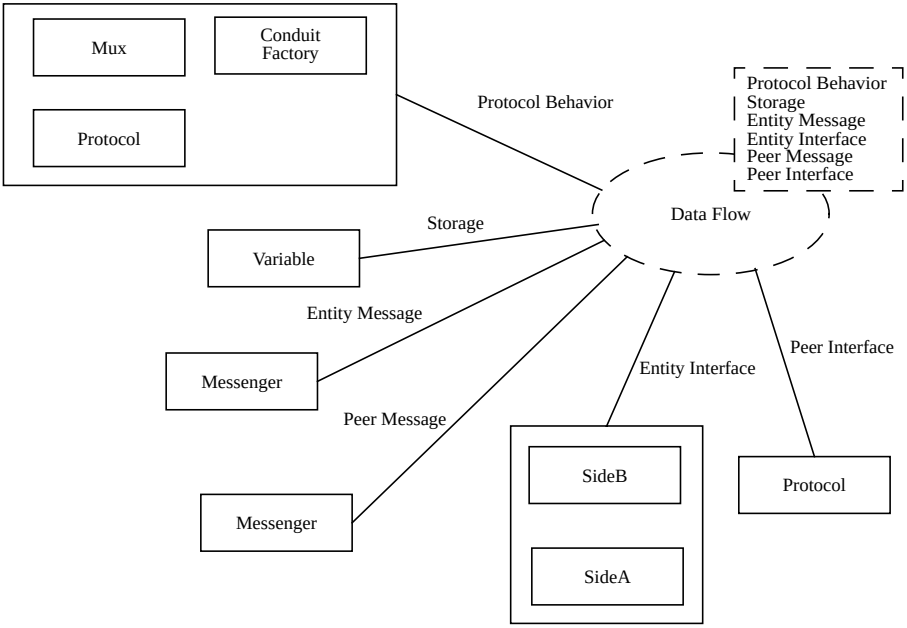

Fig. 61. Conduits protocol data flow as pattern

CVOPS

CVOPS offers two kinds of interfaces, entity interfaces and peer interfaces. The peer interface is an abstraction, and according to the OSI model PDUs are encoded and sent as a parameter in a lower layer data message. Encoding and decoding is initiated by the EFSA. Every primitive and PDU has a parameter block. This block is used for storing the data to be transmitted with the message. The internal variables of the vTask are used as storage when handling PDUs and entity messages. When the PDU is decoded the needed parameters are copied to the variables of the vTask. Likewise, during encoding the internal variables are copied to the PDU parameters. The parameters of the service primitives are also copied to and from the internal variables of the vTask.

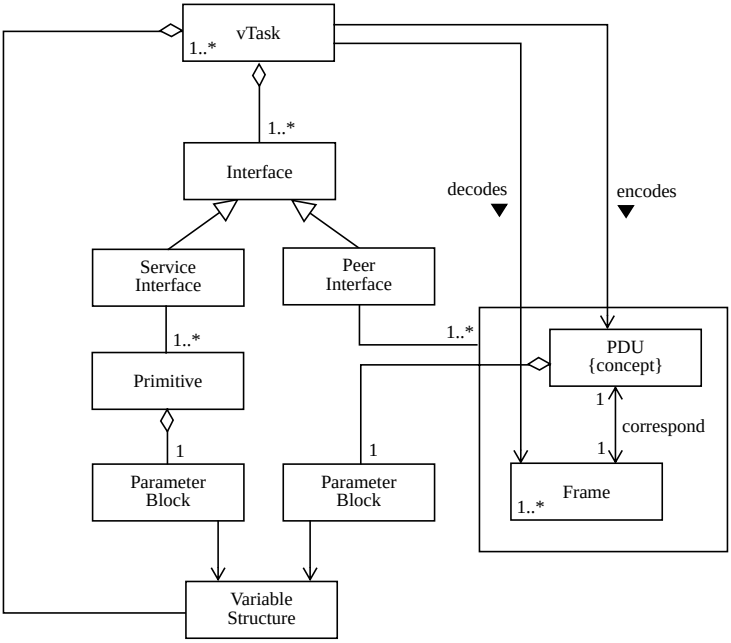

Fig. 62. Conceptual CVOPS data flow diagram

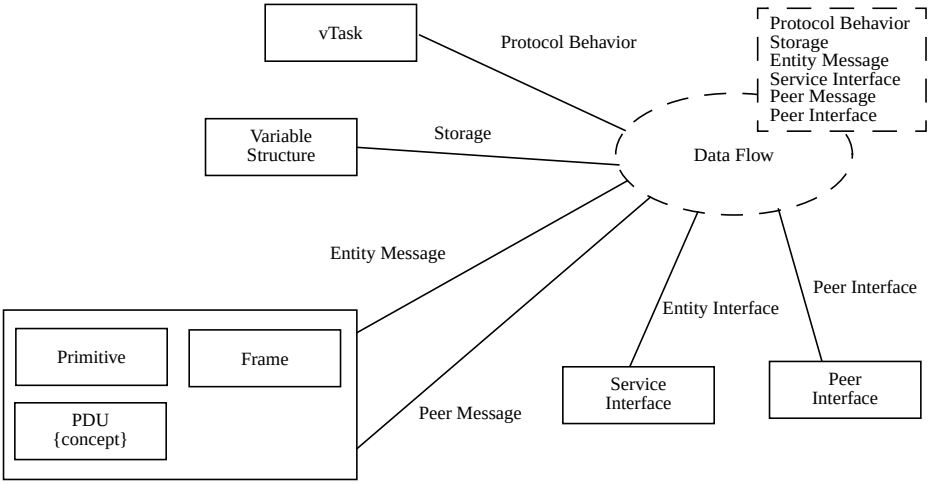

Fig. 63. CVOPS data flow as pattern

OVOPS

The PTB offers two kinds of ports - interfaces and peer ports. A interface is basically a port combined with a number of messages associated to that port. Interfaces are used for communication between layers. Peer ports are used for peer to peer communication. The peer port is an abstraction, designed to hide some parts of the behavior. In effect, a peer port identifies the type of the incoming PDU and invokes the encoding and decoding methods of the PDUs. The PDUs are encapsulated inside lower layer entity messages.

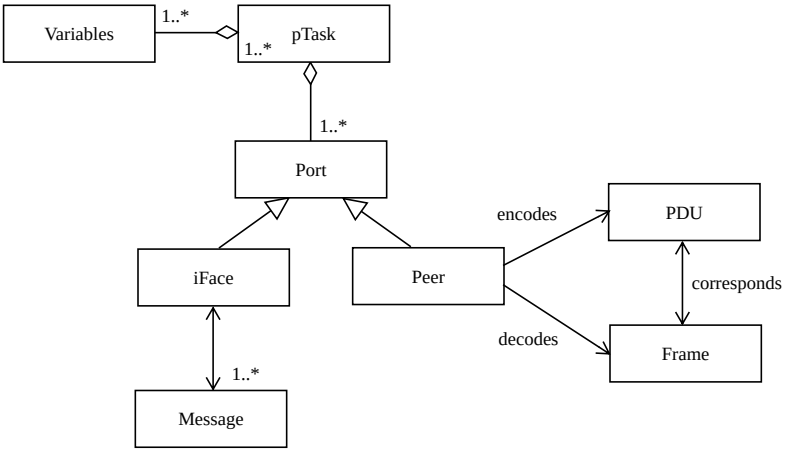

Fig. 64. Conceptual Ovops protocol data flow diagram

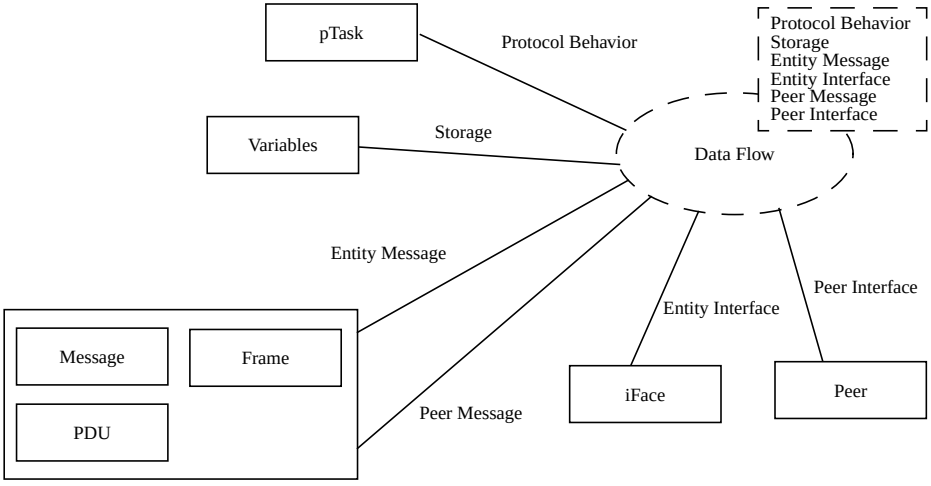

Fig. 65. Ovops protocol data flowas pattern

SDL

In an SDL protocol system the signals are used as entity messages. The signals either originate from the environment or are created by the processes e.g. as a result of an input signal. There is no distinct structure for PDUs nor a separate local syntax and transmission syntax can be defined. Therefore PDUs must be placed, for example as a strings of octets, in the parameters of signals. The signal parameters can also be used to transmit service-only parameters with a entity message.

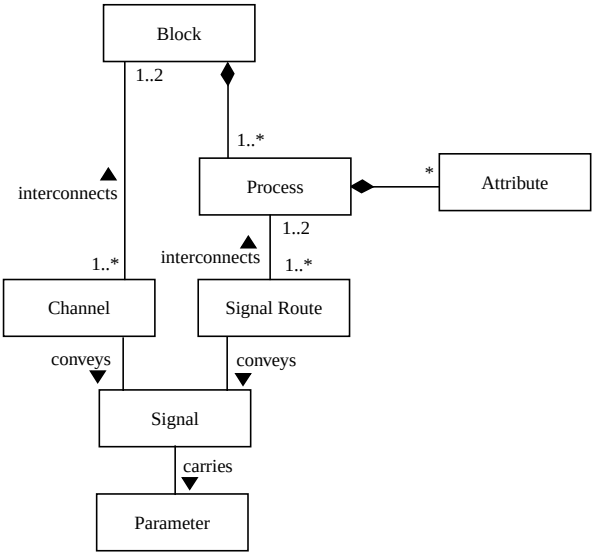

Fig. 66. Conceptual SDL Data Flow diagram

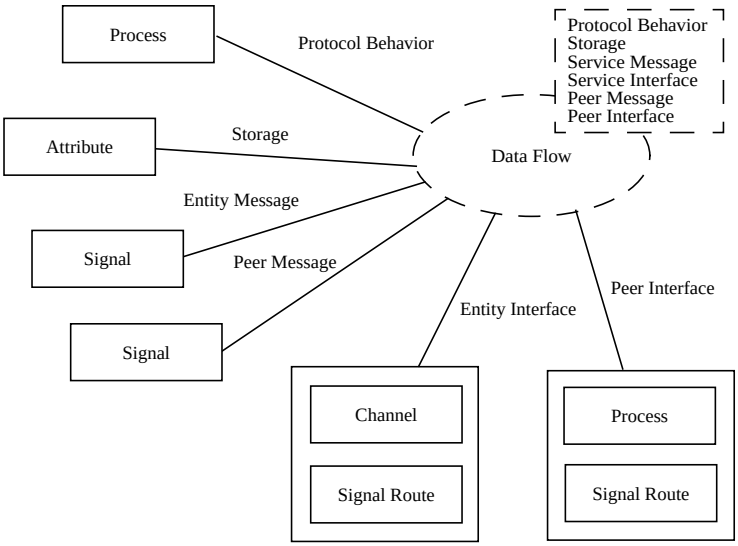

Fig. 67. SDL Data Flow as pattern

Consequences

See also

Chapter 9 Protocol Design Patterns

XXX

9.1 Connection Manager-Connection Pattern

XXX

Name Connection Manager-Connection

Context he Connection Manager- Connection pattern can be used as a compositional part of a Protocol Entity pattern.

Problem Protocol ntities may have to handle many concurrent communication sessions. These can have different communication parties, properties, and state of communication.

Solution he system contains a Communication Manager and Connections. The role of the Connection Manager is to accept requests for connections and create new Connection objects for them. The Connection Manager receives and interprets messages which cause a creation of a new Connection. The role of Connection is to remember the state of a job and act according to incoming requests and current connection state.

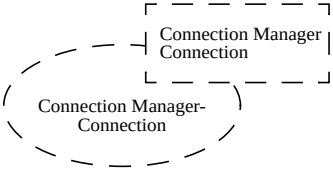

Fig. 68. Connection Manager-Connection Pattern

Structure

XXX, see Figure 69 on page 82.

Implementation There are two different models for an implementation of this pattern in the Protocol Entity: the Pipe Model and the Router Model. In these models the Manager is instantiated as the Connection Manager, and the Worker is instantiated as the Connection.

Fig. 69. Connection Manager-Connection Pattern Structure

In the Pipe Model connections of two entities communicate each other directly, see Figure 70 on page 82. This means that when a connection is created it is the responsibility of the Connection Manager to couple two Connections together. After that Connections communicate directly with each other.

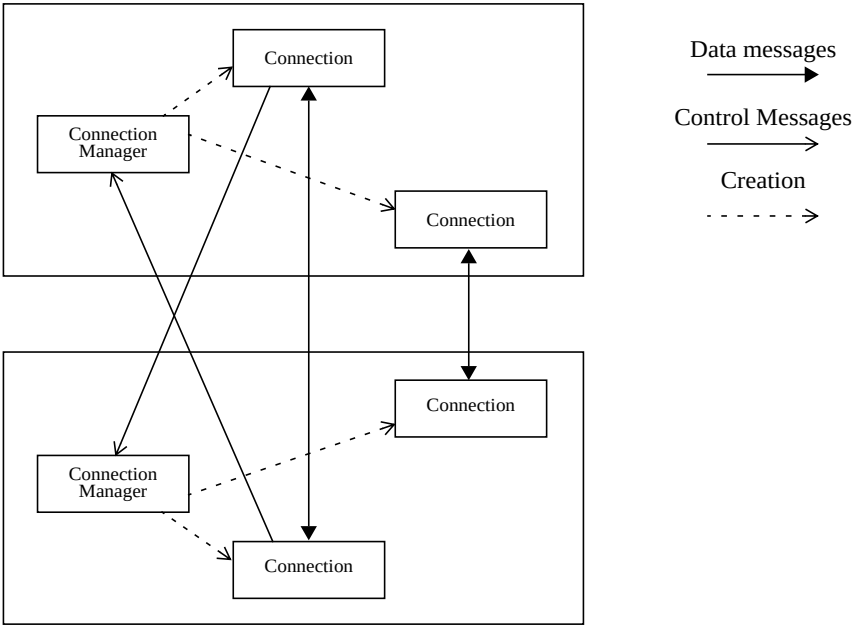

Fig. 70. The Pipe Model

In the Router Model a Connection Manager routes all the incoming messages to proper connections, see Figure 71 on page 83. Connections communicate only with other entities' Connection Managers.

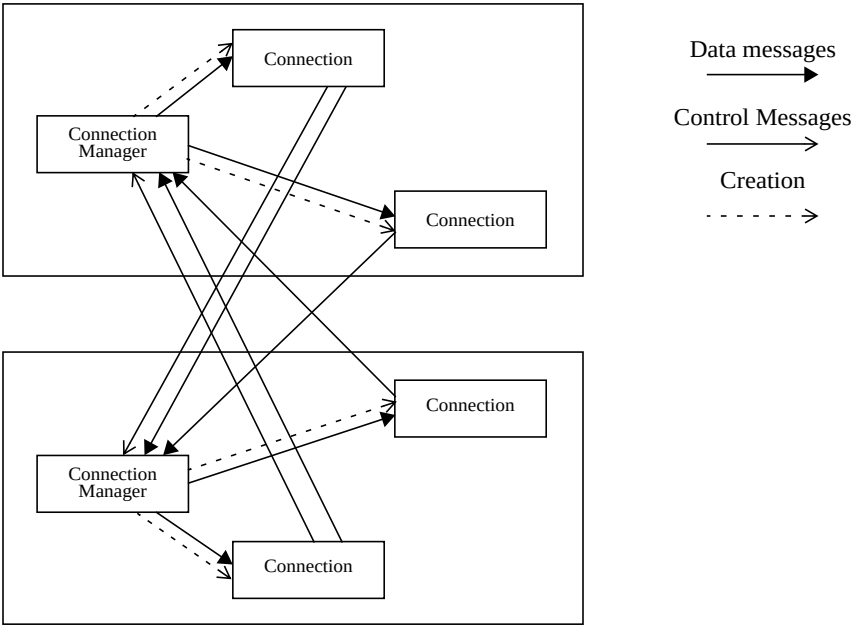

Fig. 71. The Router Model

Conduits

Fig. 72. Conceptual diagram

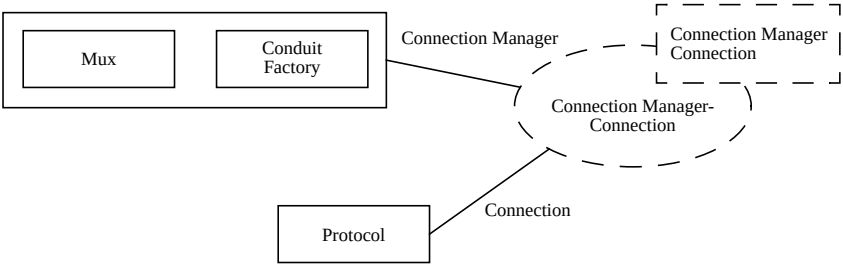

Fig. 73. Connection Manager-Connection pattern

CVOPS

Fig. 74. Conceptual diagram

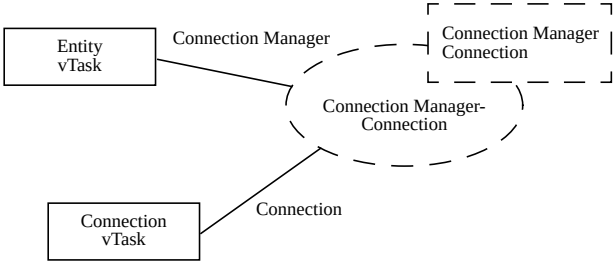

Fig. 75. Connection Manager-Connection as pattern

OVOPS

Fig. 76. Conceptual diagram

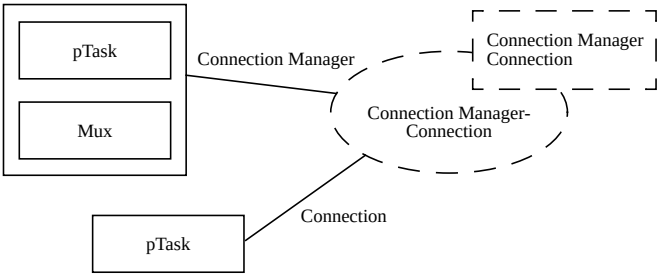

Fig. 77. Connection Manager-Connection as pattern

SDL

Fig. 78. Conceptual diagram

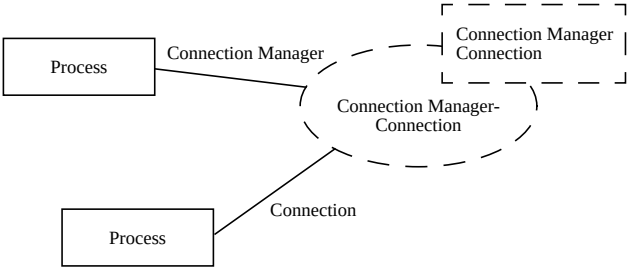

Fig. 79. Connection Manager-Connection as pattern

Consequences

See also This pattern is a variation of the Master-Slave pattern [13].

9.2 Message-Handler Pattern

Name

Context The Message-Handler pattern is used as a compositional part of a Protocol Entity.

Problem A Protocol Entity have to communicate with other entities in the same system and peer systems.

Solution The role of a Handler is to interpret Messages which are sended between local or remote systems. The role of a Message is to encapsulate all information wanted to sent to other system.

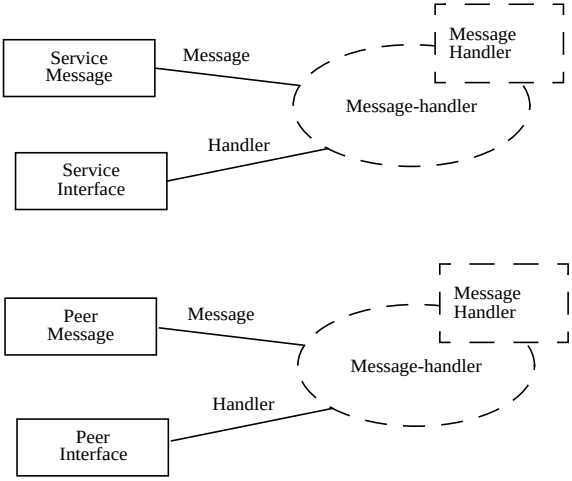

Fig. 80. Message Handler Pattern

Structure

Implementation

Conduits

Fig. 81. Conceptual diagram

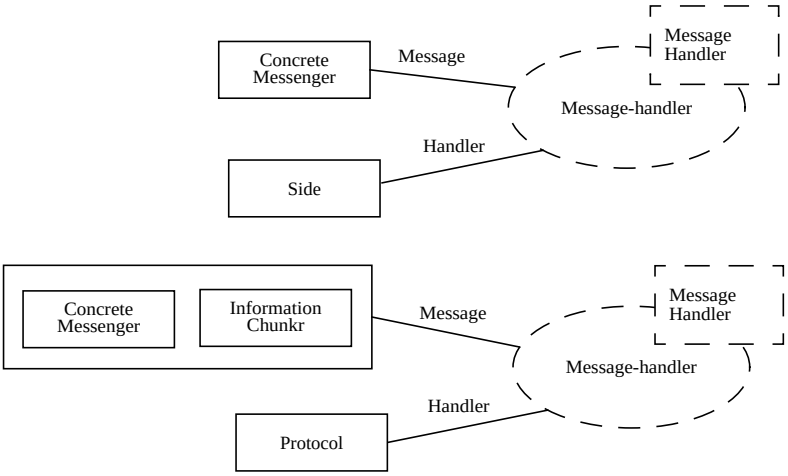

Fig. 82. Message-handler pattern

CVOPS

Fig. 83. Conceptual diagram

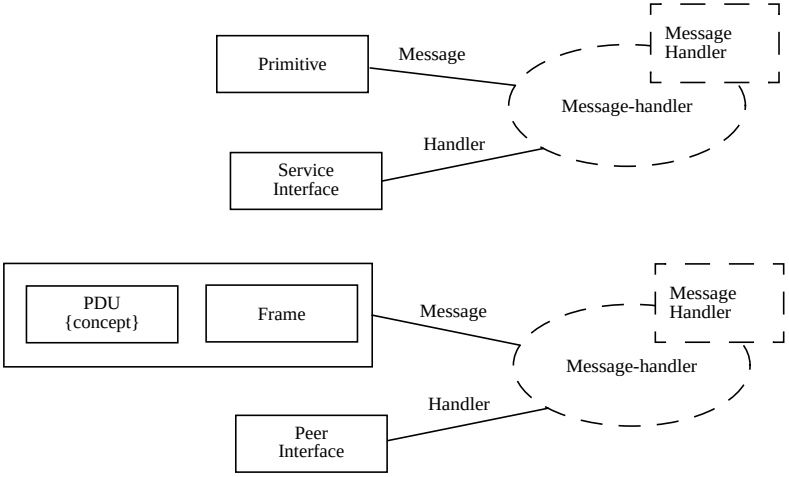

Fig. 84. Message-handler pattern

OVOPS

Fig. 85. Conceptual diagram

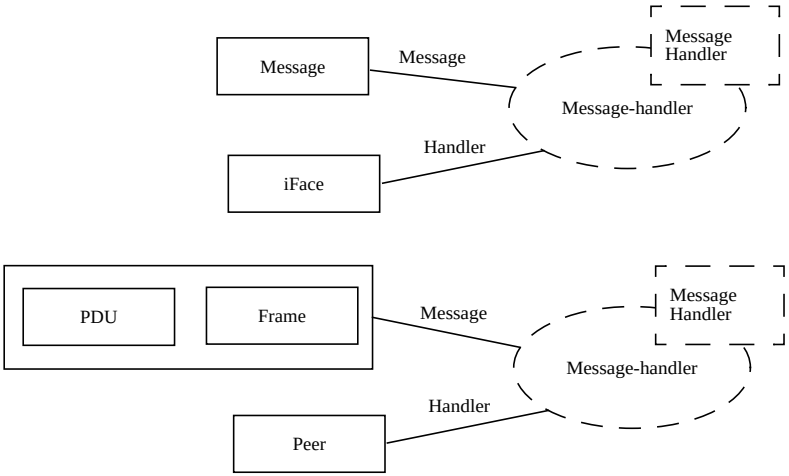

Fig. 86. Message-handler pattern

SDL

Fig. 87. Conceptual diagram

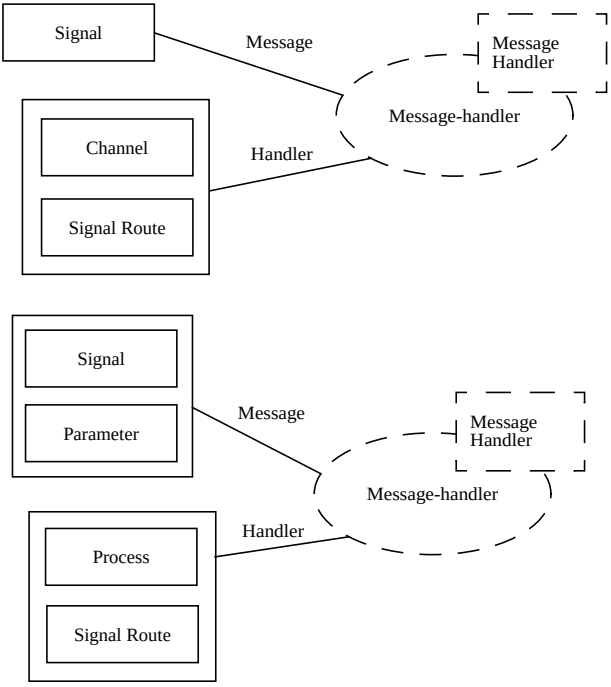

Fig. 88. Message-handler pattern

Consequences

See also

Chapter 10 Protocol Idioms and Algorithms

XXX

10.1 Event-Response Implementation Idiom

Name

Context

Problem

Solution

Implementation

Consequences

See also

Context. The Event-Response can be used to implement the Protocol Behavior.

Problem. For a Protocol Entity which receives events (i.e messages) from its peer systems, there are various things you have to concern when implementing response to event. These things don´t depend the platform you use, they are depended one the inner nature if the system state and event relation.

Solution. One solution is presented in Figure 20. Actions are explained in Table 1

TABLE 1. Statemachine actions

{ }	Groups optional part. occurs zero or one time.
EntryNActions	These are usually subset of test variable, set variable, and stop timer.
ExitNActions	These are usually subset of set variable and start timer.
EventNMActions	These are event specific actions.
EventNMGuard	This is test if this event is possible.

TABLE 1. Statemachine actions

StateActionN	These are system´s state specific actions. An action may actually be a call of function or procedure which may contain sending of messages and state transitions. For example it is convenient to separate exception handling from the actual protocol logic.
MessageNM	This is a message which is sended in response to event in concern.

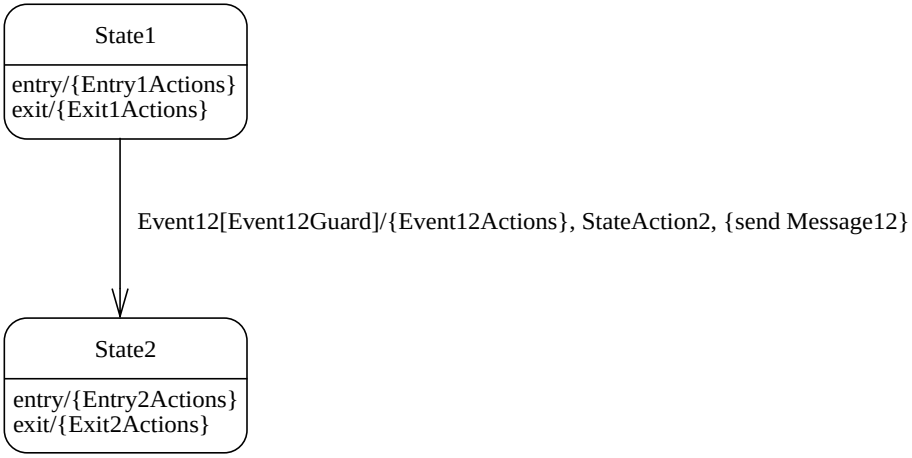

Fig. 89. Response to an event

10.2 Pipe Implementation Idiom

Name

Context

Problem

Solution

Implementation

Consequences

See also

10.3 Router Implementation Idiom

Name

Context

Problem

Solution

Implementation

Consequences

See also

10.4 Mux Implementation Idiom

Name

Context

Problem

Solution

Implementation

Consequences

See also

10.5 Encoding7Decoding Implementation Idiom

Name

Context

Problem

Solution

Implementation

Consequences

See also

10.6 Message Identification

Name

Context

Problem

Solution

Implementation

Consequences

See also

External identification is information, which is used to identify the type of an encoded PDU so that a receiver can decode the PDU correctly. A PDU may contain an explicit header that contains the id or the id can be stored in an enclosing PDU as a separate id field.

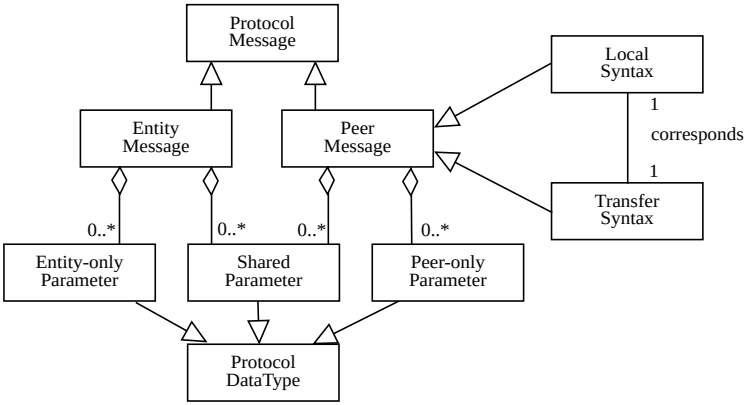

Fig. 90. External and internal identification

Example: OSI transport TCR and TCA PDU's

Example: (simplified) GSM MAP location update's ASN.1 definition and lower level structure that includes procedure call's argument:

```
UpdateLocation OPERATION ::= {
               ARGUMENT      UpdateLocationArg
               RESULT        UpdateLocationRes
               ERRORS        {
                             systemFailure |
                             unexpectedDataValue  |
                             unknownSubscriber    |
```

```
                              roamingNotAllowed
                              }
              CODE            2               --operation code
}

Invoke ::= SEQUENCE {
              invokeId  InvokeId,
              linkedID  [0] IMPLICIT LinkedId OPTIONAL,
              operation OperationNo,          --operation code
                                              -- passed here
              parameter ANY OPTIONAL          --operation
                                              --parameter value
                                              --passed here
}
```

Framework specific internal IDs for PDUs and primitives are used to map the incoming message to the framework message set. Frameworks provide means to define internal identifiers for PDUs and primitives. Internal identifier can be as simple as a unique number (CVOPS [26]) or it can be a special message specific class (conduit) or a special framework language construct (SDL signal [30]).

Internal identifiers for PDUs are relevant mostly in those frameworks, which provide direct support for encoding and decoding of PDUs. Such frameworks are for example CVOPS [24] and OVOPS [27]. Usually there is one-to-one mapping between external and internal identifiers.

For example in CVOPS and OVOPS internal ID is used as index.

CVOPS: Internal identifiers (TCR, TCA, TCC) are defined directly as numbers.

```
/*
$ pdus
*/
#define TCR 0
#define TCA 1
#define TCC 2
```

OVOPS: Internal identifiers (TSLBeginPDU, TSLContinuePDU etc) are defined as symbolic names, which have internal index values like in CVOPS:

```
Messages = {
              TSLBeginPDU = tslbeginpdu,
              TSLContinuePDU = tslcontinuepdu,
              TSLEndPDU = tslendpdu,
```

```
                TSLAbortPDU = tslabortpdu
};
```

When decoding a PDU a mapping from external ID to internal ID has to be done.

Example: a byte buffer (frame) contains an encoded TCR PDU value.

```
/*
* These define external identifiers.
*/
#define TCR_PDU_ID 0xE0
#define TCA_PDU_ID 0xD0



byte len;
byte id;
len = getFirstByte (frame);
id  = getFirstByte (frame);
switch (id) { /* External id ... */

                case TCR_PDU_ID:
                pdu = TCR; /* ... to internal id */

                break;
...
```

This is the simplest way to do the mapping between external and internal identifiers. There are other alternatives in which the mapping is explicitly specified. For example let's presume that there is a collection which contains one external id/internal id pairs. The collection provides fast operations for fetching one id according to the value of another id.

```
ExternalId find_external_id_by_internal_id (InternalId);
InternalId find_internal_id_by_external_id (ExternalId);
```

10.7 Internal id and primitive/PDU instances

Name

Context

Problem

Solution

Implementation

Consequences

See also

There is a relation between internal identification and external identification. The relation can be 1-1 or 1-N. This relation can be utilized for example in code generators.

For example in CVOPS's ASN.1 translator ASN2C generates so called entry description tables for PDU- and primitive types. The tables include all necessary information for transferring the values of types from protocol entity to another and for encoding+destruction/decoding+construction. PDU's and primitives internal ID's are used as indexes in tables.

Example: The ASN.1 to C translator ASN2C generates so-called entry description tables for PDU and primitive types. In case of the CVOPS framework the tables include information so that:

- service message parameters can be constructed from protocol storage values
- received service message parameters can be moved to protocol storage
- PDU parameters can be constructed and a PDU can be decoded
- a PDU can be decoded and parameters can be moved to protocol storage.

Internal ids are used as indexes in entry descriptor tables.

For Example in OVOPS sapg generates Ids for internal messages. Ids can be used in constructing message instances with a generic factory function.

```
/* developer written definition of provider message
TR_Begin_Ind
*/
Provider = {
              Messages = {
              TR_Begin_Ind = {

                            QualityofService *item
                            SCCPAddress *dest
                            SCCPAddress *source
                            OrigTransactionID origid,
                            ComponentPortion *components
              },


/* generated code by sap generator from above definition */
enum IndType { tR_Begin_Ind = 1, ...
```

```
TSLSAP::TR_Begin_Ind *trmsg =

              (TSLSAP::TR_Begin_Ind *)
              up.create(TSLSAP::tR_Begin_Ind);
```

10.8 Message routing

Name

Context

Problem

Solution

Implementation

Consequences

See also

Problem: There is a message and identification associated to that message. To which connection task does the manager task relay the message? How is that decision made?

Example: In CVOPS manager task's job is to find a suitable worker task. There is no prepared algorithm for this search. Linear search is the most common, but the following implementation would be more efficient with large number of connections.

ConnectionIdType is the type of the connection ID. (CVOPS's CEP_id), ExternalId-Type is the type of the external ID.

```
Typedef map<ConnectionIdType, ExternalIdType> IdMap;
Next datafield in manager task:
IdMap slaveIdMap;
```

Finding the worker task:

```
IdMap::iterator i= slaveIdMap.find (message.externalID);
ConnectionIdType slaveId;
If (i == slaveIdMap.end ()){
              slaveId = createConnection (master, message);
              slaveIdMap.insert (make_pair (slaveId,
                            message.externalId);
} else {
              slaveId = i->second;
}
```

For example in OVOPS the routing is a responsibility of a mux in the manager task. A function is registered to the mux and this is called for each worker task. The function decides if the message belongs to worker or not.

10.9 Protocol information

Name

Context

Problem

Solution

Implementation

Consequences

See also

Information associated with protocol can be either static or dynamic.

- Static information: for example ASN.1 defined remote operations and application contexts.
- Dynamic information: information from remote operation call.
- Both types have same handling needs: searching with specified criteria. Searching can be done with one or several keys.
- Data's saving method can be different because of size and handling needs. (for example: C++ array, vector<>, list<> or map<>)
- The data structures obtained can be complex. For example application context includes several operations. Application contexts together with operation packages determine whether the connections opener or receiver can use operations.

Application context

- Includes 0..4 operation packages
- Includes 0..3 operation sets
- Includes 1..N operations

If one operation can only belong to one application context how the operation including application context ID can be derived from operation ID? For example CVOPS MAP's derive_v1_ac_from_op_code function does exactly this, but completely manually.

Chapter 11 Concepts and Patterns in Protocol Engineering Process

XXX

Appendix A Used UML Notation

Bibliography

1. M. T. Rose, The Open Book, A Parctical Perspective on OSI, Prentice-Hall, 1990.
2. A. S. Tanenbaum, Computer Networks, 2nd edition, Prentice-Hall International, 1989.
3. W. Stallings, Data and Computer Communications, 4th edition, MacMillan, 1994.
4. F. Halsall, Data Communications, Computer Networks and Open Systems, 4th edition, Addison-Wesley, 1995.
5. ITU-T, Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model, Recommendation X.200, ITU, 1994.
6. D. E. Comer, Internetworking with TCP/IP Volume I: Principles Protocols, and Architecture, 3rd edition, Prentice-Hall International, 1995.
7. C. Alexander, The Timeless Way of Building, Oxford University Press, 1979.
8. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
9. D. C. Schmidt, Introduction to Design Patterns, http://siesta.cs.wustl.edu/~schmidt/patterns.html, 1997.
10. A. Weinand, E. Gamma, and R. Marty, ET++- -An object-oriented application framework in C++, Object-Oriented Programming Systems, Languages, and Applications Conference, San Diego, CA, pages 46-57, ACM Press, 1988.
11. R.E. Johnson, Frameworks = (components + patterns), Communications of the ACM, 40 (10): 39-42, October 1997.
12. M. E. Fayad, D.C. Schmidt, Object Oriented Application Frameworks, Communications of the ACM, 40 (10): 32-38, October 1997.
13. F. Bussman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, Wiley, 1996.
14. B. Appleton, Patterns and Software: Essential Concepts and Terminology, http://www.enteract.com/~bradapp/docs/patterns-intro.html, 1997.
15. H. Hüni, R. Johnson, R. Engel, A Framework for Network Protocol Software, ACM, 1995.
16. J. Zweig, An Object-Oriented Framework for Implementing Network Protocols, Master's Thesis, University of Illinois, 1991.
17. Java Conduits Beans (Jacob) Project home page, URL:http://www.tcm.hut.fi/Research/TeSSA/Jacob/jacob3.html, Helsinki University of Technology, 1998.
18. JVOPS home page, URL:http://www.necsom.com/products/jvops.html, Necsom Ltd., 1998.
19. TOVE Project home page, URL:http://www.tcm.hut.fi/Research/BBS/TOVE/, Helsinki University of Technology, 1998.

20. P. Nikader, A. Karila, A Java Beans Component Architecture for Cryptographic Protocols, Usenix Security Symposium '98, Helsinki University of Technology Laboratory of Telecommunications Software and Multimedia,1998.
21. B. Sahlin, A Conduits+ and Java Implementation of Internet Protocol Security and Internet Protocol, version 6, Master's Thesis, Helsinki University of Technology, 1997.
22. A. Karila, Portable Protocol Development and Run-Time Environment, Licentiate's Thesis, Helsinki University of Technology, 1986.
23. J. Malka, E. Ojanperä, CVOPS User´s Guide, http://www.vtt.fi/tte/tte22/cvops/, Technical Research Center of Finland, 1998.
24. Juhani Malka, Esko Ojanperä, User's guide for Cvops 6.0, XXX.
25. Juhani Malka, Esko Ojanperä, Advanced User's guide for Cvops 6.0, XXX.
26. Teleclass, Teleclass software homepage, http://www.teleclass.fi, 1998.
27. P. Heinilä, OVOPS Home Page, http://ovops.lut.fi, Lappeenranta University of Technology, 1997.
28. J. Sipilä, Ovops overview, XXX, 1998
29. O. Martikainen, P. Puro, J. Sonninen, The Ovops Environment for IN applications, 1994.
30. J. Ellsberger, D. Hogrefe, A. Sarma, SDL Formal Object-oriented Language for Communicating Systems, 1997
31. ITU-T, Recommendation Z.100 "Specification and description language (SDL)", 1993
32. SDL Forum Society home page, URL:http://www.sdl-forum.org/, SDL Forum Society, 1998
33. Rational Software, UML Resource Center, http://www.rational.com/uml/, 1997.
34. J. Rumbaught, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall International, 1991.
35. M. Fowler, UML Distilled, Addison-Wesley, 1997.
36. The OMG home page, http://www.omg.org, 1999.