Patterns for Protocol System Architecture

Juha Pärssinen and Markku Turunen

1 Abstract

When dealing with the specification, design, or implementation of a communication protocol, common parts can be found. These parts can be specified as design patterns which are independent from particular protocol specification, its implementation details, or used implementation language. It is the goal of this paper to present these parts, which are extracted from protocol specifications, designs, existing implementations, and data communication literature. These patterns provide common principles for understanding protocols and their parts, or implementing new ones. This paper can be read as a basic presentation of protocol design and implementation structure, but also advanced readers can get benefit from it.

In this paper three design patterns for a protocol system architecture are presented. These patterns are tightly related to each other. They are part of a pattern language for protocol implementation which the authors have developed during last years.

The Protocol Patterns presented in this paper include the *Protocol System pattern* which models a protocol system in general level, the *Protocol Entity pattern* which models active parts of a system, and the *Protocol Behavior pattern* which models communication between parts of a protocol system.

A basic model of communication protocol implementation and two implementation frameworks are presented in this paper to give background information. The Unified Modeling Language (UML) is used in the class diagrams of this paper, and the UML pattern notation is used to describe roles of an individual pattern.

2 Introduction

Communication protocols specify how two entities can communicate with each other, i.e. what is the set of messages they can interchange and what are the rules that govern message interchange. When different protocols are investigated common general parts and relations between them can be found. These parts and relations are independent from particular specification, implementation details or used implementation language. In this paper we present design patterns for a protocol system architecture.

Understanding of patterns behind a protocol system architecture facilitates creation of a clear, consistent, and more understandable architecture which is also efficient to implement and maintain. It facilitates integration of protocol components produced using different implementation frameworks, and helps developing protocol implementation tools. The patterns presented in this paper are currently used to implement code generation tools for a protocol engineering process [1].

These Protocol Patterns are derived by the authors from many different sources including actual protocol implementations, communication protocol implementation frameworks, existing protocol models, and related existing design patterns, including Layers architecture pattern [8].

Copyright © 2000, Juha Pärssinen, Markku Turunen.
Permission is granted to copy for the PLoP 2000 conference.
All other rights reserved.

Patterns presented here have been used in some forms in communication system engineering for decades and their original inventors are unknown as original reasons to take them in use. Because of this patterns are not presented typical context-forces-solution order, but in context-solution-forces which is more suitable in this case. In data communication literature solution has a main role, not problems itself.

This paper contains three main parts: the section 3 "Communication Protocol Structure" on page 3 contains background information about communication protocols, the section 4 "Protocol Implementation Frameworks" on page 5 contains information their implementation, and the section 5 "Patterns for Protocol System Architecture" on page 9 contains actual patterns. Advanced reader can jump straight to patterns.

A basic protocol model is presented in the section 3 "Communication Protocol Structure". This is the model which was used as a starting point to mine for these patterns. During the years many protocol implementation frameworks are developed to offer efficient and simple way to create and configure functionality of common tasks of protocols. These common tasks and two implementation frameworks, Conduits+ [10] and Specification and Description Language (SDL) [14], are presented in the section 4 "Protocol Implementation Frameworks". These frameworks are used as examples when we show how patterns presented in this paper can be implemented. Conduits+ [10] has its roots in object orientation, and SDL [14] has process oriented background. Other protocol frameworks are also studied by authors, but these frameworks or examples which utilize them are not presented here due to the lack of space.

The patterns presented in the section 5 "Patterns for Protocol System Architecture" are the *Protocol System pattern*, the *Protocol Entity pattern*, and the *Protocol Behavior pattern*. The Protocol System pattern and the Protocol Entity pattern model static parts of a protocol system, i.e. they are the components that a system is composed of, what is the structure of the components and how the system components are interconnected. The Protocol Behavior pattern models actual message exchange between different parts of a the system.

A simplified TCP/IP stack implementation is used as a running example in this paper. This protocol stack example contains only TCP and IP protocols, shows parts of their high level behavior and collaboration, but not any implementation details of them. TCP/IP is explained in many data communication textbooks, including [4]. Implementation details of TCP/IP are explained in [5].

The Unified Modeling Language (UML) is used in the class diagrams of this paper, and the UML pattern notation from [2] is used to describe roles of an individual pattern.

3 Communication Protocol Structure

Protocol systems offer a multitude of services on differing networks with a number of service options. In order to tackle this complexity, protocol systems are organized as series of subsystems, often called protocol layers or protocol entities. Conceptually different responsibilities are separated to different layers and implemented separately. As a result there are many benefits compared to the case when the entire protocol is implemented as one monolithic entity.

Each entity is designed to deal with a problem, or a set of problems, in the context of the entire protocol's functionality, using the services provided to it by the lower-level layers. The entities are interconnected through well-defined interfaces. Together they form a communications protocol system, often referred to as a protocol stack.

An entity has to be able to communicate with its peer and adjacent entities according to its specification. To fulfill these requirements an entity has to have interfaces to adjacent entities in the same system, an interface to its peer, internal data storage facilities for protocol information, and description of its behavior. In protocol engineering, an interface defines two sets of messages, the set of sent messages and the set of received messages. A entity interface corresponds to an OSI Service Access Point (SAP) [3].

Communication of an entity can be connectionless and/or connection-oriented. A connectionless communication is a simple Request-Response (or just Request) kind of message exchange.

A connection-oriented communication consists of connection establishment, message exchange, and finally disconnection phase. There can be multiple concurrent communications which can be in different phases.

Research on several existing protocol implementations revealed that protocol implementations contain the elements shown in Figure 1. The same high-level model can be found in several protocol frameworks, including SDL [14], CVOPS [12] and Conduits+ [10].

Figure 1 contains four protocol entities in two different protocol stacks. Protocol entities communicate with each other by sending messages. Protocol stacks are connected using a physical connection which represents the network. The entities in the same stack are connected to each other using message paths. Messages internal to the system are sent via these paths. Protocol entities are connected to their peer entities using virtual message paths. Messages external to the system are sent via these virtual paths. Peer communication is virtual since the messages sent to peer entities are actually sent using the interface provided by the lower protocol entity.

Figure 1 combines two different views that are presented separately in the examples in Figure 2 and Figure 3. Figure 2 presents an IP protocol peer centric view. The figure shows only those messages that are sent between TCP and IP protocol layers and between two peer IP layers. The layers below IP

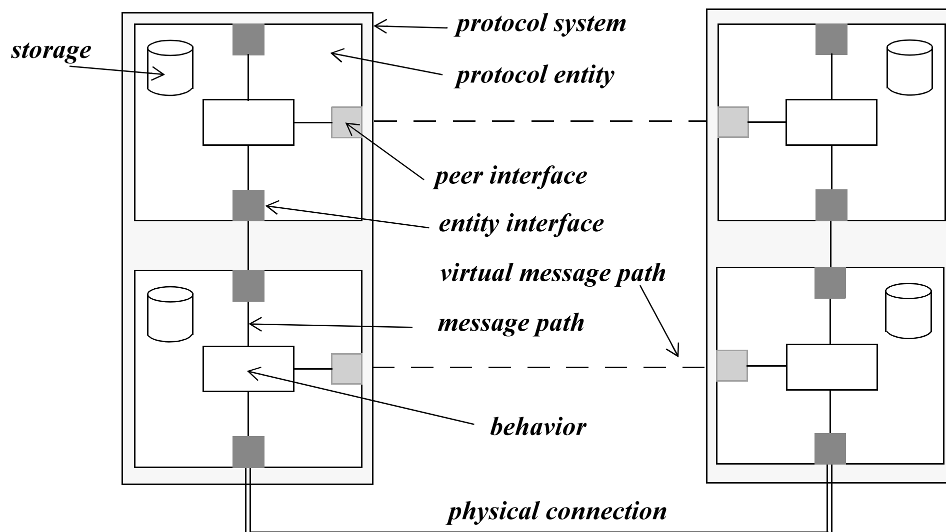

Fig. 1. Two Protocol Systems communicate with each other via physical connection

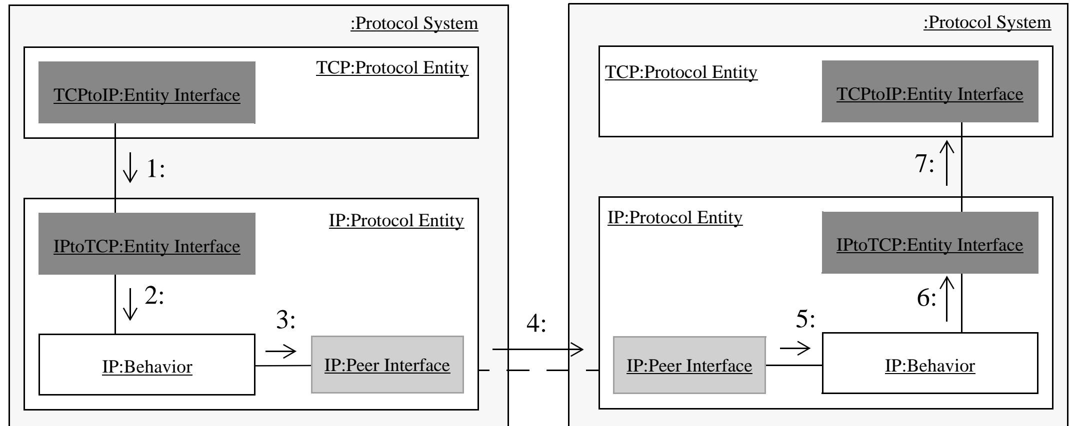

Fig. 2. The Protocol Entity IP send one message to its peer entity using a virtual message path.

are omitted. This view makes it possible to specify behavior of a protocol only in terms of IP messages.

Figure 3 presents entity interface centric view. Only real interfaces and message paths are shown. Peer communication is hidden inside protocol behavior.

Both views are very useful because they answer different questions. The first one answers the question “what is the correspondence between internal messages towards entity interface users (TCP in case of IP) and external messages towards peer protocol entities”. The second one answers the question “how protocol entities within a system communicate with each other”

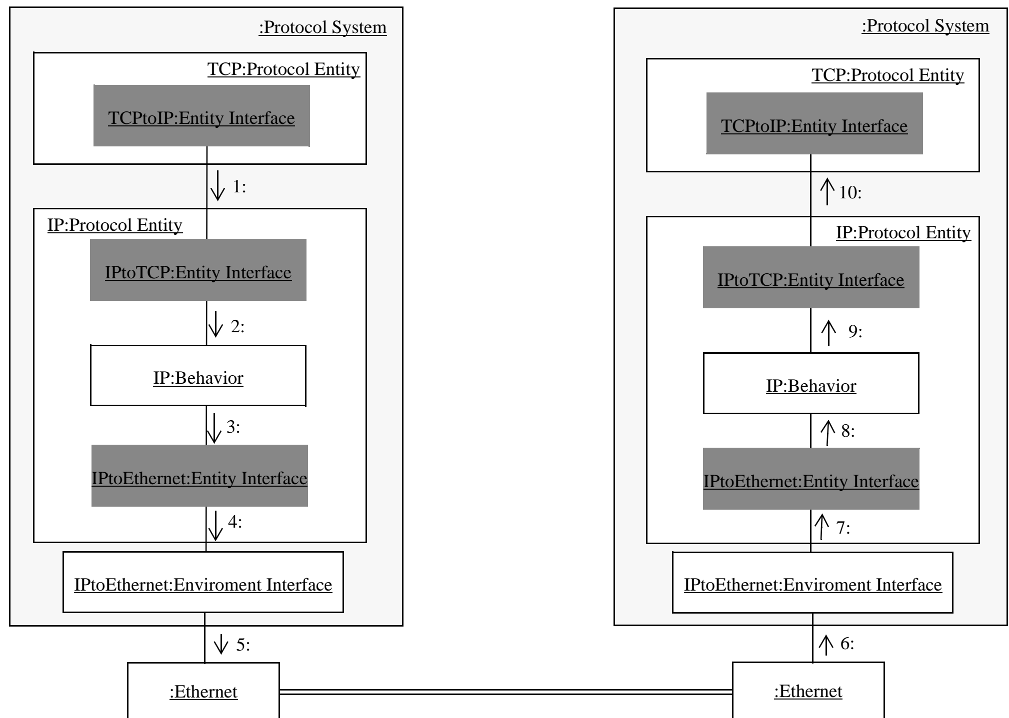

Fig. 3. Protocol Entity IP send one message to its peer entity using a real message path via an external object (Ethernet).

4 Protocol Implementation Frameworks

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 [9].

In a protocol implementation there are functionality that are always needed to be implemented. Implementation of statemachines of protocol entities, communication between protocol entities within the same system and between peer entities are common tasks that everyone working with protocols has encountered. Protocol implementation frameworks offer efficient and simple way to create and configure functionality of these common tasks. They also usually offer support for low level implementation issues like timers, memory management, and scheduling.

During the last decades many different frameworks and languages for protocol implementation have been developed. Next in this section two of them are described briefly. Conduits+ [10] has it roots in object orientation and it is implemented using design patterns from [7]. Specification and Description Language (SDL) [14] is a language that can be used when modeling communicating systems. SDL systems can be defined using either graphical or textual representation. It has process oriented background, but there are currently ongoing work to add object-oriented features to SDL.

4.1 Conduits+

The Conduits+ is a protocol implementation framework. Its ancestor, the Conduits framework, was designed by Jonathan Zweig in his Master's Thesis [11]. It was used to implement the TCP/IP stack. The framework was later improved by Hüni et al and it was then named Conduits+ [10]. In this section the graphical symbols and the roles of different conduits conform to the Conduits+ [10].

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.

There are four different conduits - Protocol, Mux, ConduitFactory and Adapter. The roles and responsibilities of the different conduits are as follows:

- A Protocol conduit is the heart of a protocol layer implementation. It encapsulates the behavior of a protocol, i.e. the reactions to incoming messages at a given time.
- A Mux conduit multiplexes messages from several conduits to a single one and vice versa. For example in case of a connection-oriented protocol a mux routes incoming messages to the correct connection handling Protocol conduit.

Fig. 4. Conduits+ class diagram

- A ConduitFactory creates new instances of conduits for example to handle a new communication session.
- An Adapter conduit is used as an interface to the world outside of a conduits graph, like some other software or hardware.

A conduit is a subclass of the Conduit base class and has capabilities to both interconnect with other conduits, and handle incoming messages. A conduit has two distinct connection points, Side A and Side B. Two conduits can be connected to each other by connecting their sides. Messages are sent and received via these connection points. Messages are carried by MessageTransporters, and they traverse the conduit graph and can trigger actions in the conduits. The class diagram of the Conduits+ is shown in Figure 4.

One example of a simple TCP/IP protocol implementation using the Conduits+ is shown in Figure 5. In this figure there are two Adapters to connect conduits graph to Ethernet and to SocketAPI (TCP/IP socket application programming interface), two Muxes to handle connection-oriented socket communication of the TCP, and single Protocol conduit to handle all connectionless communication of the IP protocol. In Conduits+ one can not separate protocol layers, or entities, by encapsulating selected conduits within other structures. In Figure 5 additional notation is used to guide a reader and make easier for him or her to see the protocol entity boundaries.

Behavior of a protocol entity is implemented in a Protocol conduit. Behavior 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 protocol messages. Events to a Protocol conduit are sent as Messengers which carried by MessageTransporters. In Figure 5 one Protocol conduit handles one socket in the TCP.

The role of a Mux is to route incoming messages to a proper receiver. When an incoming message requires a new connection then a Mux must request a ConduitFactory to create a new conduit to handle the incoming message. A Mux is used to handle multiple simultaneous sessions on a single protocol layer. In Figure 5 Muxes routes an incoming message to a proper Protocol conduit, i.e. TCPSocket, in the TCP.

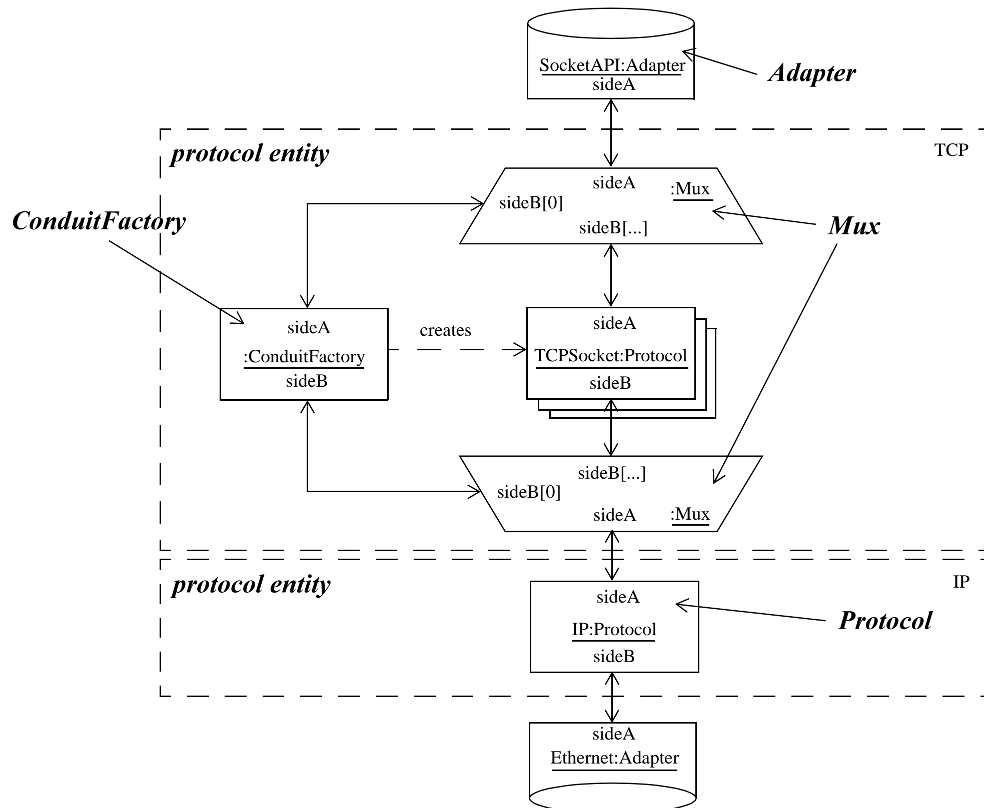

Fig. 5. Simplified TCP/IP protocol stack using the Conduits+ implementation framework

An Adapter is used as an interface to some other software or hardware. Thus the Adapters are the endpoints in the conduit graph.

The Conduits+ framework uses widely design patterns from [7]. An example of design patterns is the Visitor pattern [7] which is used to decouple Messengers from conduits. Visitors are responsible for the traversal on the conduit graph and the Messengers only trigger actions. Another example is the Protocol conduit which uses the modified State pattern [7]. For more details on the used design patterns please see the original Conduits+ paper [10]

4.2 Specification and Description Language (SDL)

Specification and Description Language (SDL) is a standard language for specification and description of communicating systems. It is currently developed by ITU-T and is defined in the Z.100 recommendation [14]. 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. There are currently ongoing task to develop SDL2000 which contains even more object-oriented features.

An SDL system 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 as an interchange format.

Currently SDL has a dual role as a specification language and also as an implementation language. In this paper we concentrate on the implementation language aspect. There are SDL tools that provide necessary features so that they can be used as protocol implementation frameworks.

An SDL system consists of blocks which can communicate with each other and with the environment surrounding the system by sending signals. In Figure 6 the system consist of two blocks TCP and IP. The blocks correspond to protocol layers. Inner structure of the TCP block is shown in Figure 7, and the IP block in Figure 8. Communication between the blocks is done via channels. Signal lists associated with the channels list the signals that are allowed to be sent to a given direction.

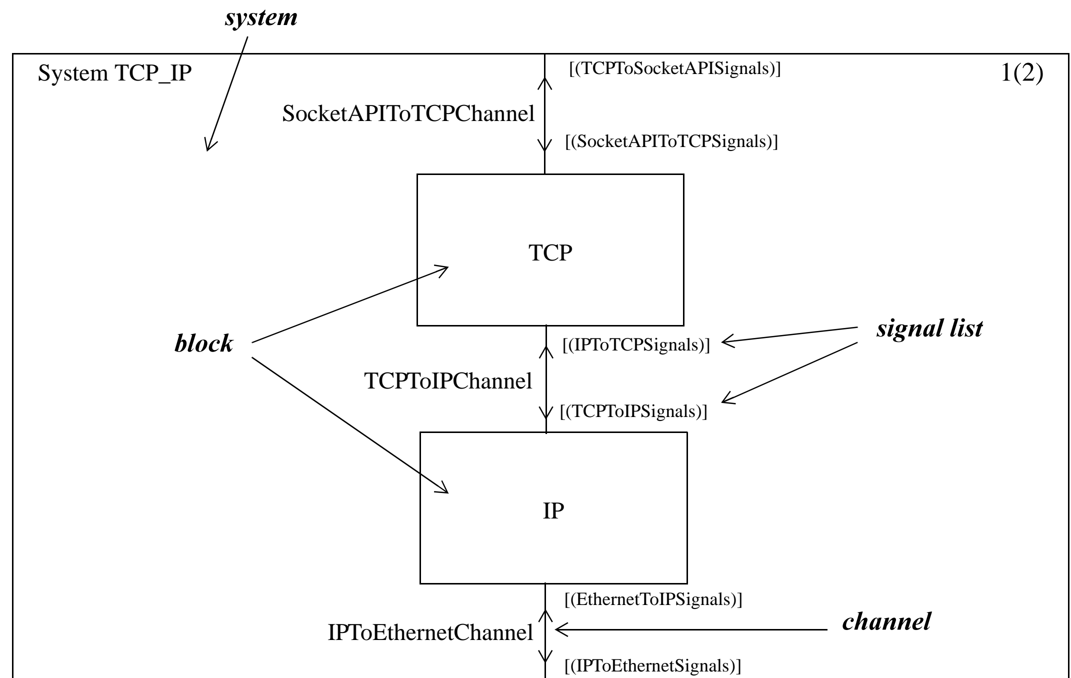

Fig. 6. TCP and IP layers in a protocol stack

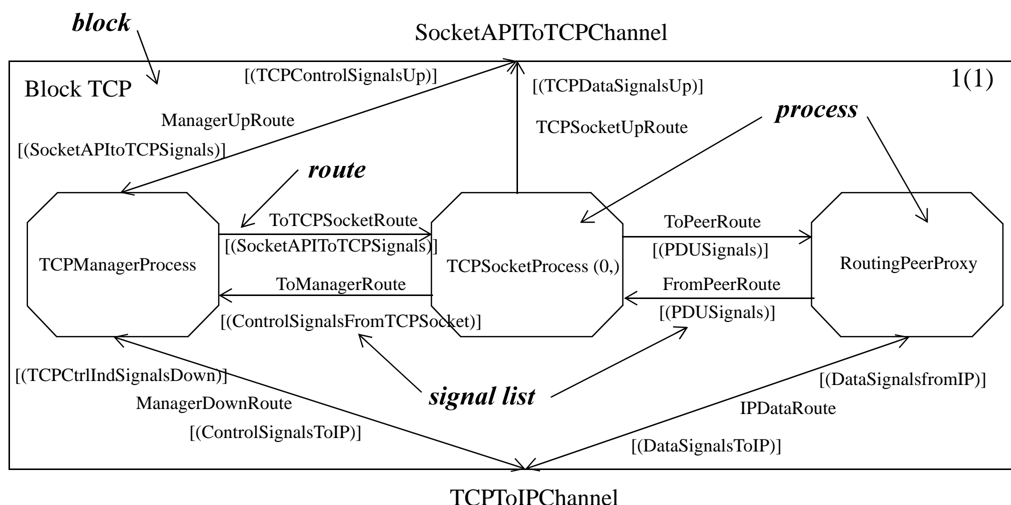

Fig. 7. Structure of the TCP

Figure 7 contains an example of SDL implementation of the TCP layer. It uses the SDL codec process idiom from [15]. There are three different roles and responsibilities:

- TCPManagerProcess handles control messages coming from the socket interface and IP interface. It also manages creation of new connections and routes incoming messages from SocketAPI to a proper TCPSocketProcess.
- TCPSocketProcess handles one communication session. There is one instance of TCPSocketProcess per one communication session.
- PeerProxy provides peer communication abstraction for TCPSocketProcess. It encodes outgoing external messages and packs them to messages going to the IP layer. It extracts peer messages from incoming IP layer messages, decodes them and routes them to a proper TCPSocketProcess.

Figure 8 contains definition of the IP layer. Because IP is connectionless there is no need for separate manager process. Also peer communication does not have an explicit representation, it is hidden inside IPProcess.

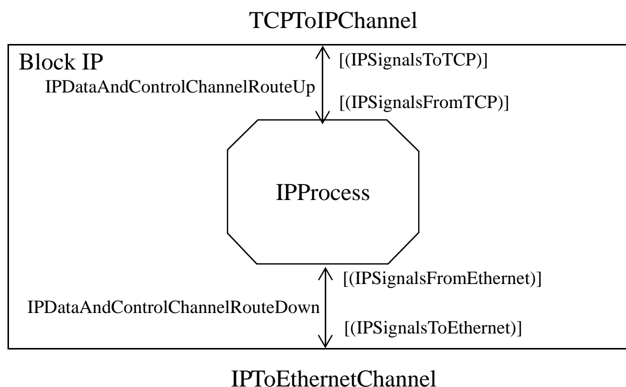

Fig. 8. Structure of the IP layer

5 Patterns for Protocol System Architecture

The common general parts and relations in different protocols can be identified and described as design patterns [7]. In this section three patterns for protocol system architecture are presented. These patterns are tightly related to each other, and are part of a pattern language for protocols.

The patterns presented in this paper are the *Protocol System* pattern, the *Protocol Entity* pattern, and the *Protocol Behavior* pattern. The Protocol System pattern and the Protocol Entity pattern model static parts of a protocol system, i.e. which are the components that a system is composed of, what is the structure of the components and how the system components are interconnected. The Protocol Behavior pattern models actual message exchange between different parts of the system. These three patterns can be considered as a an architectural patterns [8].

The design pattern format used here is a modification of several existing ones. The main modification is the order of the parts of pattern description: forces are presented after context and solution. The reason for this is simple. Patterns presented here have been used in communication system engineering for decades and their original inventors are unknown as original reasons to take them in use.

The Layers architecture pattern [8] is closely related to these patterns. However, in the Protocol System pattern, Protocol Entities, or layers, don't have to be adjacent. These patterns also describe more inner details than the Layers pattern.

Protocol System

Context

A communication system is composed of three parts: a user, a communication protocol system, and a communication media (i.e. environment or hardware). A communication system acts as a mediator between the user and the environment. The user is interested in very high-level aspects of communication, and the environment is offering only low-level communication services. If there are changes in system or environment, the user doesn't want to know it. He or she wants just to use his or her application above the communication system.

The communication system is typically specified as adjacent layers [3][8], or entities, with two types of interfaces: peer interface and entity interface. Some of these interfaces are standardized in some acceptance level: de-facto, de-jure etc. An interface between peer entities is always standardized whereas interface between entities in the same system is rarely standardized. However, there is often a recommended or informal entity interface.

Solution

The Protocol System pattern is the highest level pattern in the pattern language for protocol system. It encapsulates the whole protocol system, and forms the basis of other protocol patterns. It describes a protocol system structure in high level by specifying what are the components that a system is composed of, and how they are interconnected to each other. The Protocol System pattern models a protocol system by specifying:

- what are the components that a system is composed of,
- which are responsibilities of the components,
- how the components are interconnected, and
- how a system communicates with its environment.

The roles of Protocol System pattern are *Protocol System*, *Protocol Entity*, *Entity Interface*, and *Environment Interface*, as shown in Figure 9.

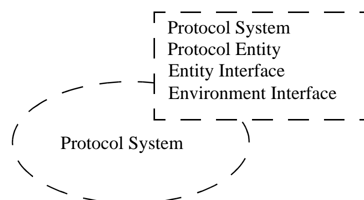

Fig. 9. The roles of Protocol System Pattern

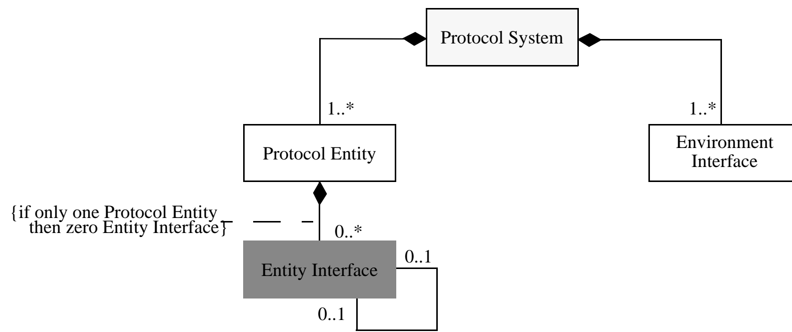

Fig. 10. Protocol System Pattern Structure

These roles, or components, that form a protocol system have relations and multiplicities as shown in Figure 10. A Protocol System encapsulates other components as a single system. A Protocol Entity represents a protocol layer or sublayer. It contains Entity Interfaces. A Protocol Entity communicates with other Protocol Entities in the same system by exchanging messages through Entity Interfaces. The Entity Interface defines the allowed set of incoming and outgoing messages. It is these bindings between Entity Interfaces that specify how system components are interconnected.

An Environment Interface 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.

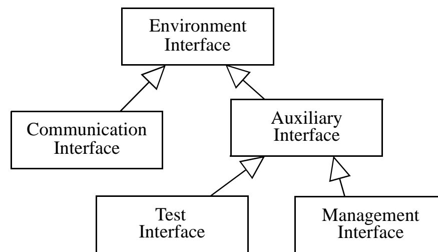

Fig. 11. Enviroment Interface class diagram

An Environment Interface may have different roles. The subclasses, Communication Interface and Auxiliary Interface, represent those roles. The Environment Interface subclasses are presented in Figure 11.

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 11. An Auxiliary Interface subclasses are as Test and Management Interfaces.

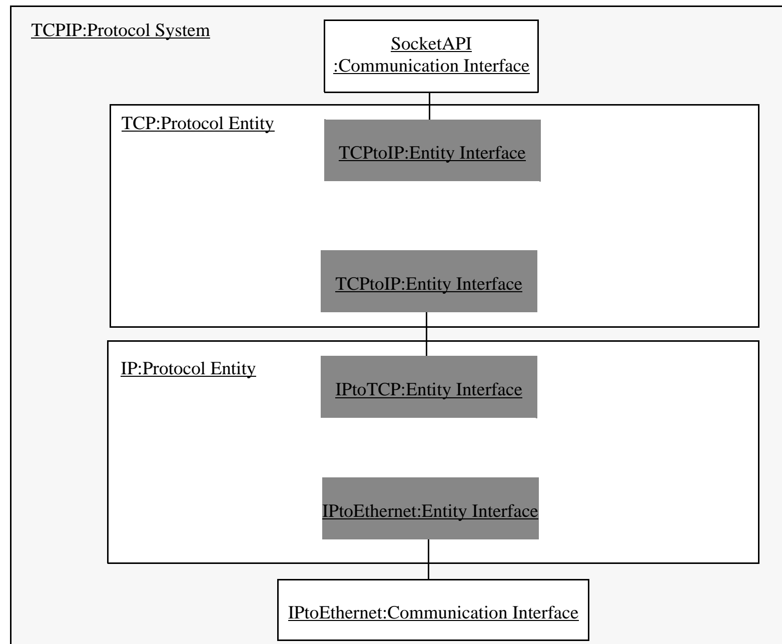

Fig. 12. Simplified TCP/IP protocol Stack implemented using the Protocol System pattern - an object diagram.

One example of Protocol System pattern use is shown as a Object Diagram in Figure 12 on page 11. In this figure a simplified TCP/IP protocol stack is presented. Between TCP and IP layers there are entity interfaces for them, and Environment Interfaces to the SocketAPI and the Ethernet.

Forces

Patterns presented here have been in practical use in communication system engineering for decades and their original inventors are unknown as original forces which lead to these.

Communication systems are specified using adjacent layers using OSI reference model [3] or other layer models. Layers are build on top of each other, higher layer uses interface, or service access point (SAP), provided by lower layer [3]. The use of interface hides lower layer from layers above it. This modularity and encapsulation of entities, which have been in use years before advent of object-oriented programming, serve three related goals:

- organization of standardization effort,
- modular, component based, implementation, testing and maintenance process, and
- transparent and reliable system for user.

These three cases are main sources of forces of this pattern. Effort of standardization organization brings following forces:

1. Some parts of a protocol system are standardized and some are not. An interface between peers is always standardized. Standardized interface is often called a normative interface. Interface between entities in the same system is not always standardized, but there is often a recommended or informal interface. This recommended interface is often called as a non-normative interface.
2. An interface in standardization terms means detailed specification of incoming and outgoing messages. Interface in object-orientation means only incoming messages (i.e. method calls).

Realization of modular implementation, testing, and maintenance process brings following forces:

1. Time to market is short. When a standard (in some acceptance level; de-facto/de-jure etc.) is published there is relatively short time to make a product. In normal case companies participate standardization process to get early information and to have effect on the standard.
2. Testing of a communication system is complicated. In normal case there are more possible combinations of message exchange than it is feasible to test.
3. Implementation of complicated communication software is slow and error-prone.
4. Parts of a protocol system are legacy and they are implemented using some old technology. Reuse of old existing parts is used to improve implementation time and save old investments.
5. Maintenance and updating of a protocol system must be easy and low cost.

6. The implementation of a system is quicker and less error-prone if used implementation language or platform support repetitive tasks of protocol system implementation.

Transparent and reliable communication system for user brings following forces:

1. There is a big gap between a user of a protocol system and a environment, or hardware, which a communication system should fill. If there are changes in environment, the user doesn’t want to know it. He or she wants just to use his or her application above the communication system.
2. A protocol system has to be robust. A protocol system has to work long time without any defect. It should also recover almost all possible defects automatically and notify this to maintenance. A protocol system should never loose any important data. One example of important data is accounting information for an operator.
3. The communication media (i.e. environment) is not always reliable.
4. A protocol implementation has to be as efficient as possible. Especially video conference and other real-time multimedia applications are very sensitive for even small delays. However, in mobile computing communication is usually bounded by wireless network, not protocol itself.

The Protocol System pattern is used to balance these forces together with other patterns in this language.

Examples

Following section contains implementation examples of the Protocol System pattern using Conduits+ [10] and SDL [14]. These are described briefly with simplified TCP/IP protocol example from the section 4 "Protocol Implementation Frameworks". This section uses figures from that section.

Conduits+

There is not one distinct Protocol System component in the Conduits+ framework. Parts of system, the conduits, are bind together when a system is initialized. The Figure 5 on page 6 contains a simplified TCP/IP protocol System example. Even if there is a conceptual separation between TCP and IP layer components (shown as a dashed box) from Conduits+ point of view there is no difference between the conduits in the TCP and IP layers. Also the Protocol Entity component is purely conceptual. A Protocol Entity consists of at least one Protocol conduit and any number of Muxes and ConduitFactories. In the Figure 5 the conduits inside the TCP box belong to one Protocol Entity and the conduit inside the IP box belong to another.

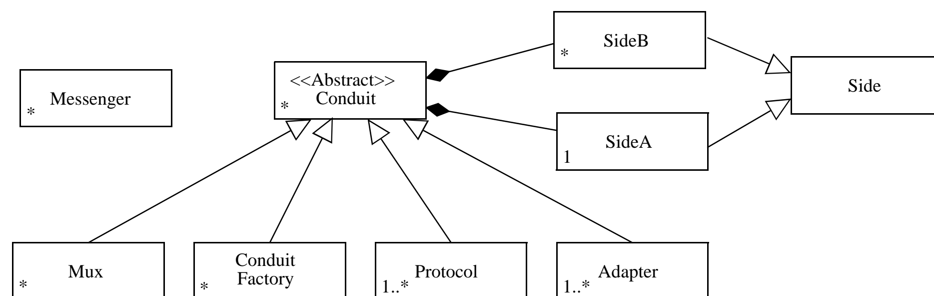

Fig. 13. Protocol System Pattern related Conduits+ classes

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 one part the Entity Interfaces in the Conduits+ framework. However, a set of Messengers is used to define all possible incoming and outgoing events of a Protocol Entity, and thus are other part of Entity interfaces.

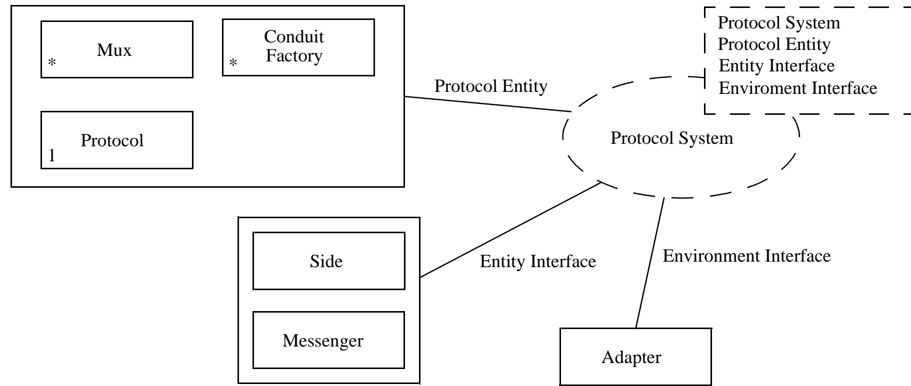

Fig. 14. Conduits+ Protocol System as pattern

In Figure 14 it is shown how components of the Protocol System pattern are mapped in Conduits+. This figure can be used as a legend to map the roles of this pattern to the TCP/IP implementation example shown in Figure 5 on page 6. However, Messengers are not shown in Figure 5.

SDL

SDL provides mechanisms for coping with large number of communicating processes. The processes can be organized in a hierarchy of blocks hiding the fine details when inspected from a higher abstraction level.

In SDL a system component has the Protocol System role in this pattern, and a block represents a Protocol Entity as shown in Figure 16. The different blocks 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 these two components together define the Entity Interface between two entities. They also define Environment interface out of the system. These components and their relations are shown in Figure 15 as a class diagram.

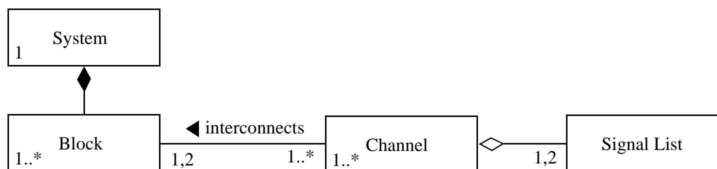

Fig. 15. Class diagram of the SDL Protocol System pattern

A concrete example of these and use of SDL to present high-level protocol definition is shown in Figure 6. The TCP block and the IP block and the channels between them and from/to environment form a Protocol System. Signal lists (e.g. IPToTCPSignals and TCPToIPSignals) associated with channels specify what are the messages that can be transmitted for a given direction in a channel.

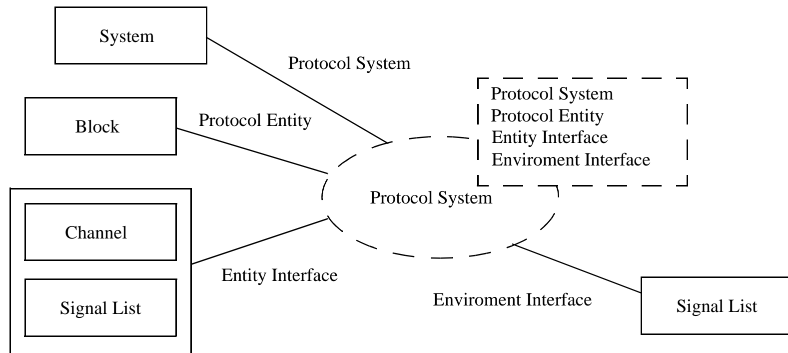

Fig. 16. SDL Protocol System pattern

In Figure 16 it is shown how components of the Protocol System pattern are mapped in SDL. This figure can be used as a legend to map the roles of this pattern to the simplified TCP/IP implementation example shown in Figure 6 on page 7.

Consequences

The Protocol System pattern has several advantages, but it also contains a few disadvantages. These are quite similar as in the Layers architecture pattern [8].

Advantages

- 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 multiple context.
- Support for standardization. Clear-defined and commonly-accepted levels of abstraction facilitates 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 great effort.

Disadvantages

- 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 efficiency. An architecture, which is implemented using the Protocol System pattern, is usually less efficient than a monolithic implementation. This is result from the large number of operations and transformations the message flow experiences when it passes thought the entities.
- Unnecessary work. Some services performed by previous entities may not needed by adjacent entities. This has of course a negative impact on performance.

See also

The Layers architecture pattern [8] is closely related to this pattern. However, in the Protocol System pattern, Protocol Entities don't have to be adjacent. These patterns also describe more inner details than the Layers pattern.

Inspirers

The following artifacts and people have inspired the authors:

- Layers architecture pattern [8],
- lectures of professor Olli Martikainen from Helsinki University of Technology,
- work with many protocol implementation frameworks, including Conduits+ [10], SDL [14], and CVOPS [12],
- many protocol implementations, including TCP/IP and GSM,
- numerous data communication text books, especially [3], and
- our shepherd, Michael Stal, and participants of PLoP2000 "network of learning" -workshop.

Authors

Juha Pärssinen, Markku Turunen

Protocol Entity

Context

Communication systems are typically organized as a series of layers or entities. An entity is capable of sending or receiving information, and it has an inner state during communication. An entity has a virtual message path to its peer entity (see Figure 2 on page 4), and real message paths to its adjacent entity (see Figure 3 on page 4) or environment as shown in Figure 1.

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].

Solution

The functionality of an entity is divided to the roles shown in Figure 17. The Protocol Entity pattern contains the *Protocol Entity*, the *Protocol Behavior* and the *Storage*, and it uses and provides the *Entity Interfaces* and a *Peer Interfaces*.

A Protocol Entity, as shown in Figure 17, contains a Protocol Entity, a Protocol Behavior and a Storage, and it uses and provides Entity Interfaces and Peer Interfaces. One Protocol Entity can be

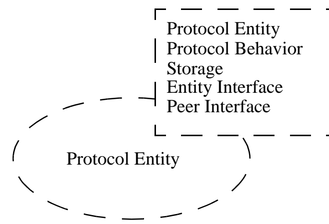

Fig. 17. Protocol Entity Pattern

used as a layer in the Layers architecture pattern [8]. A class structure of the Protocol Entity pattern is presented in Figure 18.

- The *Protocol Entity* represents one protocol layer.
- A *Protocol Behavior* handles protocol functionality. The Protocol Behavior is explained in detail in the section “Protocol Behavior” on page 18.
- 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 communication session 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.

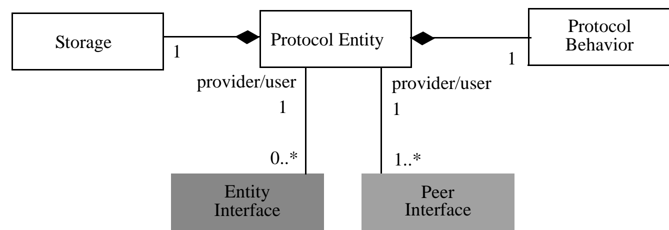

Fig. 18. Protocol Entity Pattern Class Structure

Forces

It is complex to design, implement, and test a Protocol Entity. The Protocol Entity 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, and
- communication with entities in peer systems.

Forces listed in Solution section of the Protocol System Pattern affect in this pattern.

Examples

Following section contains implementation examples of Protocol System pattern using Conduits+ [10] and SDL [14]. These are described briefly with the simplified TCP/IP protocol example from the section 4 "Protocol Implementation Frameworks". This section uses figures from that section.

Conduits+

A Protocol Entity itself is purely conceptual in Conduits+. Its structure 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. Every conduit can contain ordinary variables which can be used to store protocol specific information.

The sides of conduits are used to connect conduits to each other, and are one part the Entity Interfaces. Other part of Entity Interface is set of Messengers which are used to define all possible incoming and outgoing events of a Protocol Entity.

A concrete example of these and use of Conduits+ to present a protocol entity is shown in Figure 5 on page 6. The conduits inside the TCP box belong to one Protocol Entity. Muxes are used to route

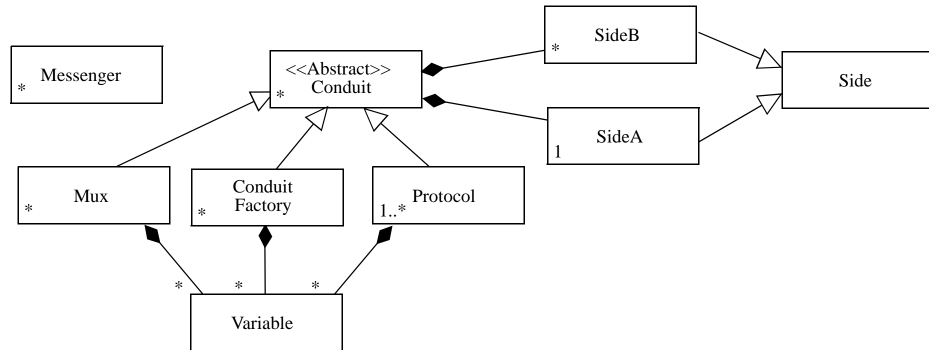

Fig. 19. Protocol Entity Pattern related Conduits+ classes

messages. ConduitsFactory is used to create new TCPSockets which are Protocol conduits. These have functionality to handle peer communication and they also contain all data of a single connection.

In Figure 20 it is shown how components of the Protocol Entity pattern are mapped in Conduits+. This figure can be used as a legend to map the roles of this pattern roles to the TCP/IP implementation example shown in Figure 5. However, Messengers are not shown in that figure.

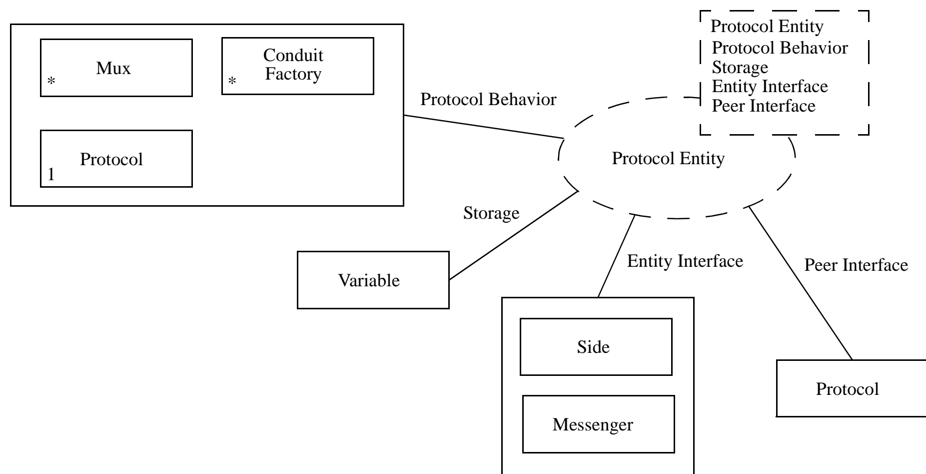

Fig. 20. Conduits Protocol Entity as pattern

SDL

In SDL a block can contain several processes or process sets. A block represents the Protocol Entity role as shown in Figure 22. 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.

Processes inside a block communicate with the block environment, i.e. the system outside them, also using 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. For example in Figure 7 on page 8 there are signal routes between a process and a block environment (e.g. ManagerUpRoute connected to the channel SocketAPIToTCPChannel) and between processes inside a block (e.g., ToTCPSocketRoute).

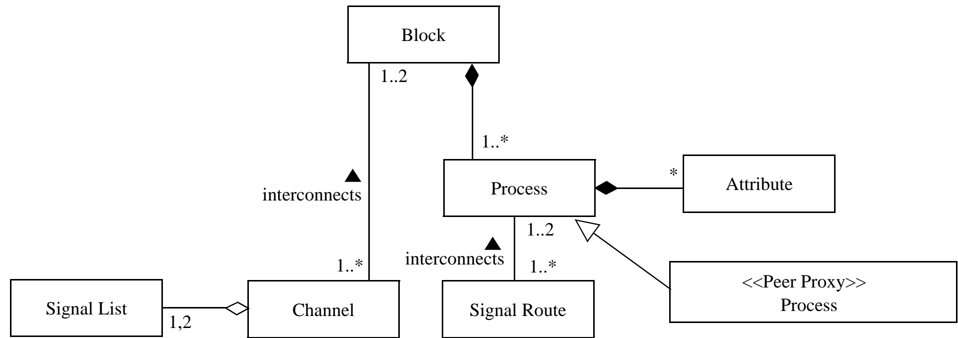

Fig. 21. Conceptual SDL Protocol Entity diagram

Signal lists can be used to define the signals that can be transmitted through a channel, these two component together define the Entity Interface between two entities. A channel name is shown in the border of a block.

The Peer Interface functionality is implemented in processes, and this forms one part of Peer Interface. The processes or process sets are connected with each other using signal routes. The signals carrying information are conveyed by these routes from a sender process to a receiver. The signal route forms another part of the Peer interface. For example in Figure 7 the routes ToPeer and FromPeer and the signals associated with them form the Peer Interface. The RoutingPeerProxy process acts as a proxy for the peer protocol entity. The process maps outgoing external peer messages to internal messages going to a lower layer and extracts incoming peer messages from lower layer messages. Use of a PeerProxy process (a.k.a a Codec process) for handling of peer messages is an SDL idiom [15].

Processes may contain attributes which are used to store information.

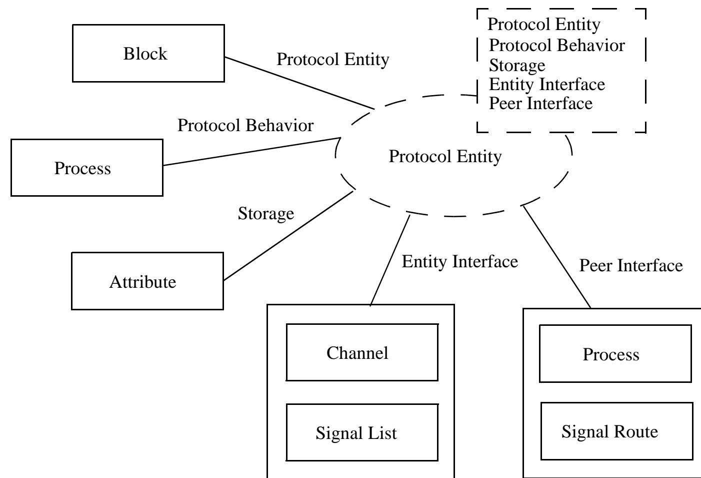

Fig. 22. SDL Protocol Entity pattern

In Figure 22 it is shown how components of the Protocol Entity pattern are mapped in SDL. This figure can be used as a legend to map the roles of this pattern to the TCP/IP implementation example shown in Figure 7 on page 8 and Figure 8 on page 8. Differences between these figures are related to the behavior of protocol in concern are explained in detail in the section “Protocol Behavior” on page 18.

Consequences

The Protocol Entity pattern has several advantages, but it also contains few disadvantages. These are quite similar as those of one layer in the Layers architecture pattern [8], see section “Protocol System” on page 9 for details.

See also

The Protocol Entity pattern uses the Protocol Behavior Pattern described on section “Protocol Entity” on page 14. The Protocol Entity pattern can be used as a compositional part of the Protocol System Pattern page 9. It can behave as a layer in the Layers architecture pattern [13].

Inspirers The following artifacts and people have inspired the authors:

- Layers architecture pattern [8],
- lectures of professor Olli Martikainen from Helsinki University of Technology,
- work with many protocol implementation frameworks, including Conduits+ [10], SDL [14], and CVOPS [12],
- Jim Coplien and participants of Tools Europe 2000 Pattern Workshop in beautiful Le Mont St. Michel.
- many protocol implementations, including TCP/IP and GSM,
- numerous data communication text books, especially [3], and
- our shepherd, Michael Stal, and participants of PLoP2000 “network of learning” -workshop.

Authors Juha Pärssinen, Markku Turunen

Protocol Behavior

Context In complex communication protocol system implementation the Protocol Behavior pattern can be used to implement entity functionality in general. Communication can be divided into two main types:

- connectionless, and
- connection-oriented communication.

Solution A specific part is dedicated to implement protocol behavior and communication between peer entities. The Protocol Behavior pattern is used to implement needed protocol functions for the system in concern. It encapsulates intelligence of protocol and contains roles which can be used to compose any kind of behavior of the protocol in concern. The Protocol Behavior patterns includes roles (shown in the Figure 12): *Protocol Behavior*, *Router*, *Communication Manager*, *Communication Session*, *Storage*, *Entity Interface*, and *Peer Interface*.

These roles and their use to compose connectionless and connection-oriented behavior are explained in detail in this section. Some of these roles are described in detail in section “Protocol Entity” on page 14.

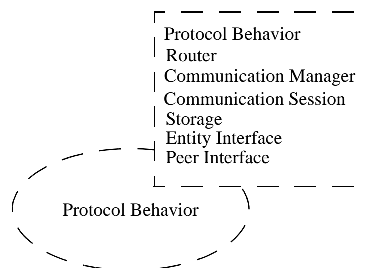

Fig. 23. Protocol Behavior Pattern

The *Protocol Behavior* contains the active parts of a protocol entity. It contains zero or more Routers, zero or one Communication Manager, and zero or more Communication Sessions as shown in Figure 24.

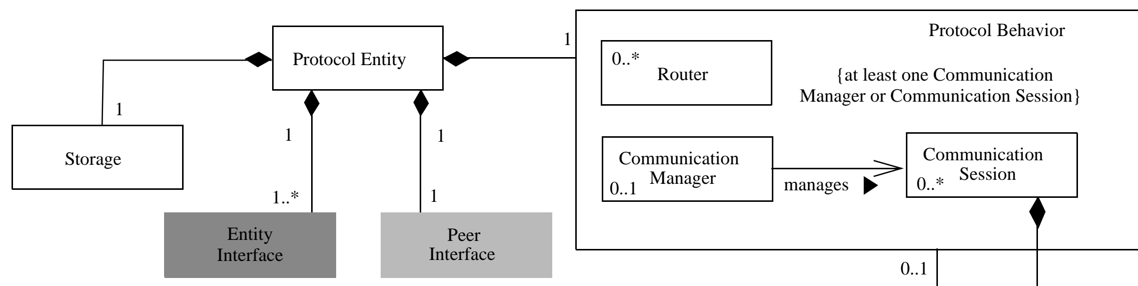

Fig. 24. Protocol Behavior Pattern Structure

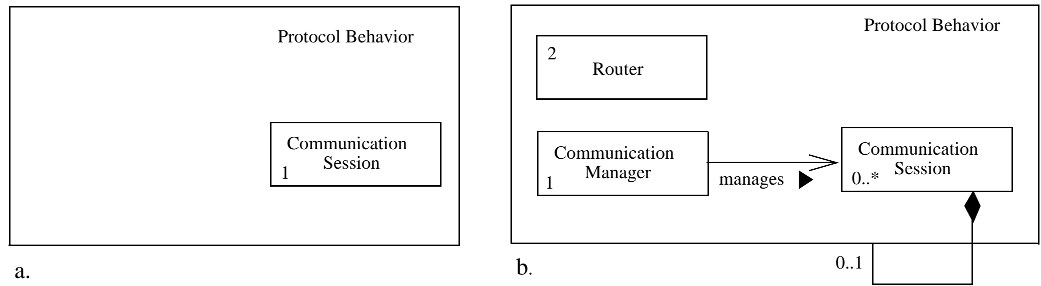

Fig. 25. Protocol Behavior Pattern Structure as an object diagram a. connectionless and b. connection-oriented nd connectionless in some cases

A *Router* is needed if there can be multiple receiving Communication Sessions for messages coming from a single entity interface. A Router routes incoming messages to correct receiver i.e. Communication Manager or one of Communication Sessions. In a case of connection-oriented protocol a *Communication Manager* creates, controls, and closes sessions as needed. An example of Router and Communication Manager at work is shown in Figure 27. The Router routes message first to TCPManager because there isn't TCPSocket for this connection yet available. After TCPSocket is created by TCPManager message is routed to it.

A *Communication Session* handles communication between two communicating peers. It uses Peer Entity to send and receive messages as shown in Figure 27. The Protocol Entity and other parts of it are explained in the page 14.

The Protocol Behavior pattern can be divided into two main types: connectionless and connection-oriented. A connectionless Protocol Behavior contains in a typical case one Communication Session which handles all communication as shown in Figure 25a. Examples of connectionless protocols are shown in Figure 5 on page 6 and Figure 8 on page 8 which presents IP protocol implementations using Conduits+ and SDL.

However, in some cases Communication Manager can also be used in connectionless protocols. A Communication Session is created to handle a single request. This way a Protocol Entity is capable to serve multiple requests at same time even if time required to fulfill one request is moderately long e.g. due to a database access.

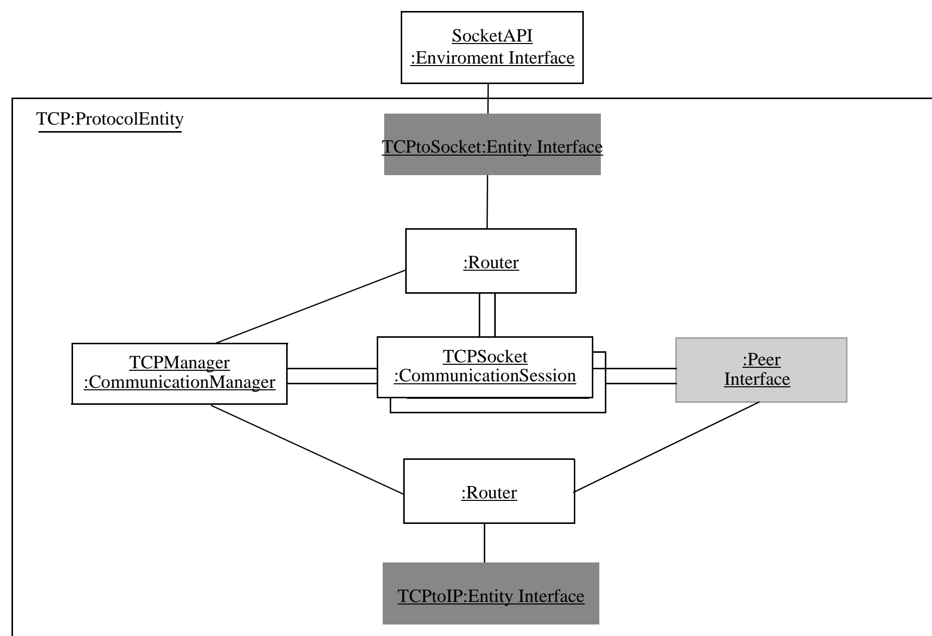

Fig. 26. Protocol Behavior Pattern Structure - object diagram

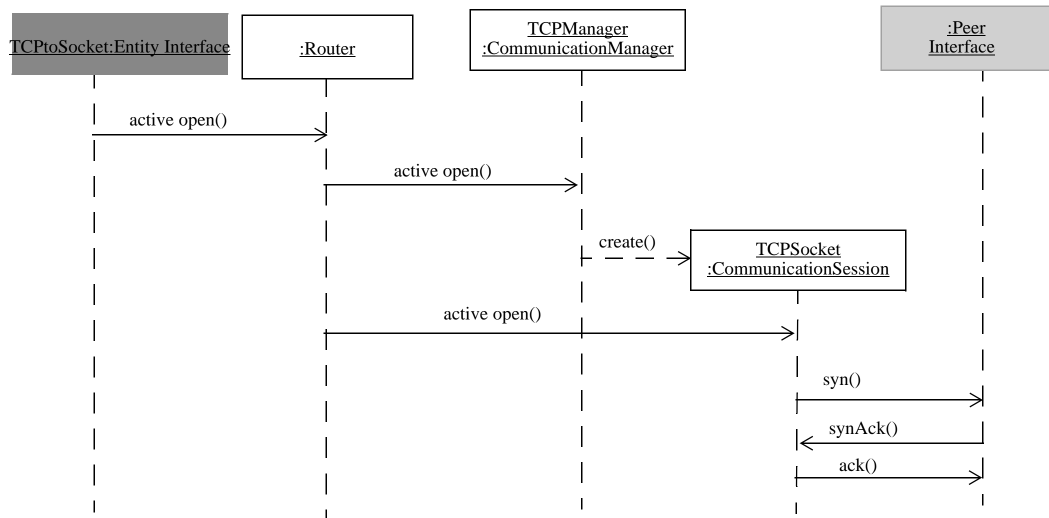

Fig. 27. Sequence diagram of TCP protocol connection establishment - client side

The connection oriented Protocol Behavior contains two Routers, one Communication Manager, and zero or more Communication Sessions as shown in Figure 25b.

One example of the use of the Protocol Behavior pattern is shown in Figure 26 as an object diagram which presents a snap-shot of a simplified TCP protocol behavior. In this diagram one connection-oriented TCP protocol layer is presented. In the current situation there are two concurrent communication sessions, TCPSockets. The creation of new TCPSocket is shown in Figure 27.

Forces

A communication entity has to manage one or more concurrent communication sessions which have their own states. Communication can be connectionless and/or connection-oriented. 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 multiple concurrent communications which can be in different phases. A special case of connection-oriented behavior is a case when one connection has sub-connections.

A protocol functionality is typically specified in standards as a finite state machine, and a list of allowed incoming and outgoing events.

Forces listed in Solution section of the Protocol System Pattern affect also in this pattern.

Examples

Following section contains implementation examples of Protocol System pattern using Conduits+ [10] and SDL [14]. These are described briefly with simplified TCP/IP protocol example from the section 4 "Protocol Implementation Frameworks". This section uses figures from that section.

Conduits+

The Connectionless Protocol Behavior is modeled by a single Protocol conduit. There is no need for a ConduitFactory on this kind of layer. A example of this is shown in the implementation of IP protocol in Figure 5 on page 6. However, if the service of this layer is supposed to be available for multiple parallel upper level protocols, an additional Mux is needed between the upper protocols and the Protocol conduit of this layer.

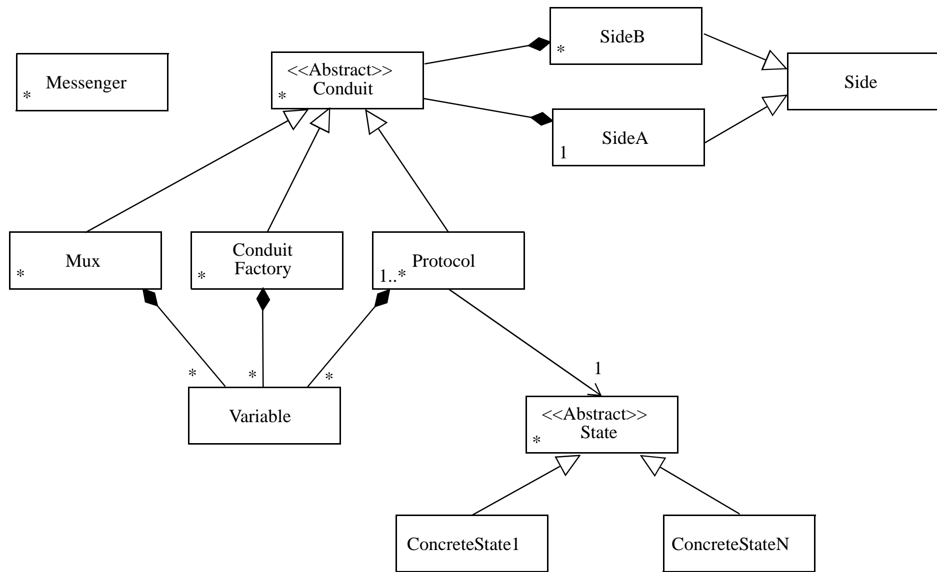

Fig. 28. Conceptual Conduits Protocol Behavior diagram

A connection-oriented protocol behavior requires a combination of Protocols, Muxes and ConduitFactories. A Protocol represents a single Connection and a Mux routes messages to and from several connections. A ConduitFactory creates new Protocol conduits and attaches them to the Mux. A example of this as the simplified TCP protocol implementation is shown in Figure 5.

The Protocol finite state machine 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. This conduit uses modified State pattern [7].

In Figure 29 it is shown how components of the Protocol Behavior pattern are mapped in Conduits+. This figure can be used as a legend to map the roles of this pattern roles to the TCP/IP implementation example shown in Figure 5. However, Messengers are not shown in that figure.

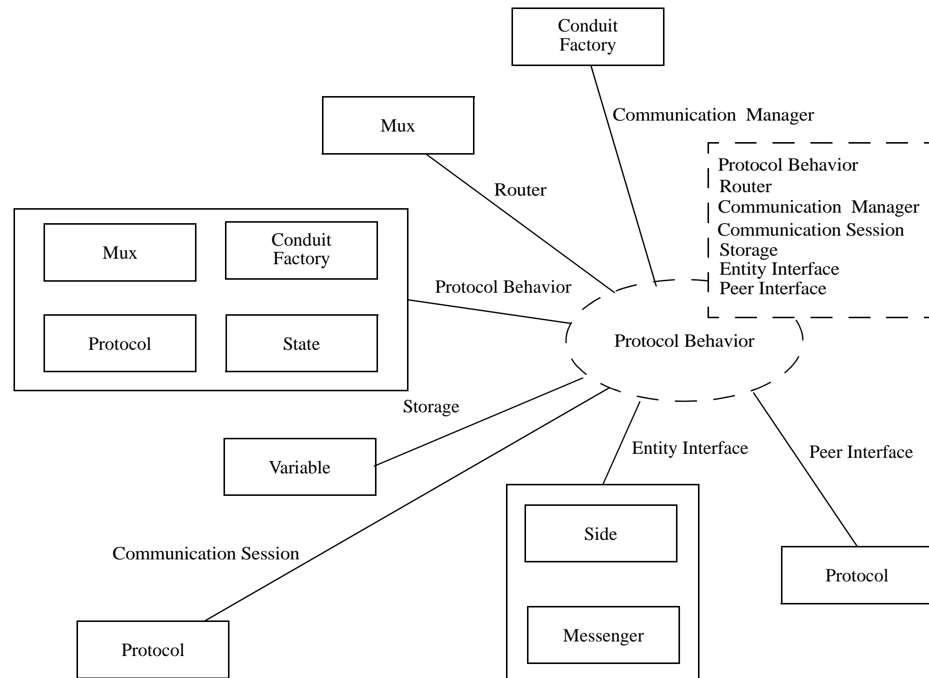

Fig. 29. Conduits Protocol Behavior as pattern

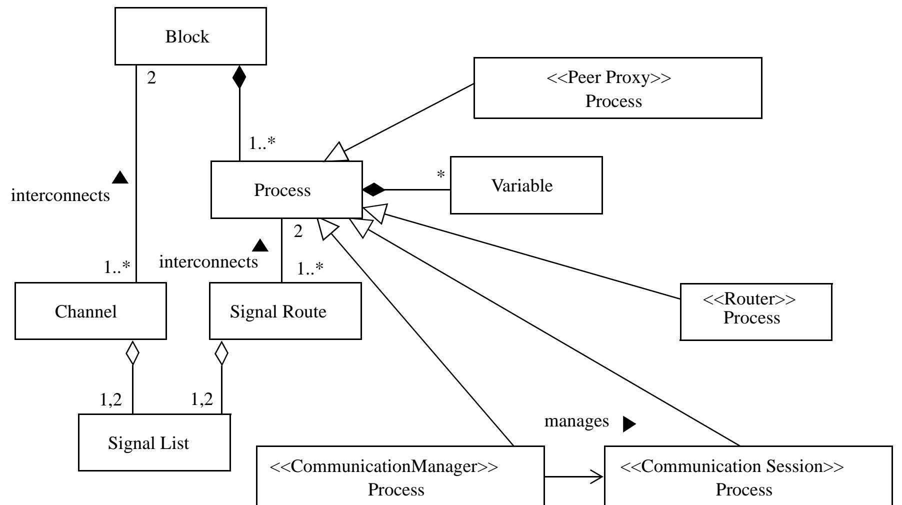

Fig. 30. Conceptual SDL protocol behavior diagram

SDL

In case of a connectionless protocol a single process can be used as a Communication Session. An example of this is shown in the implementation of the IP protocol in Figure 8 on page 8. IPProcess contains all the functionality of the IP layer. In a connection-oriented protocol the Router and Connection Manager roles can be implemented as own SDL process or they can be combined into one process. For example in Figure 7 on page 8 the process TCPManagerProcess acts both as a Communication Manager and as a Router for messages coming from a socket. Dynamically created and destroyed processes act as Communication Sessions.

In Figure 31 it is shown how components of the Protocol Behavior pattern are mapped in SDL. This figure can be used as a legend to map the roles of this pattern to the TCP/IP implementation example shown in Figure 7 and Figure 8.

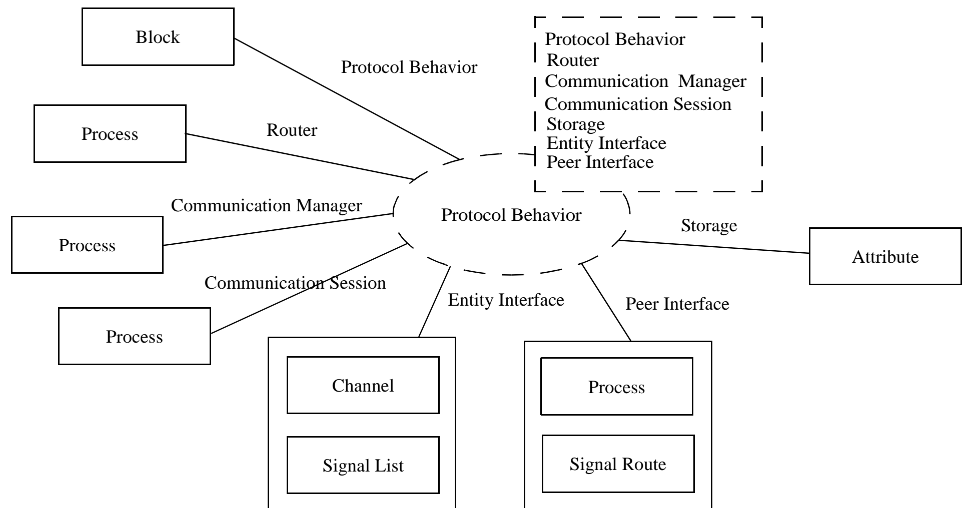

Fig. 31. SDL protocol behavior as pattern

Consequences

The use of the Protocol Behavior pattern produces a system which is clean and easy to maintain. However, it is usually less efficient than a simpler implementation. This is result from the large number of operations when a message passes through the system.

In case of a connectionless protocol there can be multiple serving Communication Sessions. The same idea can be used also in case of a connection-oriented protocol. A number of Communication Sessions can be instantiated and stored in a pool when a system is started. A Session can be taken into use when demanded and then returned to the pool when it is not needed any more. Using this approach it is fast to created a new connection, but it uses more resources because all Communication Sessions must be allocated at once, and static allocation limits number of possible connections.

See also

The Protocol Behavior pattern can be used to implement functionality of the Protocol Entity Pattern on page 14. The Finite State Machine [7][16][17] can be used to implement Communication Manager and Communication Sessions. Master-slave pattern is related to this pattern [8].

Inspirers

The following artifacts and people have inspired the authors:

- lectures of professor Olli Martikainen from Helsinki University of Technology,
- work with many protocol implementation frameworks, including Conduits+ [10], SDL [14], and CVOPS [12],
- Jukka Heinonen from Tellabs Oy and his work with WAP 1.0 Wireless Session Protocol (WSP),
- Master-slave pattern [8], and
- our shepherd, Michael Stal, and participants of PLoP2000 “network of learning” -workshop.

Authors

Juha Pärssinen, Markku Turunen

6 Example resolved

In Figure 32 there is a simplified TCP/IP stack which uses three patterns described in this paper. In TCP protocol Entity, which is connection-oriented protocol, there is one dedicated Communication Session for each socket. TCPManager manages TCpSocket, and Routers routes incoming messages to right receiver. In IP, which is connectionless protocol, on Communication Session handles all communication. Between TCP and IP layers there are well-defined interface. This interface is split two Entity interfaces: TCPtoIP and IPtoTCP. In real-life TCP/IP implementation there are not clear interface between TCP and IP, the implementation of these protocols is tightly coupled [5].

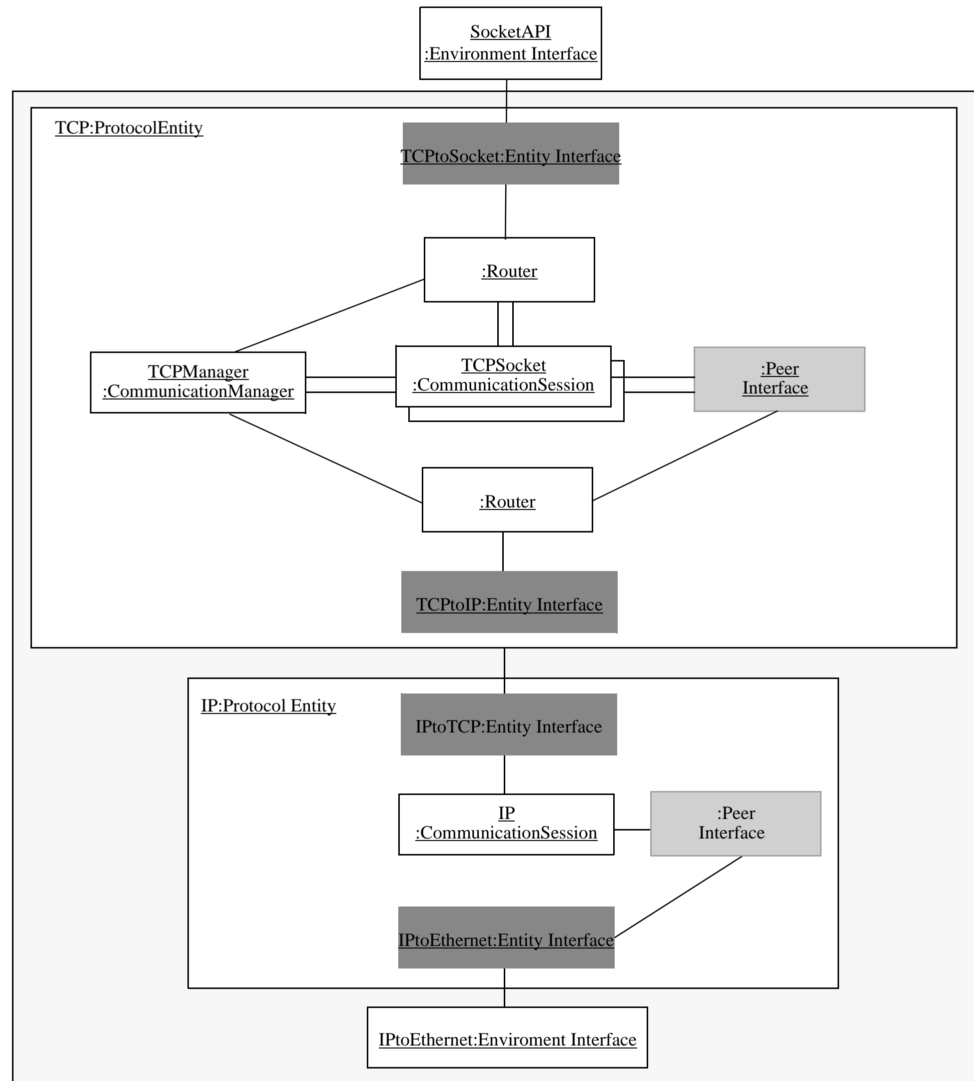

Fig. 32. Simplified TCP/IP protocol Stack implemented using the Protocol System pattern, Protocol Entity, and Protocol Behavior pattern - an object diagram.

7 Conclusions

This paper presented the patterns for a pattern language for protocols. The presented patterns are not the only existing ones. Authors have also found some other design patterns including the Message-Handler pattern which is used to encode/decode protocol messages, and the Manager-Session pattern which is used in the Protocol Behavior Pattern. The Manager-Session is a variant of the Master-Slave pattern [8]. There are also some implementation idioms for different protocol frameworks, including SDL PeerProxy process (a.k.a a Codec process) [15].

It was shown that the patterns can be found in protocol implementations implemented using different frameworks. It was also shown that there are protocol frameworks which on the surface look very different, and they have even implemented using different paradigms. However, they have and support the same concepts, or pattern roles, either explicitly or implicitly. A framework may not directly support all the concepts. Therefore framework specific idioms and workarounds have been invented in order to accommodate to the presented pattern system of protocols.

8 Acknowledgements

We would like to thank Ari Ahtiainen, Nokia Research Center, and Timo Kyntäjä, VTT Information Technology, for their support and comments, and professor Olli Martikainen from Helsinki University of Technology from his many lectures of telecommunication. We would also like to thank Jukka Heinonen, Tellabs oy, Niklas von Knorring, Helsinki University of Technology, Pekka Jäppinen, Lappeenranta University of Technology, and Arto Kvist, Necsom Ltd. for their numerous comments, critiques and improvements to these patterns.

We also like to thank our PLoP2000 shepherd, Michael Stal, and participants of PLoP2000 “network of learning” -workshop their numerous comments and improvements. Especially we like to thank Brian Marick of his idea, presented in that workshop, to describe these patterns in context-solution-forces order.

9 References

[1] J. Pärssinen, N. von Knorring, J. Heinonen, M. Turunen, *UML for Protocol Engineering - Extensions and Experience*, Tools Europe 2000, 2000.
[2] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Longman, 1999.
[3] M. T. Rose, *The Open Book, A Practical Perspective on OSI*, Prentice-Hall, 1990.
[4] W. R. Stevens, *TCP/IP Illustrated Volume 1 - The Protocols*, Addison-Wesley Longman 1994
[5] G. R. Wright, W. R. Stevens, *TCP/IP Illustrated Volume 2 - The Implementation*, Addison-Wesley 1995.
[6] ITU-T, *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model, Recommendation X.200*, ITU, 1994.
[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[8] F. Bussman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture: A System of Pattern*s, Wiley, 1996
[9] R.E. Johnson, *Frameworks = (components + patterns)*, Communications of the ACM, 40 (10): 39-42, October 1997.
[10] H. Hüni, R. Johnson, R. Engel, *A Framework for Network Protocol Software*, ACM, 1995.
[11] J. Zweig, *An Object-Oriented Framework for Implementing Network Protocols*, Master's Thesis, University of Illinois, 1991.
[12] J. Malka, E. Ojanperä, *CVOPS User´s Guide*, http://www.vtt.fi/tte/tte22/cvops/, Technical Research Center of Finland, 1998.
[13] P. Heinilä, *OVOPS Home Page*, http://ovops.lut.fi, Lappeenranta University of Technology, 1997.
[14] ITU-T, *Recommendation Z.100 "Specification and description language (SDL)"*, 1993
[15] J. Ellsberger, D. Hogrefe, A. Sarma, *SDL Formal Object-oriented Language for Communicating Systems*, 1997.
[16] P. Dyson, B. Anderson, *State Patterns*, Pattern Languages of Program Design 3, pp. 125-142, Addison-Wesley Longman, 1998.
[17] S. Yacoub, H. Ammar, *Finite State Machine Patterns*, Pattern Languages of Program Design 4, pp. 413-440, Addison-Wesley Longman, 2000.

Juha Pärssinen can be reached at the VTT Information Technology, P.O.Box 1203, FIN-02044 VTT, Finland; `juha.parssinen@vtt.fi`

Markku Turunen can be reached at the Nokia Research Center, P.O.Box 407, FIN-00045 NOKIA GROUP; `markku.turunen@nokia.com`