Pattern Language for Architecture of Protocol Systems

Juha Pärssinen, Markku Turunen

Introduction

This paper presents the pattern language for architecture of communication protocols. This is a pattern language that aids a developer faced with the task of designing a networked system for the very first time. The pattern language has to parts: one for communication protocol specification and one for communication protocol design. The part for communication protocol specification contains patterns for protocol structure and protocol messages. Patterns for communication protocol messages are one of the most important part of this pattern language, and also one of the most important part of any communication protocol architecture. The part of language for communication protocol design contains more implementation oriented patterns.

Specification and design of communication protocols are two different sides of the coin. Specification of communication protocol explains meaning of communication messages sent between protocols and tells practically nothing about system structure. Design of a communication protocol explains static and dynamic structure of protocol system and its layers, but messages are merely referred as *events* or *payloads*.

In this paper we use standard notation, UML, when we define patterns and examples. UML is used in at least two different levels: concepts and their relationships in specifications; design structures and their relations.

All patterns in this language and also pattern language itself are under continuous, but slow, development. During recent years authors have been developed several patterns and languages related to protocol architecture, and there are still lots to do in this area.

Protocol Specification Related Patterns

If you are interest in creating a new device, or a new class of node, which will live with other already existing nodes, somebody else has already specified the system for you. Proceed straight to implementation related patterns of this language, after you have understood their existing specification. Implementation patterns are introduced in the section " Protocol Design Related Patterns".

But if you are starting to build a completely new network system, continue reading.

Community of Nodes

Context

In the world of today there are many different kind of communication services. For example, there are services for human to human communication, services for humans to access to information, and even services for communication between autonomous systems without human interference.

These services, and others, are implemented using communicating devices. These devices use messages to communicate with each other.

Forces

All communicating devices have simple walk of life. First, one device becomes active. It can acquire information about other existing devices, it can initiate communication with others, and it can also respond to the initiation of communication from other devices. During the communication there can be a variety of errors that need to be handled. A device can stop communication, and finally it can become inactivate.

A variety of vendors may manufacture devices, but they have to co-operate together within a single network system. Additionally, a new device can be added as part of an old legacy system, or it can be the first existing implementation of a brand new system.

It is crucial to develop an unambiguous specification of system and behavior, else competing manufactures will create havoc by building system compliant devices that exclude competitors.

Less Machiavellian, designers of new functionality need to decipher the system definition with great precision to assure they will flawlessly interface with a legacy system.

Therefore:

Solution

Identify all communicating nodes and create a new community of nodes. Collect all needed responsibilities and functionalities and allocate them to the nodes. This can of course make also in reverse order: first collect functionalities, then identify nodes.

Some of these nodes in the community are endpoints and the rest are mediators. Endpoint nodes are the end-points of communication, while mediator nodes make communication possible between endpoint nodes.

When you have defined nodes, define communication connections between nodes. There are two different kinds of communication connections between nodes: real and virtual. Between two mediators, or between endpoint node and mediator node there are real communication connections, whereas between endpoint nodes there are always virtual communication

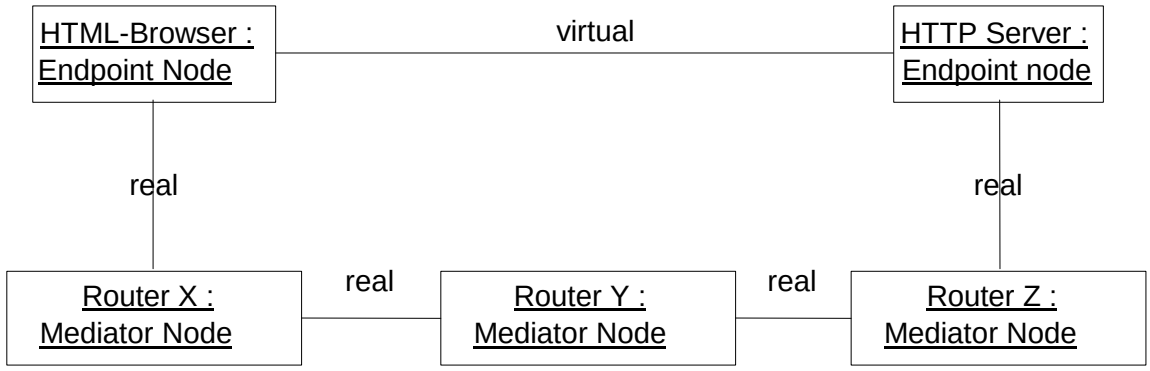
connections. To define high level messages between nodes you can use *Conversation Between Nodes* pattern.

Details

Responsibility and functionality collection and their allocation is an area, which needs its own pattern language, dealing not only with technical issues, but also economical and political issues. This pattern language is explained briefly in section "Related Pattern Languages".

Example

Cellular and computer networks are well-known examples of today's communities of nodes. In Internet HTML Browser and HTTP Server are examples of Endpoint nodes, between them there is a virtual connection. Routers are examples of Mediator nodes, as shown in UML collaboration diagrams. Between routers there are real connections.



Responsibilities of the different nodes are as follows:

- a browser displays web pages received from servers;
- a server constructs web pages and send them to a browser;
- a router transmits data packets from an end-node to an end-node.

Next Patterns

Now that you have defined the classes of nodes with their responsibilities, possibly using other pattern languages, turn your attention to how to these classes of nodes communicate with each other with *Conversation Between Nodes* pattern. When you want to split one node to manageable parts, use *Three Parts of a Node* pattern.

Conversation Between Nodes

A.k.a. Forget the Bloody Bits!

Context

After you have identified nodes from the *Community of Nodes* pattern, it is time to start specify messages between them. Nodes in the community are connected to each other. A connection is virtual, if it is between two endpoint nodes, and real, if at least one of the connection end-points is a mediator node.

Forces

Rules for communication between nodes are needed to make co-operation possible. These rules define what are the messages which nodes can send to each other and what is meaning of each message.

A message specification shall have enough description of the message format so that those interested in high level aspects can find the information they need, but not so much that multitude of details hinder understanding.

When messages are specified, experts from several fields must communicate with each other. Some might be expert on the behavioral part, knowing what information a message must carry so that participants of a communication procedure can have a common understanding of a state, what is requested and what are expected actions. Others might be experts on technical areas, like on methods of message specification.

It might tempting to start message encoding with bits and bytes because they are "concrete" in a sense that when one sees a bit table one can have an impression that one understands what a message contains.

Therefore:

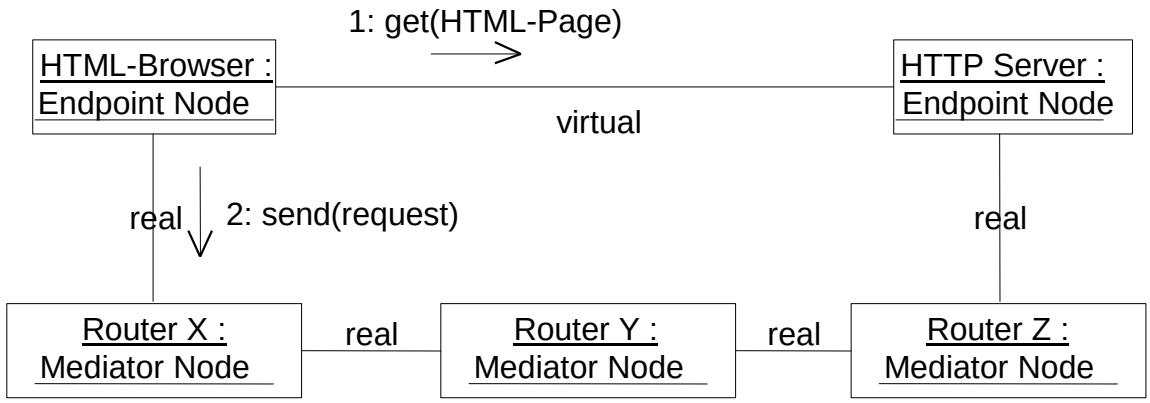
Solution

Make a few separate message specifications: one informal specification for messages between nodes; one for message logical, or abstract, contents; and one for message bit presentation. An informal specification of messages between nodes and a specification of nodes are used together to give a high level view of system. Different groups can discuss aspects of the system even if their background or interest differs. Specifications of message logical contents and bit presentation are considered in other patterns in this language.

In a specification of messages between nodes, specify only a descriptive name for each message. Only if it is not possible to make separation between messages using names then also specify a set of message parameters. Specify these parameters using descriptive names, not in bit level or any other detailed representation, if possible.

Example

In Internet between HTML Browser and HTTP Server there is a virtual connection. In this example the HTML Browser will ask for a formatted page from the HTTP server. This message is actually sent using other messages via a real connection to mediator node. In a message informal specification only high level messages are specified, e.g. get(HTML-Page) and send(request). Only meanings of these messages are specified, not detailed contents, or bit presentations.



Next Patterns

When you specify a message between classes of nodes it is possible that you need add more nodes to your community. In this case you might want to go back to *Community of Nodes* pattern. Otherwise, you have now defined classes of nodes and messages between them, turn your attention to how to split one node into manageable parts with *Three Parts of a Node* pattern.

Three parts of a Node

Context

After you have identified nodes from the *Community of Nodes* pattern, and messages between nodes from the *Conversation Between Nodes* pattern, you are ready to think about splitting these nodes in more manageable parts.

Forces

In real life, a device contains two distinct parts: software and hardware. Hardware is connected to physical media, and in the case of an endpoint node, software is connected to an end-user. In the case of mediator, software might be connected to an administrator.

The physical media between nodes offers only low-level communication service, moving bits from one point to another. This can be achieved by using, photons, radio frequencies, carrier

pigeons or other whatever available means. Physical media can be considered as a real connection.

End-users are interested only in very high-level aspects of communication; they want to send information to another end-user, for example contents of web pages.

Adaptation is needed between end-user and physical media. Also, if there are changes in physical media, then the software part of endpoint node and especially end-user doesn't want to know anything about them.

Therefore:

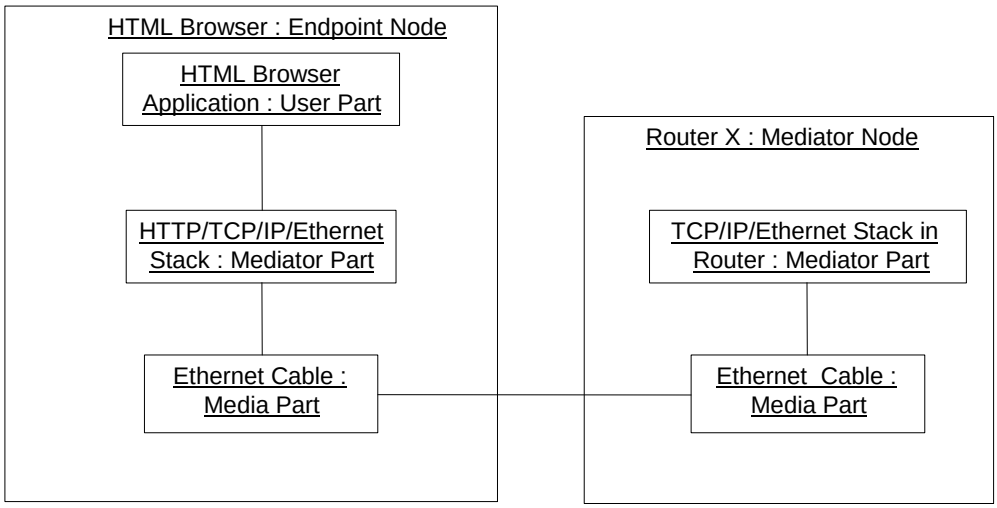
Solution

Separate structure of one node to three parts: the user part, the mediator part and the media part. In a mediator node there is not always a user part. A mediator part behaves as an adapter between a user part and a media part. A mediator transforms information from a user part to a bit-stream understood by a media part, and vice versa. A mediator can add also other functionalities to communication, e.g. reliability in case of unreliable communication channel.

Example

An HTML Browser has a protocol stack, which contains HTTP, TCP, IP and Ethernet protocols, and Ethernet cable. The HTML Browser Application is a user part, HTTP/TCP/IP/Ethernet protocol stack is a mediator part, and the Ethernet cable is a media part. The router has a protocol stack which contains TCP, IP and Ethernet protocols, which is a mediator part, and Ethernet cable, which is a media part, as shown below in the UML collaboration diagram.

Please note that in an Ethernet card, which typically is considered as hardware, also contains Ethernet protocol.



Next Pattern

Now your node has three parts: user part, mediator part, and media part. The next pattern to consider depends on your interest. If you are interest in a mediator part and its internal structure, then look at the *Elements of Mediator* pattern. If you are interest about the user part or media part then there are other pattern languages which you should read or even write down by yourself. User parts and media parts are connected to the mediator part using adapters as shown in the *From Scheme to Skeleton* pattern.

Elements of Mediator

Context

The mediator part behaves as an adapter between a user part and a media part, as shown in the *Three parts of a Node* pattern. The mediator transforms information from a user part to a bit-stream understood by a media part, and vice versa.

Forces

A user part's worldview and a media part's worldview are completely different. A media part knows only how to send bits to a media part located in next mediator or to an endpoint node via a real connection. A user part may know nothing about mediator nodes, it only knows about other user parts located in other endpoint nodes. However, the endpoint node might not know the exact location of the other node.

Either the media part and/or the mediator part may be replaced with another kind of media or mediator part. This replacement should be transparent to the user part.

Communication from one media part to another media part, which is located in another node, is not always reliable. However, the user part is expecting reliable communication. The user might expect a stream of communication, e.g. a videoconference, but the media part offers only service to send small packets.

Therefore:

Solution

Identify and define the needed functionalities for a mediator part. Separate distinct functionalities to their own subparts, or layers. You can use as an example of splitting functionalities into layers, the two well-known layer models: OSI reference model [OSI] and TCP/IP protocol stack [TCP/IP]. The OSI reference model and the TCP/IP stack are also studied in the Layers pattern [POSA]. Additionally, you can use two related pattern languages: Protocol Standardization and How to Eat an Elephant, if they are available when you read this. These pattern languages are explained briefly in section "Related Pattern Languages".

The mediator part is specified using a stack of layers. Layers are built on top of each other, higher layer uses the interface provided by its lower layer. Layers have different kind of services, which they offer to each other. The use of the interface hides lower layer implementation from layers above. This modularity and encapsulation of layers have been in use for years in the networking field before object-oriented experts claimed it as a fundamental way to program.

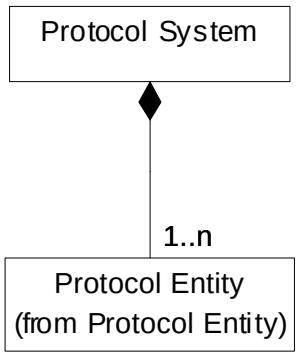
Every layer has to have rules describing how they communicate with the layer in the same level in the other party of the communication. Layers in same level are called as peer layers. These rules are known as a peer protocol. Connection between two peers is virtual. The combination of these layers is a protocol system.

The specification defines which are the layers, or entities, of a system, and defines which are the responsibilities and functionalities of an individual entity, and specifies how they are interconnected to each other. Interconnection between entities defines structure of the whole system.

The concepts and their relationships of this pattern are as follows:

- Protocol system encapsulates other components as a single system;

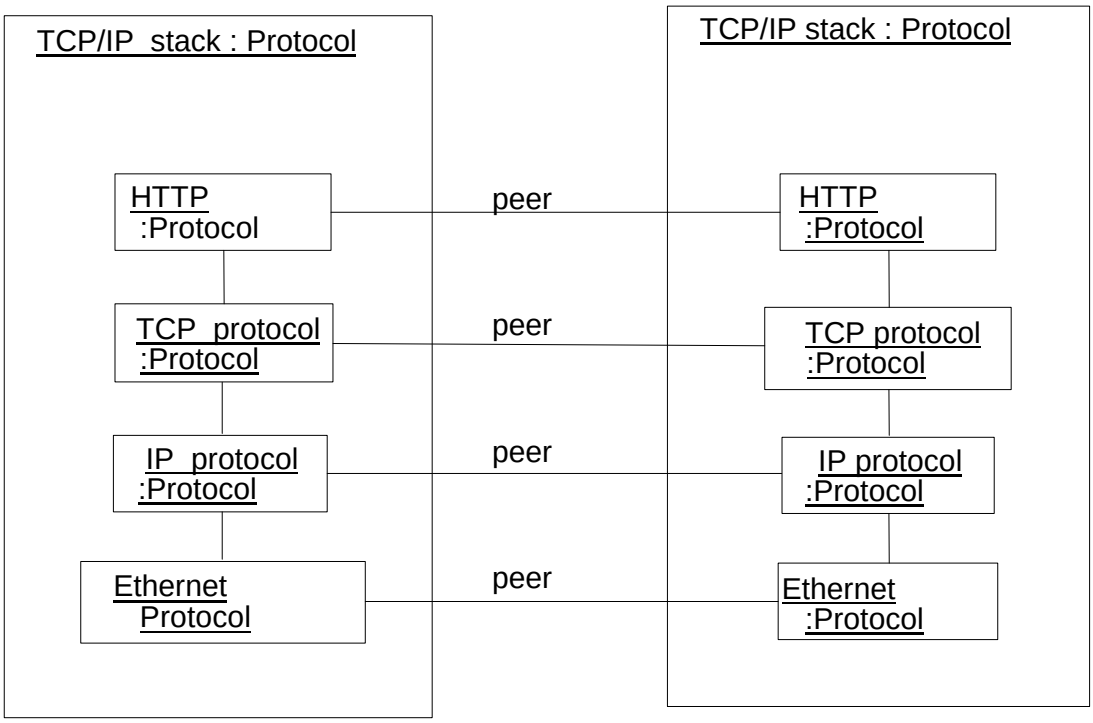
- Protocol entity represents a protocol layer.



Example

An HTTP/TCP/IP/Ethernet stack is a good example of a protocol system and the protocol entities which it contains, see the UML Collaboration diagram below. Between entities in same level there is peer protocol. Entities in same system are connected together. In this example, HTTP is a user level protocol; it delivers html-formatted textual pages between browser and server. TCP provides reliable transformation between systems, IP routing and fragmentation, and the Ethernet moves bits to the next system. The HTTP has been introduced after the TCP and IP have been introduced. It provides distinct functionalities and uses the functionalities provided by TCP.

:



Next Pattern

Now that you have specified your mediator using protocol system and protocol entities, turn your attention to the messages between protocol entities. At this point you have messages from the *Conversation Between Nodes*, use these messages as starting point with the *Informal Sentences for Entity Communication* pattern and/or the *Formal Sentences for Entity Communication* pattern. If you want to provide an abstract view of messages without too many details, use the *Informal Sentences for Entity Communication* pattern. But if you want to specify all properties of a message, excluding only its bit presentation, use the *Formal Sentences for Entity Communication* pattern.

Messages are assigned to interfaces with *Interfaces of Entity* pattern. An interface specification of an entity with messages is essential because it defines the actual protocol between entities. Without interface specification, it is not possible for two different implementers to build protocol systems, which will co-operate. The implementation of a protocol system and a protocol entity are considered in *From Scheme to Skeleton* and *One Level at a Time* pattern.

Informal Sentences for Entity Communication

A.k.a. High Level Abstract Syntax

Context

After you have identified entities from *Elements of a Mediator*, you are ready to think about messages between entities and their logical contents. At this point you also have to consider higher level messages from the *Conversation Between Nodes*.

Forces

You want to provide an abstract view for those not interested in message details.

In addition to message information content there may be additional required message properties for messages, such as priority or optionality.

The higher the level of protocol is in e.g. OSI reference model, the more complex the message structure is likely to be.

Your specification should have enough information so that bit level representation can be derived from your logical description.

The use of formal methods makes description less ambiguous, but people might be uncomfortable with them. From someone's point of view formalism hides the idea of concepts, because notations might look too much like programming languages. From another's point of view, a formal notation hides the bits, and thus removes control from their hands.

Furthermore, the following forces from *Conversation between Nodes* have to be considered in this level:

- A message specification shall have enough description of the message format so that those interested in high level aspects can find the information they need, but not so much that multitude of details hinder understanding.

- When messages are specified, experts from several fields must communicate with each other. Some might be expert on the behavioral part, knowing what information a message must carry so that participants of a communication procedure can have a common understanding of a state, what is requested and what are expected actions. Others might be experts on technical areas, like on methods of message specification.

- It might tempting to start message encoding with bits and bytes because they are "concrete" in a sense that when one sees a bit table one can have an impression that one understands what a message contains.

Therefore:

Solution

When specifying the message's logical contents, the following must be specified, at the minimum:

- message name;
- specification of message parameters;
- specification of valid value sets for parameters;
- specification of other logical parameter properties (e.g. multiplicity, optionally or conditionally).

Consider what other kind of message properties are needed. Consider how message might evolve in the future and how to be prepared for it.

Next Pattern

Now that you have specified messages and their logical structure, the next issue to consider is how to assign these messages into interfaces with *Interface of an Entity* pattern. If you want to specify in a more detailed level of messaging, including how these details are implemented see the *Formal Sentences for Entity Communication* pattern. Specification for message bit presentation is discussed in the *What to Send to the Line* pattern.

Formal Sentences for Entity Communication

A.k.a. Low Level Abstract Syntax

Context

After you have identified entities from *Elements of a Mediator*, you are ready to think messages between entities and their logical contents. At this point, you also have designed the higher level messages from the *Conversation Between Nodes*. You might also have identified messages from *Informal Sentences of Entity Communication* pattern.

Forces

You want to provide all message details.

You have messages, which are complex, and you have selected those message definitions that will be robust and future-proof. This can generate complex message definitions where the actual contents are hidden behind mechanisms of extensibility and other features.

While an informal description aids in one’s initial understanding of a messaging system, a precise formalization must exist as a basis in a court of law to determine when exclude devices claiming to be network compliant, when they are not.

The following forces from *Informal Sentences for Entity Communication* have to also be considered in this pattern:

- In addition to message information content there may be additional required message properties for messages, such as priority or optionality.

- The higher the level of protocol, the more complex the message structure is likely to be.

- Your specification should have enough information so that bit level representation can be derived from your logical description.

- The use of formal methods makes description less ambiguous, but people might be uncomfortable with them. From someone's point of view formalism hides the idea of concepts, because notations might look too much like programming languages. From another's point of view, a formal notation hides the bits, and thus removes control from their hands.

- A message specification shall have enough description of the message format so that those interested in high level aspects can find the information they need, but not so much that multitude of details hinder understanding.

- When messages are specified, experts from several fields must communicate with each other. Some might be expert on the behavioral part, knowing what information a message must carry so that participants of a communication procedure can have a common understanding of a state, what is requested and what are expected actions. Others might be experts on technical areas, like on methods of message specification.

- It might tempting to start message encoding with bits and bytes because they are "concrete" in a sense that when one sees a bit table one can have an impression that one understands what a message contains.

Therefore:

Solution

When specifying message logical contents the following must at least be specified:

- message name;

- specification of message parameters;

- specification of valid value sets for parameters;

- specification of other logical parameter properties (e.g. multiplicity, optionally or conditionally).

Determine which concepts are needed for messages and information elements (parameters) a message is composed of. These might include concepts like

- presence (mandatory, optional, conditional)
- repetition
- extensibility
- versioning
- criticality

Separate a concept and how it is realized. Provide a vocabulary for high-level specifiers. Specify unambiguous meaning for the concepts.

Next do a refinement step, where you realize the concepts using a more detailed notation. You might need a specification framework. Your formalism should support for all the needed concepts. If your notation does not explicitly support a concept, then you have to specify how a concept is mapped to the notation. To better understand this, consider the analogy of the "interface" concept and its realization in both C++ and Java.

There are several patterns in this language, which support this pattern: *Message Identification*, *Message Versioning*; *Tail Extensions*; *Parameter Container*.

Next Pattern

Now you have specified messages and all of their properties excluding their bit presentation. The next issue of concern is how to collect these messages as interfaces with *Interface of an Entity* pattern. The specification of message bit presentation is considered in *What to Send to the Line* pattern.

Interfaces of an Entity

Context

After you have identified entities from the *Elements of Mediator* pattern and messages between them from *Informal Sentences for Entity Communication* pattern and/or *Formal Sentences for Entity Communication* pattern; you are ready to think how those messages are collected.

Forces

There is a protocol between peer protocol entities. This protocol can be symmetric or asymmetric. In a symmetric protocol peer entities do not have fixed distinct roles, like server and client, but they can change their roles. In an asymmetric protocol, peer entities have distinct roles. Entities in the same system are also connected. This means that a protocol entity has two different kinds of paths: a virtual message path to its peer entity, and real message

paths to its adjacent entity. Furthermore, an entity has an inner state during communication, which requires careful design and documentation.

Therefore:

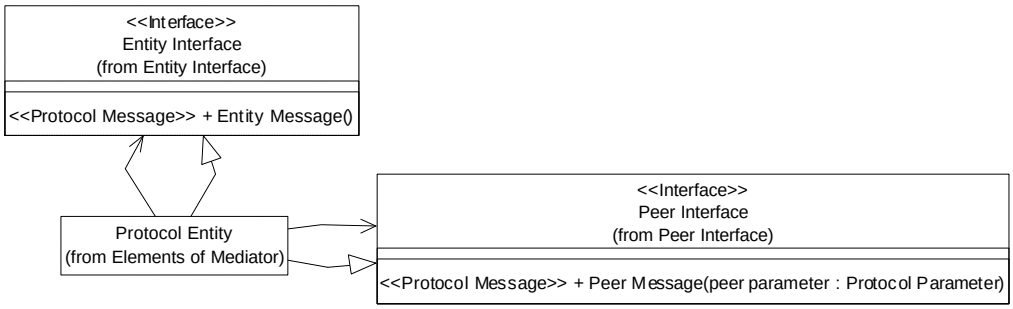
Solution

Specify interfaces for both of these paths. An interface specification contains messages, which other entity uses to send messages to current entity. In the same way a current entity uses another entity's interface to send message to it. In protocol specification terms, the interface means a detailed specification of both incoming and outgoing messages. However, in object-orientation interface terms, it means only incoming messages (i.e. method calls). In this pattern language we use term interface as it is used in object-orientation. This means that two interfaces are needed to specify a protocol between entities. One is an interface, which an entity realizes (i.e. what are the messages that an entity can receive); the other is an interface, which an entity depends on (i.e. what are the messages that an entity can send).

If UML is used as a specification notation the two different message kinds can be expressed as follows:

- The entity interface contains messages, which are used in communication between two entities in the same protocol system. These are all possible messages, which can be received by a realizing entity.

- The peer interface contains messages, which are used in communication between entities located in the peer protocol system. These are all possible messages, which can be received by realizing entity.



Details

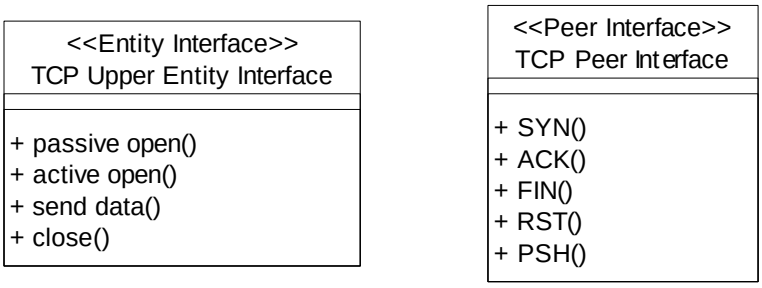
Some of these interfaces are specified in some acceptance level: de-facto, de-jure etc. An interface between peer entities is always specified. In protocol standards a standardized interface is often called a normative interface. Interface between entities in the same system is not always specified, but there is often a recommended or informal entity interface. This recommended interface is often called in protocol standards as a non-normative interface.

There are several methods how to specify externally visible behavior of a protocol entity. The most used notations are (message) sequence charts and finite state machines (FSM). Patterns

related to FSMs can be found from [FSMP]. These two notations provide different point of views: to specify complete behavior of one protocol entity use a finite state machine; to specify some relevant part of interaction between two or more entity use sequence diagrams.

Example

The following figure contains examples of TCP peer and entity interfaces. SYN, ACK, FIN, RST and PSH are different possible messages, which can be sent between peers. Entity interface contains possible messages which application can send to TCP.



Next Pattern

Now that you have specified the network system interfaces, you are ready to proceeding to our implementation patterns. The first implementation pattern to consider is *From Scheme to Skeleton* pattern.

What to Send to the Line?

A.k.a. Message Transfer Syntax

Context

You have specified a message abstract syntax using *Informal Sentences of Entity Communication* and/or *Formal Sentences of Entity Communication*. This abstract specification defines a logical content of a message that is sent between peer systems.

Forces

A message logical content is specified, but its local bit presentation can vary in different systems depending on many implementation issues like integer presentation in underlying hardware etc.

A message can be presented in a local system in a string of bits or bytes, but it can also be built as a complicated object tree.

There is a need for a common transmission format so that a message can be encoded (serialized) into a string of bits or bytes and then the encoded bits or bytes can be sent over a transmission media.

The available bandwidth might be small resulting need for short peer messages.

A design tradeoff exists between squeezed or robust encoding.

Peer messages are sent over a transmission media and thus must be encoded (serialized) to strings of bits or bytes. A receiver must be able to determine the type of a message that is contained in a bit string.

Therefore:

Solution

Specify the message bit presentation, and how this bit presentation is mapped to the message logical content and vice versa.

The separation of logical message contents, Message Abstract Syntax, specified using *Informal Sentences of Entity Communication* and/or *Formal Sentences of Entity Communication*, and message transmission format, Message Transfer Syntax, makes it possible to refer to information in the message parameters. The complexity of transfer syntax can be encapsulated and hidden from other parts of a specification.

Next Pattern

Now that you have specified the bit presentation for messages and mapping how this bit presentation is related to the message logical content, the next issue is how to move a bit string to a peer entity using a lower layer service. This is considered in *Piggy-packing* pattern.

If you are interest in how message encoding and decoding is done, see the peer proxy in the *One Level at a Time* pattern.

Message Versioning

A.k.a. Prepare for Change

Context

Bugs are found in a protocol specification. New protocol features are introduced. This results in an evolution of the protocol specification.

Forces

Modification of a protocol results in a need to extend or to modify message contents.

Backwards-compatibility. (You do not want to update 10.000.000 cellular phones just because a new version of a protocol was just released...)

Future-proofness. (Of course version 1.0 of a protocol does not have any bugs or need for enhancement...).

Compactness vs. robustness. Robustness may require auxiliary structure information in messages like encapsulation and identifiers. If bandwidth is scarce then the cost of such extra information and structure may exceed achieved benefits.

Humans understand so little.

Therefore:

Solution

You have two different ways to implement message versioning: adding message version identification to each message; adding to the message tag that informs starting of new or optional section; adding to each parameter of each message a tag which informs if the parameter in concern is critical or not.

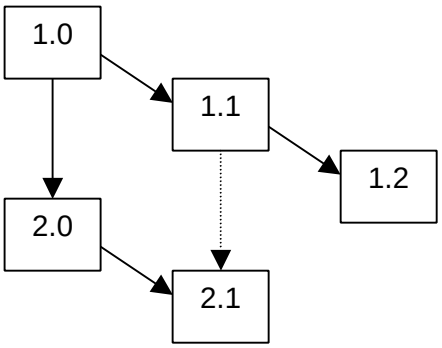
Adding message version identification to each message is straightforward solution, but if there are many versions of messages the implementation with many version could be complicated.

Another way is to add to the message a tag to mark new or optional section. Receiver can choose if optional section is considered at all.

Yet another way is to add to each parameter of message a tag which informs if the parameter in concern is critical or not. Receiver can choose which way message containing unknown parameters is handled. Receiver can drop whole message or only unknown parameters, and make decision based on criticalness of unknown parameters.

To make decision which of these ways to use consider which of the following features are needed:

- addition of new message parameters;
- modification of existing message parameters;
- removal of existing message parameters;
- replacement of existing message parameters;
- need for skipping of unrecognized message parts;
- tree-like version branching, i.e. different protocol versions may evolve in parallel and produce sub-versions



Determine how the selected features are implemented using your specification method.

Make clear separation between mechanisms that provide the needed features and the actual data carried by the messages. Hide the mechanisms from the view of specifiers that only need to access the information in the messages.

Next Pattern

To extend messages you can use two different patterns: the *Parameter Containers* and the Tail Extensions patterns.

Tail Extensions

Context

Protocol messages are evolving as shown in *Message Versioning* pattern. You don't have lot of bandwidth for messages.

Forces

Protocol messages shall be extended but there is not a lot of bandwidth for message encoding. The extension mechanism should be as light as possible producing a very small encoding.

Therefore:

Solution

Encode all new parameters and extensions to old parameters at the end of a message regardless of their logical position in a message. A receiver that does not know the extensions is designed to ignore the trailing bits.

If *Tail Extensions* are used to replace existing message parameters then you must specify how a device handles old parameters.

- Old parameters are left as garbage. This means that there are two different parameter values for one logical parameter and different protocol versions interpret messages differently.

- Old parameters can be removed. This means that there shall be a presence indication for every parameter even if it is mandatory. If an old parameter is omitted then old protocol implementations do not receive a complete message from their point of view.

Next Pattern

This is a leaf pattern in this language.

Parameter Container

Context

Protocol messages are evolving and they have to extend, as shown in Message Versioning pattern. In this case, bandwidth is not a problem.

Forces

Protocol messages shall be extended; there are no strict limitations of the use of bandwidth. A receiver shall be able to identify and skip over unknown new parameters.

Therefore:

Solution

Provide a generic container structure, which can be used to encapsulate message parameters. It shall at least contain identification and delimiting for a parameter value. It can also contain different kinds of parameter properties, e.g. if you use a container with *Message Versioning*, you should add critical or uncritical tags to each container.

Next Pattern

This is a leaf pattern in this language.

Piggy-packing

A.k.a. Data message or Data Transparency

Context

There is a stack of protocol layers. Peer entities communicate with each other using virtual connection. When a message is sent to a peer, it must be sent to a peer entity using lower layer services via real connections.

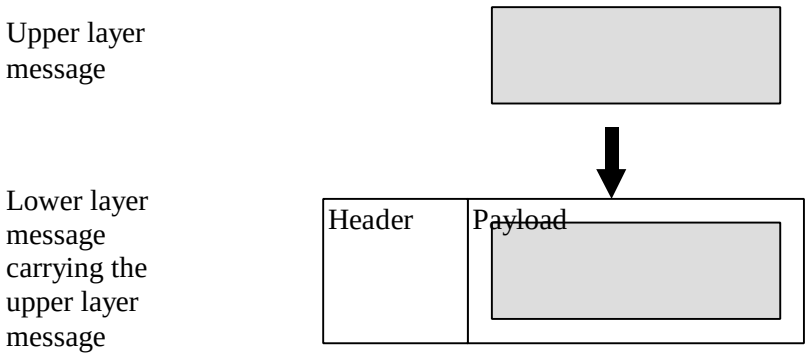
Forces

Keep layers decoupled. The same lower layer could be used with several different kinds of upper layers.

Therefore:

Solution

Provide a data message that is capable to carry encoded upper layer messages. An encoded upper layer message is seen as a payload and it is transparent from the lower layer point of view.



Next Pattern

Now you have an encoded message and can send it to the peer entity, the question to consider is how the peer entity identifies the nested upper layer message. This is considered in the *Message Identification* pattern.

Message Identification

Context

A lower layer data message carries encoded messages as a byte string.

Forces

An upper layer has to identify what kind of message it has been received.

New messages might be introduced in the future.

How unique shall the identification of a protocol message be? Shall all the messages within a protocol be distinct, or is it enough that only messages that go in one direction are distinct, or

are there several protocols that must co-operate in one network node and all the messages shall be distinct.

Therefore:

Solution

Provide a message identification field (or fields) that determine the kind of a message. Such identification can be external to protocol messages or it can be included in the messages themselves.

In the case of external identification the lower layer data message contains an identifier field in addition to the payload field. Furthermore, a table, which maps identifiers to messages, shall be specified.

In the case of internal identification, the lower layer data message contains just the payload field. It is the responsibility of upper layer messages to identify themselves. Usually there is an auxiliary identifier field in a message. The field is used as a selector over a possible set of messages.

If messages need to be globally distinct then a known mechanism shall be used for the identifier fields. If the need for uniqueness is smaller, then a lighter means can be used (e.g. a closed range of integer values).

Next Pattern

Now you have defined identifications for messages. If you are interest in how message identification is done, see the peer proxy in the *One level at a Time* pattern.

Protocol design related patterns

Following patterns are explained here only briefly.

From Scheme to Skeleton

Context

You have a specification, which contains protocol system and protocol entities. Peer and Entity Interfaces are specified for each protocol entity.

Forces

Previous patterns, which were related to specification, balance forces what are the components that a system is composed of and which contain the responsibilities of these components. Now there are two new questions: how the components are interconnected, and how a system communicates with its environment.

A system which is a logically unity, might be in real life distributed to separate processes or even devices.

Layers should be decoupled. This is easier for humans.

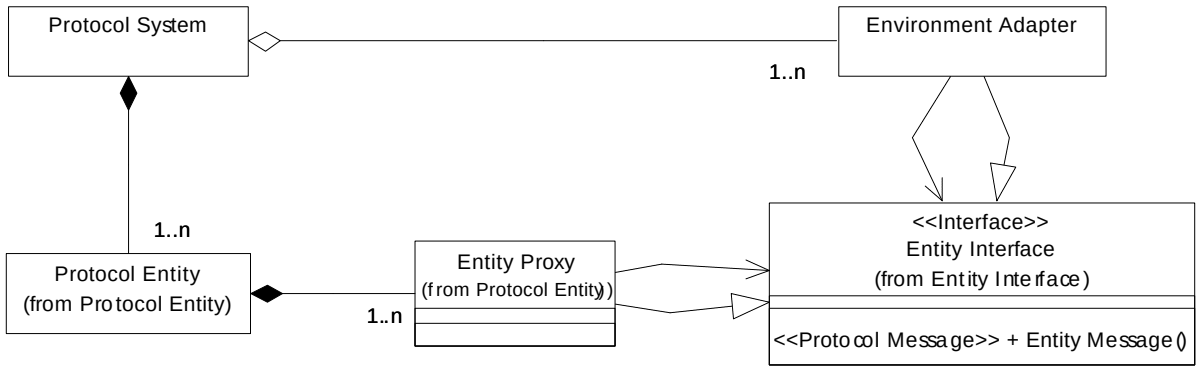
Therefore:

Solution

The protocol system pattern encapsulates the whole protocol system, and forms the basis for implementation of whole system. The protocol system contains entities, which are presented in specification, but it also contains other components, namely, the entity proxy and the environment adapter.

An entity proxy realizes entity interface and depends on the entity interface of another protocol entity. It handles communication between two entities in the same protocol system. It interprets a message, which is received from another protocol entity. It also produces message, which is sent to another protocol entity in the same system.

Environment adapter presents an environment entity, e.g. Ethernet hardware in TCP/IP. It realizes and uses its own entity interface.



Next Pattern

Now you have protocol system with entities that can communicate with each other, and the whole protocol system is connected to its environment. The next design issue is how a peer protocol is implemented in a protocol entity. This is considered in next pattern, *One Level at a Time*.

One Level at a Time

A.k.a. Layer Implementation

Context

You have a protocol system, which contains entities, and these entities are connected to each other. There exists a specification about peer protocol between entities in same level in protocol stack.

Forces

A protocol entity is capable of sending or receiving information, and it has an inner state during communication. A protocol entity can manage possible multiple concurrent communication sessions, which have their own states. Communications 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.

An entity has a virtual message path to its peer entity, and real message paths to its adjacent entity.

The implementor of an individual entity needs to consider entity initialization in a normal case or after a malfunction, as well as other possible errors during the entity's existence and also address entity shutdown.

Therefore:

Solution

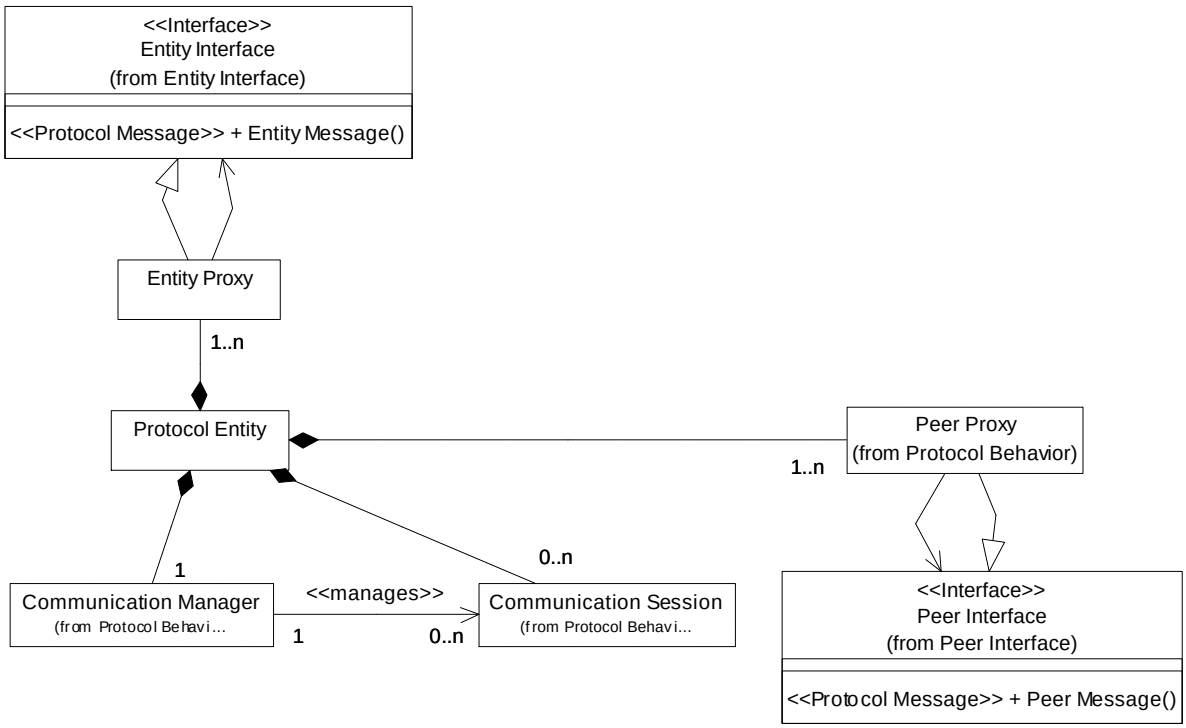
The *One Level at a Time* pattern contains:

- An entity proxy, from *From Scheme to Skeleton* pattern, handles communication between two entities in the same protocol system. It interprets a message, which is received from another protocol entity. It also produces a message, which is sent to another protocol entity in the same system.

- A peer proxy handles communication between entities located in the peer protocol system. It interprets a message, which is received from a peer entity. It also produces a message, which is sent to another entity in a peer system.

- In a case of connection-oriented protocol a communications manager creates, controls, and closes sessions as needed.

- A communication session handles communication between two communicating peers. It uses peer proxy to send and receive messages



Next Pattern

How to implement this pattern, see Master-Slave [POSA] and Finite State Machine Patterns [FSMP].

Related Pattern languages

There are two pattern languages, which are intensively used with this pattern language: Protocol Standardization, and How to Eat an Elephant.

Protocol Standardization Pattern Language

The protocol standardization pattern language considers protocol resources and functionality collection as well as their allocation to relevant nodes and layers. This work deals not only with the technical issues, but also the economical and political issues. Here is no reference to this language. It is not yet written, or at least we don't know.

How to Eat an Elephant Pattern/Pattern Language

This pattern or pattern language considers problems, which always are faced during design and implementation of complicated systems: "How to manage its complexity? How it is possible to make it at all?" As reader might already know, elephants are usually eaten one bite at a time. We would like to include a reference to this language, however we don't know if anybody has written this using pattern notation. It is worth of it.

Acknowledgements

We would like to thank our shepherd, Norm Kerth, for his valuable comments and support during shepherding process. Without his excellent support and inspiring feedback to us this work would not exists. With his help we started from our group of patterns [PPSA] and ended up to this pattern language, which is still under development. We would like to thank also all participants of our workshop at EuroPLoP2001. They all helped us to improve this pattern language.

References

[PPSA] J. Pärssinen, M. Turunen, *Patterns for Protocol System Architecture*, a pattern workshop paper presented at PLoP2000, August 13-16, 2000.

[OSI] M. T. Rose, The Open Book, A Practical Perspective on OSI, Prentice-Hall, 1990.

[TCP/IP] W. R. Stevens, *TCP/IP Illustrated Volume 1 - The Protocols*, Addison-Wesley Longman 1994

[POSA] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture: A System of Pattern*s, Wiley, 1996

[FSMP] 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`