How Layering Protocol Software Violates Separation of Concerns

Matthias Jung, Ernst W. Biersack
Institut Eurécom, 2229 Route des Crêtes, 06190 Sophia Antipolis, France
{jung,erbi}@eurecom.fr

In ECOOP Workshop on Aspects and Dimensions of Concerns (ADC), June 2000, Cannes, France

Abstract

We show how layering in communication software violates the principle of separating concerns. Examples illustrate that

- layering couples receiver-and sender functions,
- sharing message headers among layers creates dependencies between layers that largely compromise re-usability,
- QoS management is scattered over all layers, and
- layers are tightly coupled with the sub-system that executes it.

Layering hence leads often to large and hardly re-usable modules, tangles possibly independent parts of code, and makes implemented protocols difficult and error-prone to extend.

I. Introduction

Many new distributed applications came up with the success of the Internet. Concerning the underlying transport protocol, these applications have the choice between a fully reliable and ordered service (TCP [Postel 81]) and a simple datagram service (UDP [POST 81]). This "all-or-nothing" choice reflects the diversified needs of new applications often in an insufficient manner. As a result, own home-grown protocols must be implemented, which often involve transport layer functionality and thus become cumbersome, difficult, and expensive to realize.

Organizing protocols in re-usable components that are integrated in sub-system frameworks are the common way to cope with the cost of implementing protocols [Hutchinson 91], [Unix 90], [Hüni 95]. These frameworks normally encapsulate protocols in layers and allow to combine them to protocol stacks.

One characteristic of layered architectures is that layers communicate exclusively with their neighbour layers. The standard interface between protocol layers comprises a *down-call* function –called by the higher layer to pass a message to the lower layer– and a *up-call* function –called by the lower layer to pass a message to the higher layer. In the output direction (down-calls) each layer adds specific information to the message header. In input direction (up-calls), this information is then stripped by the corresponding layer of the peer entity and used for state update. Layering of protocols has been standardized in the 7-layer ISO/OSI model [ZIMM 80]. The layers used in common communication systems are very coarse-grained and themselves often badly structured. We rather want to re-use a single protocol function than, for example, the complete bundle of mechanisms implemented in a TCP layer.

There is a common agreement about the advantages of layering: simplicity, the re-use of whole building blocks, and the reduction of dependencies among parts of the system compared to a monolithic design [Buschmann 96]. Although separating different concerns in communication systems is the primordial motivation behind layering, this article gives a number of examples that show how layering can violate the separation of concerns in protocol software. The goal of this paper is thus to contribute to a discussion that seeks to overcome the liabilities of layered protocol architectures.

II. Tangling Concerns in Layered Protocol Software

A. Tangling of Protocol Functions

Example Protocol 1: Consider a stop-and-wait protocol. The sender gives each application data chunk a sequence number, copies, buffers, and sends it. The sender also maintains a timer, which is set each time a chunk is send (the timer-interval is pre-defined and constant). The receiver acknowledges each incoming packet based on its sequence-number. When the sender receives an acknowledgement for the currently buffered chunk, it frees its buffer, stops the timer, and proceeds with the next data chunk. When the timer expires before an acknowledgment arrives, a loss is assumed and the currently buffered data chunk is resent.

A.1 Receiver- and Sender-Functions

This simple protocol would typically be implemented within a single layer. When the layer is used on the sender side, application data is handled by *down-calls*. If no data chunk is outstanding (unacknowledged), the data chunk is sequenced, buffered, and given to the lower neighbour layer (in this case sent). If the last data chunk sent is not yet acknowledged, the chunk is written to a queue. During an *up-call*, the layer must be aware if it acts in the role of sender or receiver, and then check if the corresponding header fields of the message are correct. It then either implements acknowledgement handling (remove outstanding data chunk from the buffer and process the next data chunk in the waiting queue) or user data handling (strip header information and give the message to the higher layer). This is illustrated in Figure 1.

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

The implemented layer is also hardly re-usable for uni-directional communication. Modifications would be necessary to suppress functionality at each side. Additionally, many simple sender functions in protocols (such as logging or gaining statistics) do not even have a peer function at the receiver. The down-call interface is not implemented in these cases. Hence, layering is for fine-grained protocols often a too inflexible abstraction. It mixes up receiver- and sender functionality and impedes that these functions can be re-used independently.

Problem 1: *Layers are implicitly symmetrical and thus tangle receiver- and sender functionality.*

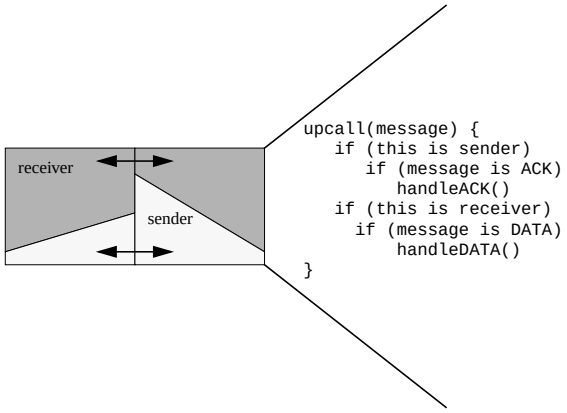

Fig. 1. Mixing Receiver/Sender functions

A.2 Inter-Layer Communication

In order to show another violation of concerns in layered architectures, we take the example from the last section and extend it by using an adaptive instead of a static timer-interval. In networks with long RTTs a static timer-interval causes too many time-outs and retransmissions. In environments with short RTTs, loss is detected lately.

In order to determine a good timer-interval, the sender must estimate the round-trip-time to the receiver by measuring and averaging the times between the emission of the data chunks and the reception of the corresponding acknowledgements. How would we extend our protocol above to allow for dynamic timer-intervals? Should we implement a new layer or modify and extend the old one?

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

Implementing stop-and-wait and RTT-estimation in two distinct layers, however, raises other problems. It would be extremely inefficient, if both layers would implement their algorithms completely independently from the other, e.g.

when the RTT-estimation layer would make measurements based on own data-requests sent from time to time, which are answered by the RTT-estimation peer-layer. This would cause processing overhead and waste of network resources.

The only way to deal with this problem, is breaking up the layers and allowing layers to share header data and exchange control messages. When two layers are operating on the same header field, both need to keep information about this header field (how to parse it, how many bytes are involved, how to represent it). Any changes – imagine that we realize that the sequence-number space in the message header is not sufficient and reserve now 4 instead of 2 bytes in the message header – concern both protocol layers and require modifications possibly at different places in the protocol code. This renders tailoring of protocols fairly error-prone. Additionally, the question arises, which of the two layers is responsible to add and strip the respective information from the message to make the message readable for the adjacent layers. Layers operating on the same header fields are implicitly tangled and can hardly be re-used without modifications.

Problem 2: *Sharing header data between layers is crucial for reasons of efficiency, but scatters information about headers over multiple layers and creates dependencies between layers.*

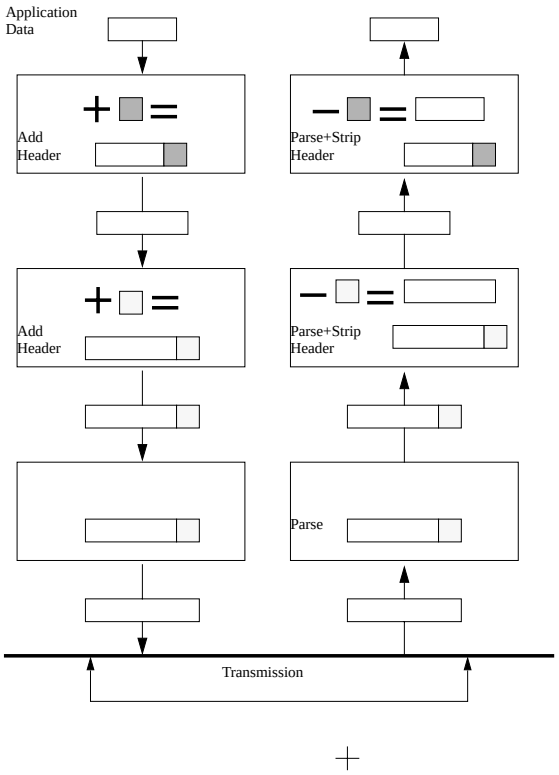

Fig. 2. Sharing Header Fields couples Layers

Figure 2 illustrates this problem. The two higher layers work as normal, the lowest layer however uses data already appended by its higher layer. It therefore does not append header data, but assumes that the higher layer already has done this. In input direction, the lower layer parses the message, but does not strip its data (latter will be done by the middle layer). Hence, the lowest layer can not be re-used without modifications in other layered systems (or together with the middle-layer).

B. Tangling of Protocol Functions with Vertical Aspects

Example Protocol 2: Consider a protocol that provides reliable transmission (EC) and rate-based flow/congestion control (FCC). This protocol is divided into two layers: the FCC on top and EC as the lower layer of FCC. We don't consider the protocol layers below (which could be UDP/IP/Ethernet in the operation system kernel).

In output direction, FCC receives data from the application and temporarily buffers it. The buffered (and possibly accumulated) data is given to the neighbour layer in one message according to a rate $1/\tau$, i.e. FCC assures a time-

interval of at least τs between two packets. EC adds to each message a sequence-number, copies it into a buffer, and sends it. When the buffer was empty before, a timer is started. Upon timer expiry, all packets in the buffer are re-sent and the FCC module is notified to decrease its sending rate.

In input direction, the receiver's EC layer triggers an acknowledgment for each correctly arriving message (i.e. with the right sequence-number). The receiver's FCC layer has no work to do but forwarding the message up to the application. When the sender's EC receives an acknowledgment, it removes the copy from the buffer. When there is still data in the buffer, the timer is reset.

B.1 Quality of Service

Imagine a telnet-like application using the protocol above. There are some control keys like ESCAPE, which should have higher priority than simple letters like A,B,C. Control keys thus must somehow overtake simple letters. In order to realize this, FCC must recognize control keys and give them directly to EC without introducing any delay. When EC recognizes control keys, it must stop processing current data (if there are some) in favor of control keys, and assure that the control key data is sent before other data. In ideal, every layer should provide an interface for high priority data and do its best that control keys arrive as fast as possible at the destination. But this is definitely not the original task of the layer, and rather relates to a different aspect of the communication system.

There are a number of similar examples, where the introduction of a new service or new mechanisms cause cascades of code modifications in almost every layer. Almost any form of QoS differentiation involves functions at all system levels to be effective. This also applies to measuring QoS and the introduction of billing mechanisms.

It may even happen that the change of behaviour of a layer requires changes of other layers. When the physical layer (the transmission medium) changes, certain mechanisms in higher layers may not be appropriate any more: e.g. while TCP/IP works well over long-hauled trunks, it shows bad performance over satellite networks and needs fixing.

Problem 3: *Layers only impose horizontal structuring, while for certain aspects of protocols a vertical structure would be more appropriate*

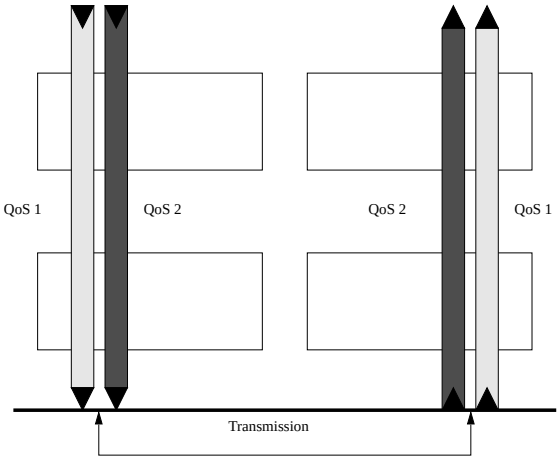

Fig. 3. Scattering QoS management over layers

Figure 3 shows two streams with different QoS requirements passing a layered system. Every layer has to cope with the respective requirements.

B.2 Interaction with Sub-System

In order to increase flexibility, maintainability, and portability, protocols are embedded in a sub-system that provides development facilities and execution support of protocol stacks (like X-Kernel [Hutchinson 91] or STREAMS [Unix 90]). One important aspect of such environments is the thread-model used to execute protocols. The best-known thread models are thread-per-layer and thread-per-message. In a thread-per-layer model, each layer runs an own thread to process data. Communication with the neighbour layers are done via queues. The thread looks into the queues, reads a message, and performs processing of that message. While this model is easy to implement, it suffers from overhead due

to the queues and context switches. For a high-number of layers, this model would be unacceptable. In a thread-per-message model, a thread is created for each message (or better taken from a thread-pool) that guides a message through the entire protocol stack. Carefully implemented, this model can provide better performance due to less CPU overhead –particularly for stacks with a high number of layers. At the other hand, this model more complicated to implement. Both models are depicted in Figure 4.

We now examine the influence of the thread-model on the implementation of a layer. For the thread-per-message model, each time a new message is created (e.g. an outgoing acknowledgment by the EC layer), a new thread must be allocated to execute processing of this message – and each time a message is thrown away, the thread of this message must be stopped (e.g. an incoming acknowledgment after freeing the buffer in the EC layer). Hence, every layer must have access to the thread-pool. For the thread-per-layer model, new messages are appended to the queue of the adjacent layer. Throwing a message away just means to not forward it. Neither inter-layer communication, nor thread-allocation are original tasks of a protocol. All sub-system related code should clearly be separated from protocol processing. If not, it is nearly impossible to later change the thread-model applied to a protocol stack. The code controlling the threads is scattered over all layers.

The thread-model is only one example for tangling protocol code with sub-system code. The same problem can be observed when protocols are simulated for reasons of testing. The creation of log-files for debugging also belongs to this family of aspect tangling.

Problem 4: *Layers tangle protocol code with code of the sub-system that executes the protocol code. Changes of the sub-system may concern all layers.*

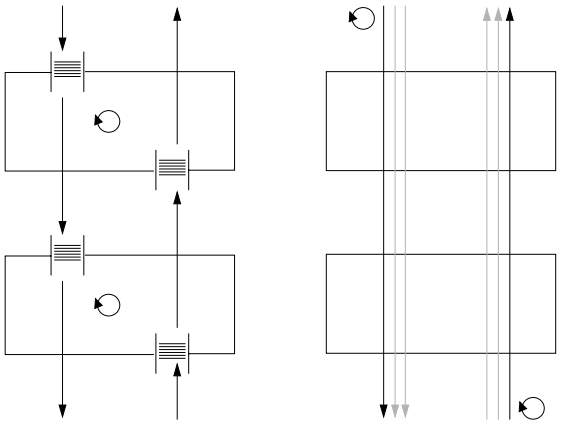

Fig. 4. Different Thread-Models

III. Conclusion

We gave four examples to illustrate how layered protocol architectures violate the design principle of separating concerns. 1) The symmetric design of layers leads to tight coupling of possibly independently re-usable receiver- and sender functions. 2) Efficiency reasons often force layers to share message header information. As a consequence, dependencies between layers are created that make layers hardly re-usable. 3) Quality-of-service management has been identified as an own aspect within a protocol system. In practice, every layer must be extended for the realization of QoS. 4) Finally, we showed how layering contributes to the tangling of protocol code with sub-system dependent services like thread control.

Most of the examples suggest that the strictly horizontal structure of layered architectures are the main cause of aspect scattering. The introduction of vertical structuring elements therefore seems to be a key to overcome the problems of layered protocol software.

Acknowledgments

This work is sponsered by Siemens ZT IK2, Munich.

REFERENCES

[Buschmann 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *A Systems of Patterns*, John Wiley & Sons, 1996.

[Hüni 95] H. Hüni, R. Johnson and R. Engel, "A Framework for Network Protocol Software", *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*, ACM Press, 1995.

[Hutchinson 91] N. Hutchinson and L. Peterson, "The x-kernel: an architecture for implementing network protocols", *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[POST 81] J. B. Postel, "User Datagram Protocol", *Internet Request for Comments*, RFC 768, August 1981.

[Postel 81] J. Postel, "Transmission Control Protocol", *Internet Request for Comments*, RFC 793, September 1981.

[Unix 90] Unix, "STREAMS Programmer's Guide", *Unix System V Release 4*, 1990.

[ZIMM 80] H. Zimmerman, "OSI reference model-the ISO model of architecture for open systems interconnection", *IEEE Transactions on Communications*, COM-28(4), April 1980.