A Component-Based Architecture For Software Communication Systems

Matthias Jung, Ernst W. Biersack
Institut Eurecom
2229 Route des Crêtes, 06904 Sophia-Antipolis, France
E-Mail: {jung,erbi}@eurecom.fr

In the Proceedings of IEEE ECBS 2000, Edinburgh, Schottland

Abstract

We examine the usefulness of component-based software-engineering for the implementation of software communication systems. We present an architecture that allows to divide protocol software into fully de-coupled components that can be plugged together using visual builder tools to rapidly prototype flexible, robust, and application-tailored communication protocols. We show the feasibility of component-based protocol engineering by demonstrating how a simple transport protocol was realized. A discussion about advantages and impacts concludes this paper.

1. Introduction

While a general purpose transport protocol such as TCP has been for many years the protocol of choice for popular applications like *telnet* or *ftp*, first problems with the use of TCP arose with the success of http [25]. For new applications like audio- and video-streaming, TCP is completely unsuited. We belief that the popularity of the world wide web and new technologies like Sun's Jini (TM) [16] will produce a number of distributed applications with very specialized and diversified communication needs that can not be met in an optimum manner by a general purpose protocol like TCP.

Since protocol implementation from scratch is expensive and error-prone, it is important to structure protocols in a way that protocol elements can be re-used in different contexts and adapted to the special needs of an application. Additionally, protocol implementations should be prepared for change and hence easy to extend and maintain.

The recently emerged software-engineering paradigm called *Component-based development* (CBD) – with Java Beans (TM) and Active-X (TM) [24] as its most prominent representatives – promises a new dimension of re-usability and rapid prototyping of software compared to the simpler object-oriented approach. CBD fosters the division of software into *components* that can be easily configured and plugged together using visual builder tools or mark-up languages to create new applications. In this paper, we will examine if and how component-based software engineering can be useful to implement network protocols.

The rest of the paper is structured as follows. In Section 2, we outline the characteristics of component-based software and present the Java-Beans component model [20]. In Section 3, we revise the functionality of a typical end-to-end protocol and show how a protocol can be mapped to independent components. In Section 4, we demonstrate the feasibility and usefulness of component-based software-engineering for protocol implementation based on a framework prototype with Java Beans. Section 5 reviews related work. A summary closes this paper.

2. Component-based Software Engineering

2.1. Definition

D'Souza et al. [14] define component-based development (CBD) as *an approach to software development in which all artifacts – from executable code to interface specifications, architectures, and business models; and scaling from complete applications and systems down to small parts – can be built by assembling, adapting, and wiring together existing components into a variety of configurations.*

A component is usually characterized by the following attributes:

- Components are largely de-coupled; they can be independently developed and delivered
- Components have explicit and well-specified interfaces for the services they provide

- Components have explicit and well-specified interfaces for services it expects from other components
- Components can be customized and composed with other Components without modification of code

CBD is often confused with object-oriented development (OOD). This stems from the fact that OOD can be seen as the most adapted implementation technique to CBD. In contrary to an object, a component provides a richer range of intercommunication mechanisms, a higher degree of re-use and adaptability, and usually a larger granularity. A component may comprise one or more objects.

2.2. The Java-Beans Component-Model

Java-Beans [20] is the component model provided by Sun. Java Beans is written in the Java programming language [32] and hence profits from Java's machine-independence and portability.

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

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

Any tool that is used to configure and wire the components together can identify properties, events, and behaviour of a bean by analyzing its Java code. A Java class that intends to be Java Beans compliant has to follow certain naming conventions, e.g. properties are identified by method names that start with `set` or `get`. For more details see the Java Beans specification [20].

2.3. Visual Builder Tools

Components can be configured and wired by either using mark-up languages or – the more popular approach – programming environments that allow visual specification, representation, and configuration of components. Such a tool is commonly referred to as **visual builder tool**. The visual builder tool we are using for visual implementation is *Visual-Age for Java* [11] by IBM.

To build a program visually, already existing beans are arranged on a screen called *visual builder window*. Beans are selected either by symbol or by name. The visual builder then analyzes the selected bean to identify properties, events, and methods. Figure 1 shows how properties can be modified in a window called *property editor*.

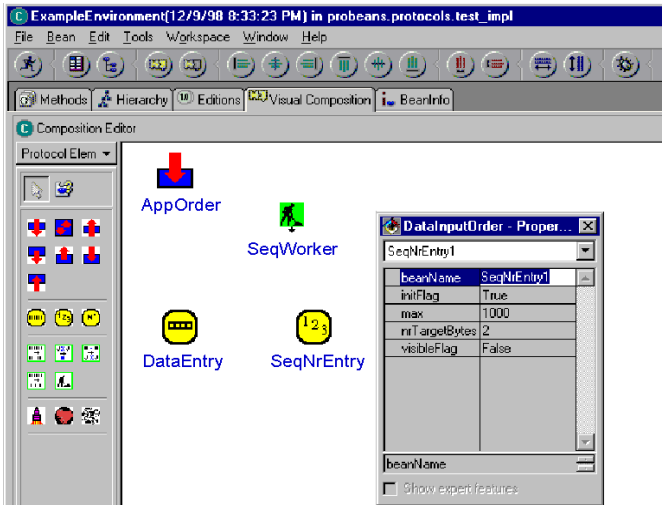


Figure 1. Property configuration (screen-shot using Visual Age)

To establish event-connections between beans, first an event source bean must be selected. A window opens to show all possible events raised by that bean. After selection of the event, the target bean is selected. A window indicates possible behaviour by showing all accessible methods (see Figure 2). If the target method needs one or more parameters (beside the event object itself), these must be attached to the event connection. Possible parameters are properties of any bean in the visual builder window. At run-time, every time the source bean fires the specified event, the method of the target bean is called with the right parameters.

After the design process is terminated, the builder tool creates code based on the information specified. That means, it writes the values of the properties, creates methods to assemble all specified beans, and creates methods to represent the event connections.

3. An Architecture for Component-based Protocol Structuring

3.1. What does an End-to-End Protocol?

The following sequence of processing steps can be observed in a typical end-to-end protocol. An application gives data to a protocol instance, which adds header information to the data, performs computing operations, and changes internal state. After processing the data is sent over

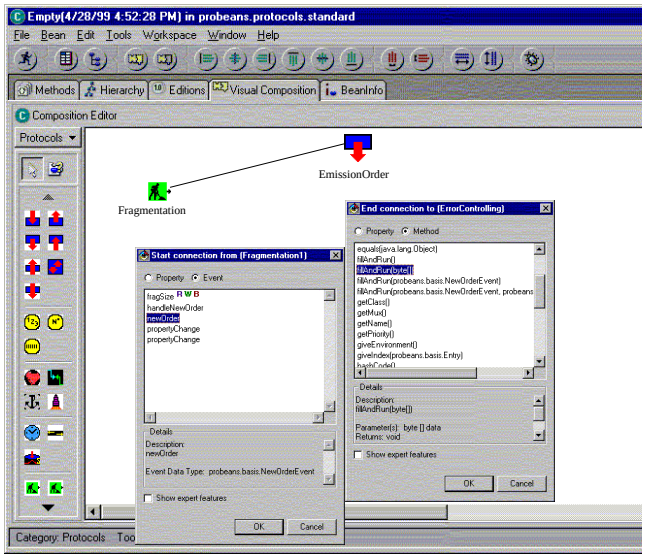


Figure 2. Building event connections (screen-shot using Visual Age)

the network to the communication partner, where the incoming message is parsed and de-multiplexed to the right protocol instance. When the message is processed, the header information is stripped, and the state information is updated, the payload data is delivered to the attached application. An incoming message may trigger an acknowledgment to be sent, hence a protocol instance may also create new messages. Message processing may include checksumming/checksum verification, sequencing/re-ordering, encryption/decryption, fragmentation, etc. State information may include flow-control window-sizes, connection/session state, buffers, information about sequence numbers, etc. Other typical protocol functions are the provision of timers and handling of time-outs.

We resume the identified main tasks of an end-to-end protocol:

- Provision of interfaces for the application
- Processing of messages in output-direction - normally adding header information
- Provision of interfaces to the network
- De-multiplexing of incoming messages
- Processing of messages in input-direction - normally parsing and stripping header information
- Creation of new messages

3.2. How to map protocols to components?

Our main goal is **flexibility** referred to as the possibility to build a variety of protocols out of existing components. Granularity and interface design are the most important issues to achieve flexibility. Traditional layered architectures (as in TCP/IP) follow an extremely coarse-grained approach. As a result, layers are software blocks of high complexity, but can be connected in a simple way to other layers. We will follow a *fine-grained* structure that allows us to combine components in a flexible manner.

Components must be configurable and completely (not only partially) re-usable in different contexts. **Re-usability** requires that components are largely de-coupled from each other, i.e. they can work without any knowledge about the type of other components they interact with. An important design goal is hence to provide autonomy to components and minimize context based knowledge in the components.

In the following, we present an architecture and its design elements that map protocol functionality to de-coupled components. We will then discuss how our approach provides flexibility and re-usability.

3.3. Design elements

An **entry** is a component that encapsulates message header information (like sequence number, window-size, data-payload). The information is represented in two forms: as a byte array (the raw data as it is written to the network) and as an object of an predefined type (to allow comfortable access during processing). Every entry type can be configured **visible** to indicate that it should be transformed into bytes after processing. Entries need to be configured visible when their data is written to the network or given to the application. An entry type can be configured **initializable** to indicate that it should be filled with data *before* processing starts, i.e. with data from network or application. Entries can be both, initializable and visible. The general case for processing in output direction is that all entries are visible and only the data payload initializable. The general case for processing in input direction is that all entries are initializable and only the data payload visible.

A **worker** is a component that encapsulates a functional step during protocol processing (like checksumming, re-ordering, fragmentation). To de-couple the format of a message from its processing, a worker operates directly on entries and never on the message itself. It hence does not perform message parsing and does not need to know where its relevant information can be found.

A worker defines a standard method to accept the data to be processed. In contrary to a classical layer, a worker provides processing either in output- or in input-direction to

enhance re-usability. Besides the standard data processing interface, a worker usually defines a set of properties, events, and public methods to communicate with other components.

An **order** is a component that defines a data path (or thread) through a protocol. It associates a protocol graph (i.e. an ordered set of workers) with its header data (i.e. an ordered set of entries). It has three main responsibilities. Firstly, the coordination of protocol processing by parsing data, mapping bytes to entries, and calling workers with their relevant entries. Secondly, the decoupling of message creation from message processing by reifying the data paths of a protocol. For instance, a worker whose task it is to acknowledge incoming data messages, does not need to know the data format of the message it creates nor how this message will be processed. It just needs to signal an event, which activates an (acknowledgement) order to perform the necessary steps. The third responsibility is the provision of interfaces to application and network.

Different order types can be distinguished based on their interfaces to application and network. An order that is initialized with application data is referred to as **acceptance**, an order initialized with network data is referred to as **reception**. These two order types are parsing the data by considering all their entries configured initializable. An order that delivers processed data to the application is referred to as **delivery**, an order that writes processed data to the network is referred to as **emission**. These two order types transform all entries configured visible into byte arrays before delivery or emission. Orders that are created by other orders and end up without delivering or emitting are called **internal orders**. Emission and reception orders must specify information for multiplexing and de-multiplexing, respectively. Delivery and acceptance orders provide application interfaces for reading and writing, respectively. We name orders that are both acceptable and emittable **output orders**, and orders that are both receptable and deliverable **input orders**.

A protocol **environment** replaces the notion of a protocol stack. Instead of comprising different layers, a protocol environment integrates all end-to-end protocol functionality a particular distributed application needs. Technically, an environment is a component that provides an interface for protocol implementors to register orders. It initializes and deactivates orders and provides coordination between the various components.

Anchor: One typical protocol task is message multiplexing (mapping messages from different sources to one sink) and de-multiplexing (distributing different messages from one source to different sinks). This may comprise protocol (de)-multiplexing (based on the protocol type), session (de-)multiplexing (based on the protocol instance), and functional (de)-multiplexing (based on the message/order type). In layered architectures, (de)-multiplexing is present in every layer and defines the data path of a message.

In order to avoid that the main units of re-use, i.e. the worker and entry components, are not involved with (de)-multiplexing, we concentrate it outside the protocol environment in an extra design element called **anchor**. An environment that wants to work on top of an anchor needs to be registered. The anchor obtains all de-multiplexing information of the various reception orders from the registering environment and puts them into a hash-table. Incoming messages can then be de-multiplexed directly to the respective order. That is, protocol-, session-, and order de-multiplexing are performed within a single hash-lookup without compromising the re-usability of other components.

Events are the key mechanism to hide implementation details of a component from another component it interacts with and hence crucial to provide re-usability and flexibility. In our architecture, we provide two types of standard events. An **order-creation event** signals the intention of creating a new order. A **notification-event** signals the intention of notifying other components about change of state. Both events are normally raised by worker components and can be combined with any method and parameterized with any property of any component. However, the sink of an order-creation event should be an order type, which triggers processing of a new message (e.g. an incoming data message triggers an acknowledgment). The sink of a notification-event is normally another worker (e.g. an incoming acknowledgment signals that retransmission buffer space can be freed).

A **SAP** (Service Access Point) represents an access point to the network (corresponding to the notion of a service access point of the ISO/OSI layer model). The idea of this component is to de-couple protocols from dependencies of the underlying network. A SAP can implement Ethernet access as well as a UDP socket. A **AAI** (application access interface) represent an access point to the application. We distinguish read AAIs (for delivery orders) and write AAIs (for acceptance orders).

An overview of the whole architecture and the relations between the design elements can be seen in Figure 3.

3.4. Summary

We shortly resume how our architecture provides re-usability and flexibility and therefore perfectly fits the component based design paradigm.

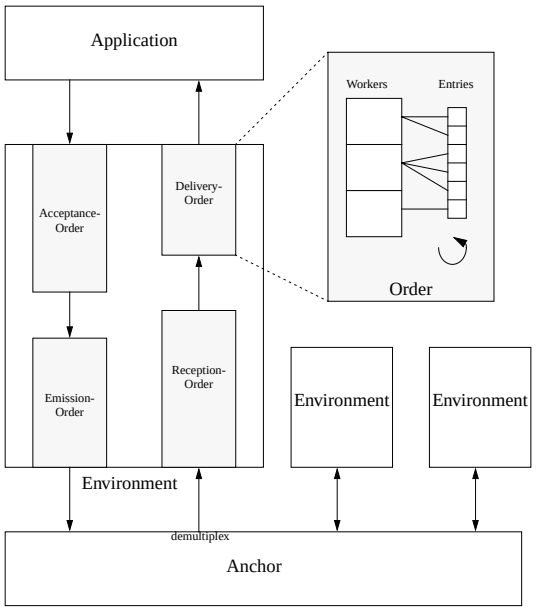


Figure 3. Architecture of the Component-Based Communication System

- De-multiplexing (which requires the knowledge of protocol specific information and where the relevant information can be found within a packet) is removed from the processing path of a message and concentrated in the network anchor. A worker hence does not need to know which worker will next process a message.
- Parsing of messages is removed from the processing path of a message and put into orders. A worker hence does not need to know where in the message the relevant information can be found. It obtains the relevant information as entries.
- The Java Beans event mechanism allows to combine arbitrary components. A worker hence does not need to know the type of the workers he interacts with.
- Input- and output processing are separated in different workers. This enables to de-couple sender and receiver functions.
- For each message, its data path is clearly defined by orders. When a worker creates a new message, it does not need to know anything about format and further processing of this message.
- Typical operations on header fields can be easily re-used, since they are encapsulated in entries (e.g. sequence-numbers). Changing the header format (e.g. changing from a 2-byte to a 4-byte sequence-number) requires just a configuration of the respective property and does not impact any other part of the protocol code.

Java Beans allows components to be represented by icons. In Figure 4 one can see the icons for the protocol components described above.

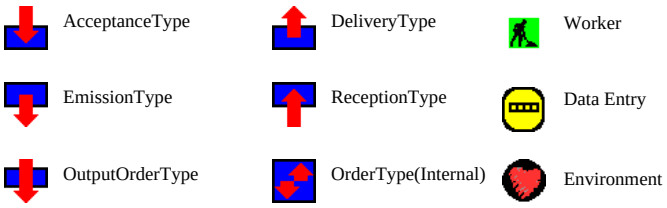


Figure 4. Java Beans Symbols for our Protocol Components

4. A Case-Study

To better understand our approach, we first describe a simple example protocol that ensures uni-directional reliable transfer of data. We then show how this protocol is structured following the concept exposed in the last section. The protocol is instantiated and started after a connection protocol established a connection between a sender application and a remote receiver application. We skip the connection protocol and concentrate at the data transfer phase.

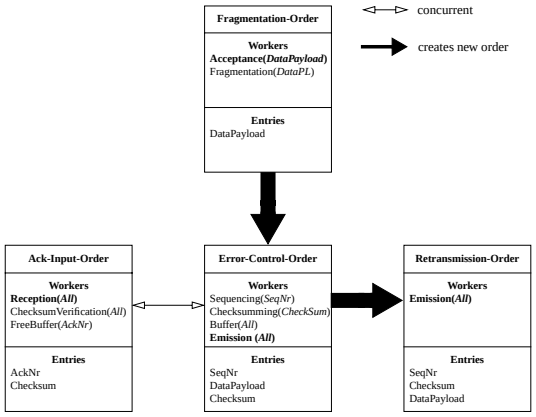
4.1. An example protocol

After a connection is established, the sender application writes a byte stream in form of segments (byte-arrays) to its protocol interface. When the segment exceeds a predefined length (maximum segment length), the segment is fragmented in smaller segments. Each of these segments gets a sequence number and a checksum, is copied to a buffer, and written to the network. A timer is maintained for the oldest segment in the retransmission buffer. If this timer expires, the oldest segment in the queue is retransmitted (selective retransmission strategy).

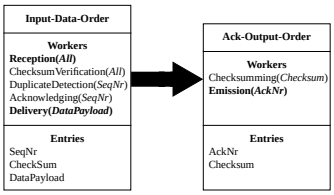
On the receiver side, the sequence number of the incoming segment is used to selectively acknowledge this segment. There are no negative or accumulative acknowledgements. If the data is no duplicate, it is delivered to the receiver application. Note that our protocol does not assure in-order delivery.

On the sender side, incoming acknowledgements free the buffer holding the segment with the sequence number specified in the acknowledgment and restart the transmission timer (only if there is still data in the buffer to be acknowledged).

4.2. Identification of orders in the example protocol



(a) Orders of the sender



(b) Orders of the receiver

Figure 5. Identified Orders of the Example Protocol

Error-Control Order: When the size of the original application data does not exceed the maximum segment size, it is given a sequence number and a checksum. It will then be buffered and sent to the network (while the retransmission timer may be started if this segment is the first to be buffered). This functionality can be done within one thread. The corresponding order is referred to as *error-control-order*. The header information needed for this order is a sequence number, a checksum field and the data payload from the application.

The *error-control-order* ends up with sending data to the network. It is therefore of type *emittable*. It consists of three workers: the *sequencing worker* assigns sequence numbers to each data that comes along. It contains a counter that starts with a value represented by a beans property. The second worker calculates a checksum on an arbitrary set of data and writes this value into an entry representing an integer. The third worker called *retransmission worker* operates on a sequence number and an array of arbitrary data. It buffers both and maintains a retransmission timer. Upon expiry of the timer, the retransmission worker fires a *NewOrderEvent* containing sequence number and the other data buffered. The needed entries are hence sequence number, checksum value, and data payload. Only the data payload is marked *initializable*, since sequence number and checksum value are calculated while the order is processed. All entries are marked *visible* since they are all written to the network.

Fragmentation Order: When the application data exceeds the defined maximum segment size, it must be fragmented in smaller segments. The process of fragmentation can be done within one thread. The corresponding order is referred to as *fragmentation-order*. The data processed by this order is the original application data, the result is a number of segments with a size smaller than the maximum segment size.

The *fragmentation-order* processes data from the application and is therefore of type *acceptable*. It contains one worker *fragmentation-worker* which takes an array of bytes as input and throws an event *NewOrderEvent* for each new segment resulting from the fragmentation process. The maximum segment size is specified as a configurable beans property of the *fragmentation-worker*. The only entry needed represents the array of byte which can be of variable length. This entry is marked *initializable* since it is filled with data when the order is initialized. It does not need to be marked *visible* because the data is not delivered or emitted. Fragmentation and error-control can not be done within one thread and have hence different orders since fragmentation results in several pieces of data that all need individual processing (sequencing, buffering).

Retransmission Order: When the retransmission timer expires, the oldest data in the queue (including sequence number, checksum, and data payload) is read from the buffer and sent to the network. This operation requires an own thread and hence an own order that we refer to as *retransmission-order*.

The *retransmission-order* is of type emittable and does not define any workers. It just takes the sequence number entry, the checksum entry, and the data payload (all are visible) from the *error-control-order* and writes it to the network.

Input-Data Order: A packet sent by the *error-control-order* or the *retransmission-order* contains an predefined de-multiplex identifier used to associate the incoming data with a thread of the *input-data-order* which serves to deliver the data to the application. Before delivery, the checksum must be verified, a check for duplicate sequence number is made, and an acknowledgment is requested. If the checksum is not correct or the data was identified as duplicate, the data is not used anymore. Otherwise the data payload is delivered to the application.

The *input-data-order* gets its data from the network (*receptable*) and delivers it to the application (*deliverable*).

It is therefore of type *input-order*. At first an *ack-out-order* is generated, then a *checksum-verification-worker* checks if the value of the checksum field corresponds with the value calculated over the whole data. A *duplicate-check-worker* finally assures that data is delivered only once to the application. When checksum and duplicate check did not lead to dropping the data, the payload is delivered to the application. All entry fields – sequence number, checksum value, and data payload – are marked *initializable*. The data entry is additionally marked *visible* since it is delivered to the application.

Ack-Output Order: There are two orders not yet mentioned needed for acknowledgment handling. One order called *ack-out-order* sends a message to the network that contains the sequence number of the acknowledged data. The *ack-output-order* is of type *emittable* and defines an acknowledgement number as its only entry. Besides checksum verification it contains no workers.

Ack-Input Order: The last order needed is called *ack-in-order* and notifies the retransmission-buffer that the segment with the sequence number of the acknowledgment can be deleted from the buffer. The *ack-input-order* is of type *receptable*. It comprises two workers, one to verify the checksum, the other to notify the retransmission worker.

Figure 5 shows the identified orders, their workers, and the entries defined. The big arrows indicates that an order creates another one. The small double-sides arrows indicate that two orders access instances of the same variable, e.g. the *ack-in-order* accesses the retransmission buffer to free data that the *error-control-order* wrote to.

4.3. Visual Composition Process

Once workers and entries are implemented, our protocol is built without writing any additional line of code. We just select for each identified order, entry, and worker the corresponding symbol and put it on the screen. We connect workers and entries to their orders (a special method implemented by the order bean allows to register workers together with their entries to express a parameter relationship), and orders to their environment. We configure the properties of the workers (like the maximum segment size for the *fragmentation worker*), entries (like the range of sequence numbers, the *initializable* and *visible* flags for any entry), and orders (like the de-multiplex information for reception orders). We specify *creates-order* relationships by connecting the *NewOrderEvent* of a worker with the order it should create (like the *fragmentation-worker* creates the *error-control-order*). We specify any other relation-ship between workers (e.g. state-updates) or between entries and workers (like the sequence number entry of *acknowledgment-order* is given to the *retransmission worker* to free the retransmission buffer). In Figure 6, we depicted a screen-shot after building the sender part of the example protocol. It shows all protocol components and their relationships.

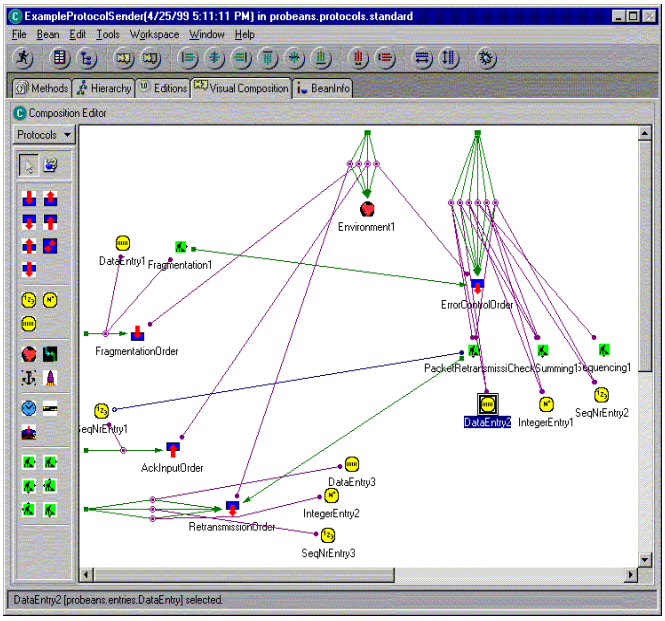


Figure 6. Example protocol (sender) built with Visual Age (screen-shot using Visual Age)

Running protocols within our framework, i.e. initialization of the environment and the anchor, the registration of orders, entries, and workers, the allocation of threads, filling entries with data, executing orders, is transparent to the visual composer, i.e. part of the implemented framework. It is out of the scope of this paper to describe the details of the runtime process of our framework.

4.4. First Experiences

Rapid Prototyping and Testing: Dividing protocols in small modules already facilitates implementation significantly. The process of configuring and combining various components to a working application is a process of minutes. Due to the guidance of the visual builder tool, the generated code is very robust. The clear structure largely simplifies testing and debugging.

Re-usability: All protocol components are 100% reusable without any modification of code. However, while entries can be applied in almost all protocol contexts, the applicability of workers is often limited to certain protocol families. The re-usability of whole orders is a rare case, the re-usability of complete protocol layers is not yet supported within our framework.

Compatibility: Our structuring approach may conflict with specifications of existing protocols (e.g. centralized de-multiplexing, multiple header formats). Since we intent to provide a tool to rapidly implement and test new protocols instead of re-implementing existing ones, we do not consider compatibility as a major issue.

5. Related Work

Protocol implementation is a well explored field and a high number of papers have addressed areas related to this work. The idea to give the application more control over the protocols it uses, was first expressed by Clark and Tennenhouse and called Application Level Framing (ALF) [9]. Moving protocol code into user space is one step in that direction. Experiences with user space protocol implementations are reported in [34], [6], [15]. Our work consequently adopts the paradigm of ALF and user space protocol implementation.

The idea of replacing general purpose protocols by application tailored protocols requires ease of implementation and rapid prototyping of new protocols. Protocol frameworks are an important tool to assist protocol implementation by providing implementation and runtime support. There exist a high number of protocol frameworks, each of which has a different focus. The most prominent one is the X-Kernel [18] residing in the operating system kernel, which allows to connect protocol layers by a standard interface. The acceptance of the object-oriented programming paradigm lead to a number of object-oriented protocol frameworks ([17], [4]). Our work comprises also a framework to build protocols. However, it widely differs from the cited frameworks not only in the software engineering technology used, but also in its architectural concept, the granularity, and the structure it imposes.

The majority of all these protocol frameworks follow rather coarse-grained, layered structuring approaches. The advantages of fine-grained structuring and modularization – higher flexibility and improved re-usability without serious performance degradation – were first exposed by O'Malley and Peterson [27]. DaCaPo [28] and ADAPTIVE [30] demonstrate the higher flexibility of fine-grained modularity by featuring dynamic configuration and assembly of protocols with classified requirements. Bhatti [2] overcomes problems of the X-Kernel environment to implement fine-grained fault-tolerance multicast applications. The Java protocol framework JChannels [21] uses a fine-grained, object-oriented approach to facilitate protocol maintenance and modification. Fine-grained structuring is one requirement to achieve the main goal of our work, i.e. to build the maximum number of communication software out of a set of re-usable and configurable components. In this sense, our work is close to the cited work that follows fine-grained structuring. However, it widely differs in its structuring concept, the programming comfort, and the degree of re-usability of protocol functions.

Work related with the structure of protocol software goes back to the early eighties. Layering has been the main conceptual approach to structure protocols. The advantage of layering is to reduce complexity of communication systems by hiding information between different layers. However, a number of problems with regard to layering have been identified such as inflexibility [35], inefficiency [33], and even unexpected side-effects [12].

A lot of work tackles the efficiency problems of layering. Clark [7] and Cooper [10] propose to carefully break up the borders and allow adjacent layers to exchange control information. Clark [8] and Atkins [1] suggested to apply vertical process models to protocol software in order to reduce context switches. Renesse [36] shows that techniques such as header prediction, packet filtering, and message packing can significantly improve latency. Mosberger and Peterson unified optimization mechanisms as ILP [5] [13], fbuf [13], or packet classifiers [23] [19] in the abstraction of a *data path* and implemented dynamic path creation in the Scout operating system [26].

Our work does not intend to mask performance problems of layered architectures, but to replace layering in end-systems by an architecture that supports complete re-use, fine-grained structuring, and high flexibility. The idea behind our design element *order* is related to Mosbergers [26] definition of a *data path*, i.e. a logical channel or a common sequence of instructions through a complex system. However, while in Scout paths are used to apply a vertical process model and to optimize inter-layer communication, we use orders to structure protocols itself with the goal of supporting rapid and visual prototyping of new protocol implementations. That is, we consider how protocols should be structured to be implemented in a fast, flexible manner instead of considering how protocols should be implemented to optimize performance.

We don't know of any work concerned about component-based programming for protocol implementation. There is at least little work related to component-based software engineering in system design. Kon and others [22] use component-based programming to provide high configurability of a flexible operating system (2K). Singhai [31] demonstrates the usefulness of component-based software engineering for the development of middle-ware systems.

The concept of de-multiplexed architectures represented by our element *anchor* was first expressed by Tennenhouse [33] and was implemented in [29] [3].

6. Summary

The goal of this paper is to show that component-based software technology is an excellent means to rapidly implement application-tailored end-to-end protocols out of fully re-usable and configurable software components. We proposed a structuring approach to map protocols to components, which provides fine granularity and complete decoupling of protocol functions and message headers. We implemented a runtime-system that follows our structuring approach to test and run component-based protocol software.

Based on a simple transport protocol, we showed the usefulness, power, and simplicity of the implementation process. We could see that even visual programming of complex software as network protocols is feasible and provides a new dimension in rapid prototyping and programming comfort.

Acknowledgment

This work is sponsored by ZT IK2, Siemens, München, Germany.
Special thanks to Sergio Loureiro for the fruitful discussions that helped to clarify my ideas.

References

[1] M. Atkins. Experiments in sr with different upcall program structures. *ACM Transactions on Computer Systems*, pages 365–392, 1988.

[2] N. Bhatti and R. Schlichting. A system for constructing high-level protocols. In *ACM SIGCOMM Symposium*, pages 138–150, Aug. 1995.

[3] E. W. Biersack and E. Rütsche. Demultiplexing on the atm adapter: Experiments with internet protocols in user space. *Journal on High Speed Networks*, 5(2):193–202, May 1996.

[4] S. Boecking, V. Seidel, and P. Vindeby. Channels. a run-time system for multimedia protocols. 1995.

[5] T. Braun and C. Diot. Protocol implementation using ilp. In *Proceedings of ACM SIGCOMM'95*, pages 151–161, 1995.

[6] T. Braun, C. Diot, A. Hoglander, and V. Roca. An experimental user level implementation of tcp. Technical report, INRIA, 1995.

[7] D. Clark. Modularity and efficiency in protocol implementation. Request for Comments (Informational) RFC 817, Internet Engineering Task Force, July 1982.

[8] D. D. Clark. The structuring of systems using upcalls. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, Oakland, CA, December 1985.

[9] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. ACM SIGCOMM '90*, pages 200–8, Philadelphia, PA, September 1990.

[10] G. H. Cooper. An argument for soft layering of protocols. Technical Report MIT/LCS/TR-300, MIT, May 1983.

[11] I. Corp. *Programming with VisualAge for Java Version 2*. Number SG24-5264-00. IBM Redbook, 1998.

[12] J. Crowcroft, I. Wakeman, and Z. Wang. Is layering harmful? *IEEE Network*, 6(1), Jan. 1992.

[13] P. Druschel, M. Abbot, M. Pagels, and L. Peterson. Network subsystem design: A case for an integrated data path. *to be Published in IEEE network 7'93*, 1993.

[14] D'Souza, D. Francis, Wills, and A. Cameron. *Objects, components and frameworks with UML : the Catalysis approach*. Addison-Wesley, 1998.

[15] A. Edwards and A. Muir. Experiences implementing a high performance tcp in user space. In *Proceedings of ACM SIGCOMM*, Boston, MA, Oct. 1995.

[16] W. Edwards. *Core JINI*. Sun Microsystems Press, 1999.

[17] H. Hüni, R. Johnson, and R. Engel. A framework for network protocol software. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*. ACM Press, 1995.

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

[19] V. Jacobson. 4bsd tcp header prediction. *Computer Communication Review*, 20(2):13–15, Apr. 1990.

[20] JavaSoft. *Java Beans 1.0 API specification*, October 1996.

[21] M. Jung, E. Biersack, and A. Pilger. Implementing network protocols in java - a framework for rapid prototyping. In *Proceedings of ICEIS'99, Portugal*, Mar. 1999.

[22] F. Kon, A. Singhai, R. H. Campbell, D. Carvalho, R. Moore, and F. J. Ballesteros. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[23] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX*, San Diego, CA, Jan. 1993.

[24] S. Microsystems. Component-based software with javabeans and activex. White Paper, Oct. 1996.

[25] J. Mogul. The case for persistent-connection http. In *Proceedings of SIGCOMM'95*, pages 299–314, sep 1995.

[26] D. Mosberger and L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of OSDI*, pages 153–168, Oct. 1996.

[27] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[28] T. Plagemann, P. B., M. Vogt, and W. T. Modules as building blocks for protocol configuration. In *Proceedings of the international Conference on Network Protocols (ICNP-93)*, Sept. 1993.

[29] V. Roca, T. Braun, and C. Diot. Demultiplexed architectures: a solution for efficient streams based communication stacks. *IEEE Networks Magazine*, June 1997.

[30] D. Schmidt, D. Box, and T. Suda. Adaptive - a dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency Practice and Experience*, 5(4), June 1993.

[31] A. Singhai. *Quarterware: A Middleware Toolkit of Software Risc Components*. PhD thesis, University of Illinois, 1999.

[32] Sun Microsystems. The java virtual machine specification. Technical report, 1995.

[33] D. L. Tennenhouse. Layered multiplexing considered harmful. In H. Rudin and R. Williamson, editors, *Proc. IFIP Workshop on Protocols for High-Speed Networks*, pages 143–148, Zurich, Switzerland, May 1989. North-Holland Publ., Amsterdam, The Netherlands.

[34] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocol at user level. *IEEE/ACM Transaction on Networking*, 1(5):554–565, Oct. 1993.

[35] C. Tschudin. Flexible protocol stacks. In *Proceedings of ACM SIGCOMM*, Zurich, Switzerland, Oct. 1991.

[36] R. van Renesse. Masking the overhead of protocol layering. In *Proceedings of ACM Sigcomm*, Sept. 1996.