Order-Worker-Entry: A System of Patterns to Structure Communication Protocol Software

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

EuroPLoP 2000, Bad Irsee, Germany

Abstract

We present a system of architectural design patterns for the implementation of flexible, extensible, and maintainable application tailored network protocol software in end-systems. The pattern system mainly follows a vertical structuring approach. *Outsourced De-Multiplexing* reduces cross-talk between different protocol sessions. *Data-Path Reification* assures service extensibility and quality of service control for protocols. *Data Path Partitioning* assures the re-usability of protocol components and allows for easy modification and configuration. *Data Path Classification* enables application and network to flexibly interact with a protocol environment without compromising the protocol's autonomy.

I. Introduction

The popularity of the Internet lead to many new applications with diversified communication service requirements that are not optimally met by the standard transport protocol TCP [1]. A common technique is therefore to program new, application tailored, proprietary protocols and integrate them into the application code. These protocols often build upon UDP [2], which is a lightweight transport protocol that only performs demultiplexing and assures that a packet delivered is not corrupted.

Implementing and testing new protocols is reputed to be extremely cumbersome and time-intensive, maintenance and modification to be difficult and error-prone. We believe these difficulties are mainly due to the surprising fact that most protocols are still implemented from scratch and in an ad-hoc manner. Dividing protocol software into fine-grained, re-usable, configurable components is hence a very obvious (but not always trivial) solution to reduce development costs and improve the quality of protocol software. In this paper, we present a catalogue of design patterns that help make protocol software more flexible and re-usable. We also hope to put a software engineering spin in a domain that is often considered purely dominated by efficiency concerns.

II. Motivation and Overall Context

A variety of distributed applications have requirements that can not be met by general purpose protocols (e.g. multicast protocols, audio- and video-streaming, applications with partial-order requirements) and hence need to implement their own specialized protocols. Our intention is to structure end-to-end communication protocol software in a way that it can be tailored easily to the needs of the application that uses it. In other words, we follow the question **how to make end-to-end protocol implementations and design more flexible, easier to adapt and the parts of it easier to reuse across different protocols**.

On the one hand, different applications have diversified communication service requirements and thus need different protocols; at the other hand, they must share system and network resources. The first problem to solve is therefore how applications can be de-coupled from each other while efficiently making use of the common resources. We tackle this aspect in Section IV with the *Outsourced De-Multiplexing* pattern.

The applications' requirements evolve over time. Protocol software should therefore be open for adding or removing services. It is furthermore important that each service meets individual QoS requirements. We tackle these aspects in Section V with the *Data-Path Reification* pattern.

Re-use, configurability, and granularity of protocol components are important issues to allow the rapid development and construction of new protocols. How fine-grained structuring is applied to protocol software to assure easy modification, extension, and autonomy of all protocol components is presented in Section VI in the *Data-Path Partitioning* pattern.

From the perspective of an distributed application programmer the programming interface to communication protocol services is of great importance. The protocol software should make (almost) no assumptions about the application and the underlying network to be not affected by any changes of those. We present in Section VII the *Data-Path Classification* pattern to balance the forces described above.

The four patterns we present form a *system of patterns* to structure higher-level end-to-end protocols in a way that new protocols can be constructed from re-usable modules and tailored easily to the needs of an application. These protocols are run within the application that uses it (in contrary to a typical general purpose protocol like TCP, which is part of the operation system).

Note that the models and the pseudo-code we describe during this paper are not ready for production ”prime time”. They serve to improve the understandability of the patterns we describe, rather than giving guidance how to concretely implement them.

III. A Related Pattern – Layers

The best known design pattern for communications protocols is the *Layers* pattern described in [3]. A layer communicates exclusively with its adjacent layers via a simple interface to exchange messages. In output direction, each layer adds its data information to the current message, which is used and removed by its remote peer layer in input direction. A layer normally comprises a set of related protocol functions (like a transport-layer error-control, flow-control, congestion-control).

Layering is an easy and well-understood design pattern that localizes dependencies of different parts of a system and allows to exchange or re-use whole building blocks. However, there are also a number of problems associated with layering:

- stream synchronization: layered protocols do not allow for quality of service differentiation for different streams. This is due to the fact that all streams of layer n are multiplexed into a single stream at layer $n - 1$.
- header overhead: this phenomenon arises since different layers do not share header information (e.g. in TCP, acknowledgements are shared by error as for flow-control; putting these two functions in different layers would result in redundant information to be sent)
- processing overhead: a message is traveling each layer of a protocol stack. When the layer finds out that the message is not meant for it, it just forwards it to its adjacent layer (chain-of responsibility pattern). This way, a useless message may be examined several times before it can be thrown away.
- cascades of changing behavior: although layers depend only on neighbour layers, modifications or change of behavior of one layer may involve the whole stack. That is, layers can not be considered generally independent components.

Since efficiency and usefulness of layered systems degrade with the number of layers, we consider layering useful only as a coarse-grained structuring approach to protocol implementations. However, tailoring application protocols requires fine-grained modularity. The design patterns we describe in this paper, can thus be considered as an alternative to the use of *fine-grained* layering or as solutions to the problem of how to structure a coarse-grained layer itself.

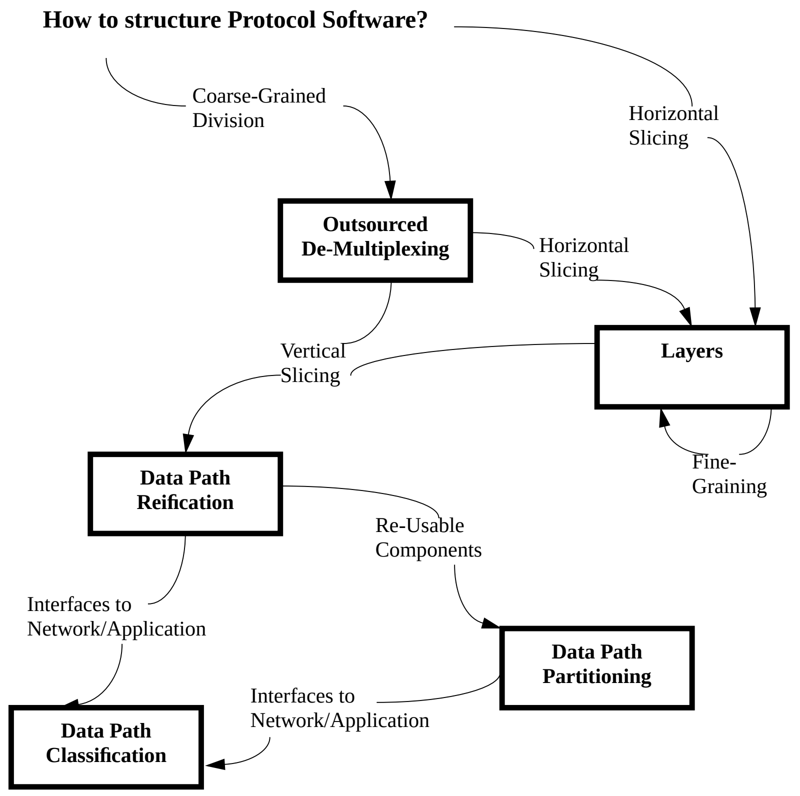


Fig. 1. Design Pattern Relationships

In Figure 1, you can see a map that depicts the relationships among the design patterns we describe. The map also includes the *Layers* pattern to illustrate how layering is related to the design patterns we will describe.

IV. Pattern: Outsourced De-Multiplexing

A. Context

New distributed applications need protocols that are tailored to their communication requirements.

B. Problem

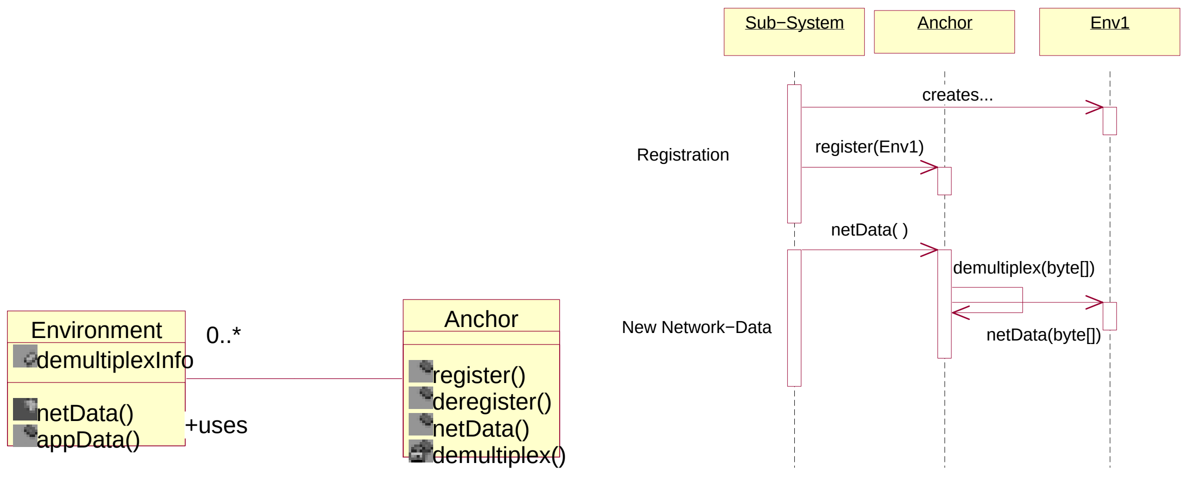
Which architecture is best-suited to allow protocols to be tailored to the needs of its application? How can flexibility and quality of service control of a protocol be maximized?

C. Forces

- different applications must often share network resources
- different applications have different service requirements resulting in different protocols
- multiplexing couples streams of different protocols
- de-multiplexing is performance intensive, but unavoidable

D. Solution

Integrate a protocol session into a single object, which replaces the notion of a protocol stack, and delegate all de-multiplexing to a lower layer.



(a) Class diagram (b) Dynamics

Fig. 2. Outsourced De-Multiplexing

Environment: Instances of this class represent protocol sessions. An environment replaces the notion of a protocol stack but, in contrary to a protocol stack, it does not layer, but integrate all protocol functions (including a state-machine). It provides demultiplexing information, but does not perform demultiplexing itself.

Anchor-Layer: This component can be considered as a protocol layer, the only task of which is de-multiplexing. It therefore maintains a list of all active environments and the associated de-multiplex information.

There are mainly two variants to apply this pattern. Firstly, the anchor-layer can be run in user space and maintains object references of all active environments. In that case, each active environment must be registered with its de-multiplex information. The anchor-layer obtains data from a transport service access point (like a UDP socket), looks in each packet and decides to which environment object the packet is forwarded. This variant is necessary when the underlying transport protocol does not perform de-multiplexing or when several environments would share one *transport* service access point.

The second variant is based upon lower layer de-multiplexing (e.g. TCP, UDP, IP). In this case, the kernel protocol stack acts as anchor-layer, which de-multiplexes incoming data to the specified socket based on the port number. Each environment is associated with its own transport socket, from where it gets directly its data. This variant is not only simpler since it avoids implementing the anchor layer, but also promises better latency since kernel processing is prioritized and highly optimized.

E. Discussion

Besides the advantage of reducing cross-talk between flows of different applications, this pattern simplifies the protocol to be implemented by putting complexity into the anchor layer. The pattern therefore is particularly useful and efficient when de-multiplexing is delegated to the operating system.

As a drawback this pattern increases the use of logical resources (i.e. session state objects)

```
public class Anchor {
  /** a dynamic list of all environments registered */
  Vector allRegisteredEnv=new Vector();
  /** a dynamic list of all demultiplex information of the environments */
  Vector allDemuxInfos=new Vector();
  ...

  /** registration of a new environment */
  public synchronized void register(Environment env) {
    //check if de-multiplex information is unambigous
    if (isOK(env.demuxInfo)) {
      allRegisteredEnv.addElement(env);
      allDemuxInfos.addElement(env.demuxInfo);
    }
  }
  /** deregistration an environment */
  public synchronized void deregister(Environment env) {
    int idx=allRegisteredEnv.indexOf(env);
    if (idx>=0) {//is the environment to be removed really registered?
      allRegisteredEnv.removeElementAt(idx);
      allDemuxInfos.removeElementAt(idx);
    }
  }
  /** arrival of data from the network */
  public void netData(byte[] data) {
    Environment target=demultiplex(data);
    if (target!=null)
      target.netData(data);
  }
  /** internal method to demultiplex data to the right environment */
  private Environment demultiplex(byte[] data) {
    for (int i=0; i<allRegisteredEnv.size(); i++) { //linear check - space for optimization
      DemuxInfo currentDemux=(DemuxInfo)allDemuxInfos.elementAt(i);
      if (currentDemux.matchesWith(data))
         return (Environment)allRegisteredEnv.elementAt(i);
    }
    return null;
  }
  /** checks if the demultiplex information is unambigous */
  private boolean isOK(DemuxInfo di) {
    ...
  }
  ...
}
```

Fig. 3. Pseudo Code for Anchor

compared to a strictly layered model, where lower layer protocol sessions are shared among applications.

When cross-talk between flows of different applications is necessary (like an integrated congestion-manager for all flows of a system) this pattern may rather complicate then simplify the design of the whole system.

F. Known Uses

Roca [4] modified a TCP/IP stack within the Streams [5] environment in the BSD Unix kernel to concentrate de-multiplexing just on top of the network drivers. He demonstrated a performance speed-up, more flexibility and simplicity compared with layered de-multiplexing.

Rütsche [6] ported a kernel TCP/IP implementation into user-space that delegates de-multiplexing to the underlying ATM adapter to allow better control over protocol processing to guarantee the QoS of the networked data.

Braun [7] implemented a user space version of TCP, which lets de-multiplexing reside in the kernel to increase flexibility without unnecessarily compromising performance.

The JChannels environment [8] implements *Outsourced De-Multiplexing* to de-couple application protocols running in the same Java Virtual Machine.

G. Resulting Context

The *Outsourced De-Multiplexing* pattern does not give a solution to the problem how and by whom a new environment instance is created and how the environment is notified about the arrival of network data. These problems are related rather to communication-system design than to protocol design. We therefore reference the interested reader to a set of communication design patterns developed in the ACE environment [9] like Acceptor/Connector or Reactor. What we call *Environment* would be a *ConcreteServiceHandler* in the Acceptor/Connector pattern.

V. Pattern: Data Path Reification

A. Context

While *Outsourced De-Multiplexing* reduces cross-talk between flows of different protocol sessions and allows for application-specific quality of service, it does not tackle the problem how the protocol itself should be structured to be flexible, extensible, and configurable enough to be tailored to the applications needs.

B. Problem

How should communication protocol software be structured to allow for tailoring to the needs of its application, i.e. how can it be implemented efficiently but being open for extensions?

C. Forces

- applications expect different quality of services from protocols
- application requirements evolve over time and new services must be foreseen to be integrated later
- applications have high requirements with regard to latency and throughput
- the thread-model of a protocol is a very important design issue in terms of efficiency and programming comfort; unfortunately, one does not always know which is the best before implementing a prototype (and then it may be a lot of work to change)

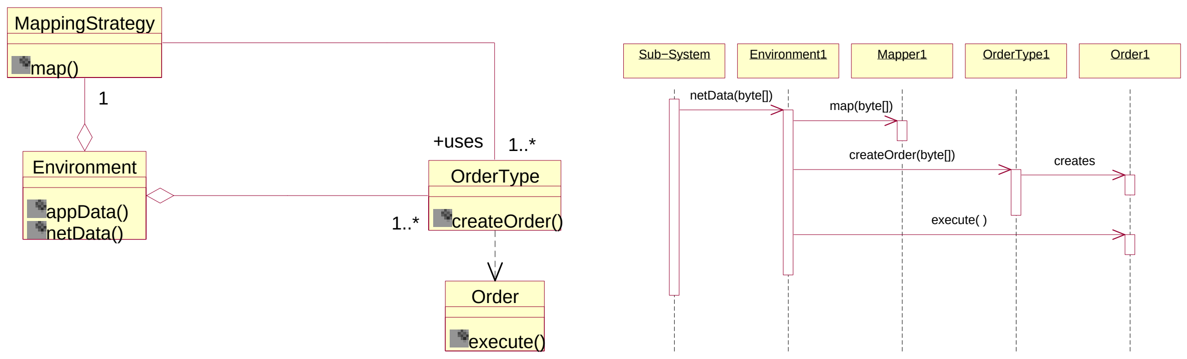
D. Solution

Cut a protocol into vertical slices that reify all data-paths through a protocol stack.

OrderType: an order type represents a data-path through a protocol, i.e. all operations and data manipulations necessary to process a piece of data.

Order: an order is the runtime representation of an order-type, i.e. it performs the execution of a data-path with concrete data. The relation between order-type and order is similar to the relation between code and a running program.

Environment: the environment represents a protocol session (see above). It consists of a set of order-type objects and provides interfaces for incoming data (either from the network or the application). Before an order can be executed, incoming data must be mapped to the right order-type. While for application data this can be done easily by the application itself (see pattern *Data Path Classification* below), network data must be mapped to its order-type based on information carried by the message. The environment delegates this task to a mapping object (see below). It then initializes and executes the created order objects.



(a) Class relations (b) Dynamics

Fig. 4. Data Path Reification

Mapping Strategy: a mapping strategy object maintains references to all possible order-types and maps network data to order-types. Typically, this is done by assigning an identifier to each order and by looking for this identifier in each packet. If only one order type is supported, the mapping strategy simply needs to return this order type.

E. Dynamics

Data coming from the network is given to the environment via `Environment.netData()`. The environment then determines the corresponding order-type object via `MappingStrategy.map()`. It creates a new order object via `OrderType.createOrder()` and executes it via `Order.execute()`. For application data, the application can directly determine the order to be executed when calling `Environment.appData()` and hence no mapping via the mapping-strategy is needed.

F. Discussion

Data Path Reification allows for highly efficient protocol implementations since it supports and integrates a set of high-performance protocol implementation techniques like ILP [10], fast-path processing, discard of unnecessary work, or by-passing of needless functions.

It furthermore makes the dynamics of a protocol visible and thus allows to easily and flexibly map orders to threads, to associate priorities, and to optimize scheduling. Each order type could implement its own thread, could use threads from thread-pools, or delegate processing to a single system thread.

New services can be added in form of new orders, which do not interfere with already existing services, which makes protocols highly extensible, flexible and allows for different QoS within one protocol session.

However, *Data Path Reification* does not inherently support modularity and re-usability of protocol components, particularly layering of complete protocols is rather difficult.

Furthermore, difficulties may arise when a protocol to be implemented is exposed to compatibility constraints that do not match very well with the vertical structure (e.g. TCP consists of several data paths, but a single message header).

G. Known Uses

A data path is a fundamental and natural abstraction and used in many different contexts. Business organizations are structured along work-flows, plans are structured along related activities,

```
public class Environment {
  /** the demultiplex information of the environment */
  DemuxInfo demuxInfo;
  /** the mapping strategy */
  MappingStrategy mapStrat;
  /** array of order-types */
  OrderType[] myOrderType;
  ...

  /** parses and processes data from the network */
  public void netData(byte[] data) {
    OrderType currentOrderType=mapStrat.map(data);
    if (currentOrderType!=null) {
      Order newOrder=currentOrderType.createOrder(data);
      newOrder.execute();
    }
  }
  /** parses and processes data from the application */
  public void appData(OrderType ot, Object[] data) {
    Order newOrder=ot.createOrder(data);
    newOrder.execute();
  }
  ...
}

public interface MappingStrategy {
  OrderType map();
}

public class SimpleMapping implements MappingStrategy {
  /** array of all order-types */
  private OrderType[] allOrderTypes;

  /** maps data to an order by inspecting the first byte of the data */
  public OrderType map(byte[] data) {
    return allOrderTypes[data[0]];
  }
}
```

Fig. 5. Pseudo Code for Environment and Mapping Strategy

the TV program is structured in channels with different emissions. In the context of computer systems, the idea of grouping related tasks that share resources comes in various shapes. The concept of concurrency [11], [12], which is applied in all modern operating systems (e.g. BSD UNIX [13]), abstracts the execution of applications as *processes*. The *thread* concept [**?**], [14], [15] allows to structure application processes along related tasks. Modern languages like Java [16] give explicit language support for threads and concurrency issues to enhance the productivity of development.

The Scout operating system [17] is built around the notion of a *path* as a fundamental structuring technique. Main motivation were efficiency concerns and the integration of several OS mechanisms into a unified infrastructure.

Most TCP versions use a technique called header prediction [18] to map incoming data to the most common data path like payload-processing or acknowledgment-processing. These so called *fast paths* have been optimized to speed up performance.

Integrated Layer Processing (first mentioned in [10]) promotes the integration of inter-layer data manipulations to reduce CPU cycles due to less data touching, more register caching and reduced loops. An implementation of this technique has been done in [19].

H. Related patterns

The relation of order-type and order follows the Type-Object pattern [20]. We use it here instead of the prototype pattern to separate static and dynamic properties.

Data Path Reification can be considered a refinement of *Outsourced De-Multiplexing*: while the latter promotes the isolation of data streams of different protocol sessions, the former assures isolation of data streams within a protocol session.

Orders may be implemented following the *Active-Object* pattern [21]. That is, the OrderType object maintains a queue to store requests for new orders and a single Order object, and implements a thread, which reads requests from the queue, gives the data to its order-object, and triggers execution.

I. Resulting Context

To reduce the network load, the packets produced by different orders may be collected and sent into a single packet. This pattern has been identified and described in [22]. An example are RTP compound packets [23].

VI. Pattern: Data Path Partitioning

A. Context

While *Data Path Reification* vertically structures protocols and makes them extensible for new services, whole orders are hardly re-usable and changing them would still require expensive modifications.

B. Problem

How to maximize decoupling of variable protocol components such as protocol processing and data representation, while providing a fine-grained structure at the same time to assure flexibility?

C. Forces

- performance and efficiency must be carefully traded off: fine-grained structuring is necessary to be flexible but may result in overhead
- implementing new protocols requires a lot of testing, experimenting and runtime modification
- the message header may be object of change, too
- protocol functions are heterogeneous, they reach from simple data-manipulations up to complex interactions with other functions
- re-usability requires independent modules; but protocol functions often largely depend on parameters provided by other protocol functions
- message processing may result in producing and processing new messages (which thread should do that?)

D. Solution

Partition data-paths into a data-representation part and a functional-part. Both parts are divided in small, configurable components. The data part encapsulate properties of message headers, the functional part is horizontally sliced into elementary operations.

Entry-Type: encapsulates a header-field type (e.g. a sequence-number, a checksum-value, or a time-stamp). Entry types are used to create Entry objects, i.e. the runtime representation of an EntryType object (see below). An EntryType can be configured *initializable* and/or *visible*. All runtime entries of initializable entry-types are filled with data from either application or network before processing. All runtime entries of visible entry-types are delivered to the application or sent via the network after processing.

Entry: is the runtime-representation of an entry-type. It provides methods to parse raw data and transform it into typed variables (used when the corresponding entry-type is initializable) and methods to transform typed variables into an byte array (used when the corresponding entry-type is visible).

Worker: encapsulates a protocol operation (e.g. sequencing, checksumming, round-trip-time measurement). In contrary to a layer, a worker encapsulates either an output or an input function. Another difference with a layer is that a worker does not operate on complete messages, but only on entry objects. Hence, it does not have any knowledge about the message format and does not need to perform any parsing or message manipulations.

Order-Type: reifies a data path of a protocol by registering all functional steps (set of worker objects), the data header format (set of entry-type objects), and the relation between both (i.e. which entry-type serves as a parameter for which worker). It provides a method to create new order objects.

Order: an order object delegates parsing to its entry objects and then calls all its workers with their specified parameter entries.

E. Dynamics

We distinguish three phases in the dynamics of this pattern. In the **registration phase** the application registers all workers and entry-types at the corresponding order-type objects via `OrderType.register()` and configures them. This phase represents the protocol construction.

In the **initialization phase**, the protocol environment has determined a special order-type object for a piece of data and creates a new order object via `OrderType.createOrder()`. It then initializes the order object with the piece of data calling `Order.initialize()`. The order object creates new entry objects using `EntryType.create()` and gives the data to all those entry objects that are *initializable* via `Entry.fill()`. Each entry object then takes out and represents the data that is meant for it.

In the **execution phase**, the protocol environment calls the `Order.execute()` method. The order object calls all its workers with their corresponding entry-object via `Worker.call()`. A worker may obtain and manipulate the entry objects via `Entry.getValue()` and `Entry.setValue()`. After processing, all entries configured *visible* are sent or delivered.

F. Known uses

In the context of business processes (e.g. a client orders an article by telephone), a worker corresponds to a certain task (e.g. talking with the client, fetching the article from the store, pack the article, preparing the bill, sending the article) and entries correspond to information needed during execution of the tasks (e.g. client number, name, address, article number, etc.). In the context of operating systems, a worker entity corresponds to a task, an entry corresponds to a resource.

The relation between worker and entry-type also corresponds to the relation between the fundamental abstractions of function oriented languages *function* and *parameter type*. Both use parameterization to provide re-use and share information with other workers/functions. A worker thereby reifies a function, an entry reifies a parameter. The idea is the same: decouple a function from its context and thus improve its re-usability.

Dividing protocols in small functional units has first been proposed, implemented, and evaluated by [24] based on TCP/IP. The result is a system that breaks up layers and structures a protocol stack into a high number of fine-grained partly re-usable micro-protocols that are con-

```
public class OrderType {
  /** dynamic list of workers */
  Vector workerList=new Vector();
  /** dynamic list of entries */
  Vector entryTypeList=new Vector();
  /** array of entry-types */
  Vector parameterList=new Vector();

  /** registration of workers and entry types */
  public void register(Worker w, EntryType e) {
    int workerIndex=workerList.indexOf(w);
    if (workerIndex==-1) {//worker not yet registered
      workerList.addElement(w);
      parameterList.addElement(new Vector());
      workerIndex=workerList.size();
      }
    }
    int entryIndex=entryList.indexOf(e);
    if (entryIndex==-1) {//entry-type not yet registered
      entryTypeList.addElement(e);
      entryIndex=entryTypeList.size();
    }
    ((Vector)parameterList.elementAt(i)).addElement(new Integer(workerIndex));
  }

  /** creation of new orders using byte arrays */
  public Order createOrder(byte[] data) {
    Order newOrder=new Order();
    newOrder.workerList=workerList;
    newOrder.entryTypeList=entryTypeList;
    newOrder.parameterList=parameterList;
    newOrder.initialize(data);
    return newOrder;
  }

  /** creation of new orders using objects */
  public Order createOrder(Object[] data) {
    Order newOrder=new Order();
    newOrder.workerList=workerList;
    newOrder.entryTypeList=entryTypeList;
    newOrder.parameterList=parameterList;
    newOrder.initialize(data);
    return newOrder;
  }


}
```

Fig. 6. Pseudo Code for Order Type

nected via so called virtual protocols that encapsulate IF functions. Micro-protocols are very similar to workers.

Plagemann et al. [25] allow flexible composition of protocol modules by providing them with meta information about the relevant fields and their position in the data header. This idea is similar to entries and has been implemented in the DaCaPo framework.

Renesse developed a Protocol Accelerator [26] that ignores layer boundaries to let layers share message header data.

G. *Related Patterns*

Data Path Partitioning is tightly coupled to *Data Path Reification* and intents to solve the problems of vertical structuring. However, it may be applied also to simple layered stacks.

Like OrderType and Order, also EntryType and Entry follow the Type-Object pattern to sepa-

```
public class Order {
  /** list with all workers (obtained by OrderType object) */
  Vector workerList;
  /** list with all entry-types (obtained by OrderType object) */
  Vector entryTypeList;
  /** list with all parameter relations (obtained by OrderType object) */
  Vector parameterList;
  /** list with all entries */
  Vector entryList=new Vector();

  /** creates new entry objects and fills them with a byte array */
  public void initialize(byte[] data) {
    int startByte=0;
    for (int i=0; i<entryTypeList.size(); i++) {
      EntryType et=(EntryType)entryTypeList.elementAt(i);
      Entry newEntry=et.createEntry();
      entryList.addElement(newEntry);
      startByte+=newEntry.fill(startByte,data);
    }
  }
  /** creates new entry objects and fills them with an object */
  public initialize(Object[] data) {
    int counter=0;
    for (int i=0; i<entryTypeList.size(); i++) {
      EntryType et=(EntryType)entryTypeList.elementAt(i);
      Entry newEntry=et.createEntry();
      entryList.addElement(newEntry);
      if (newEntry.initializable==true)
        newEntry.setValue(data[counter++]);
    }
  }
  /** calls all workers with their corresponding parameter entry objects */
  public execute() {
    for (int i=0; i<workerList.size(); i++) {
      Worker currentWorker=(Worker)workerList.elementAt(i);
      Vector paramPerWorker=(Vector)parameterList.elementAt(i);
      Entry[] paramEntries=new Entry[paramPerWorker.size()];
      //build parameters
      for (int j=0; i<parameters.size(); i++) {
        int idx=((Integer)parameterList.elementAt(j)).intValue();
        paramEntries[j]=(Entry)entryList.elementAt(idx);
      }
      currentWorker.call(paramEntries);
    }
  }

...

}
```

Fig. 7. Pseudo Code for Order

rate static and dynamic dimensions of header fields.

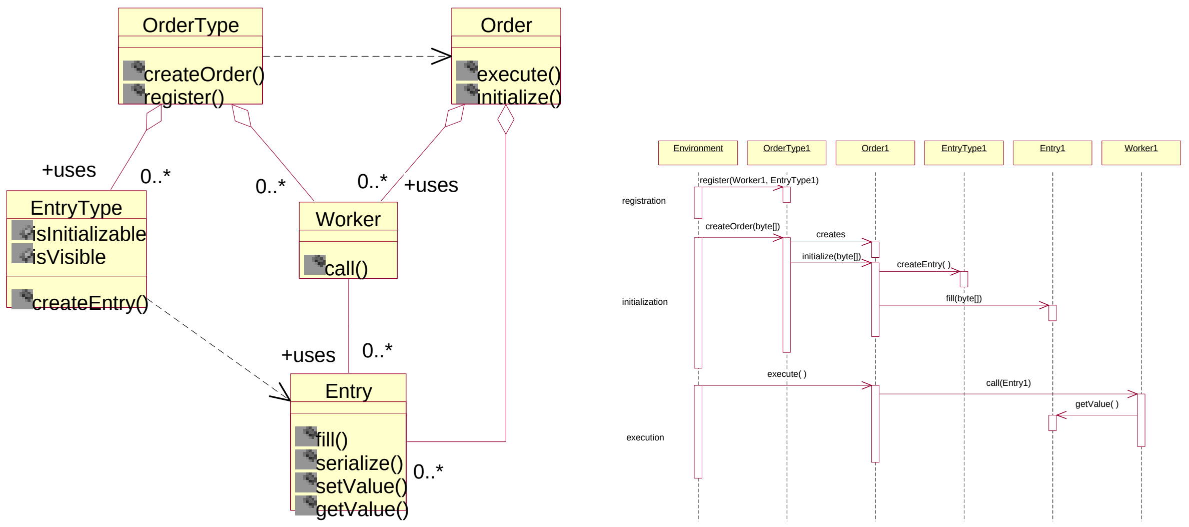
A worker is similar to a command in the *Command* pattern [27]. Environments can be considered as managers from the *Manager* pattern [28].

The pattern *Configurable System* [29] describes an overall architecture that corresponds to the way we propose to implement *protocol* systems. *Data Path Partitioning* can be considered a way to design configurability for *Configurable System* in the particular area of protocol system design.

The data-flow architecture pattern [30] also promotes dividing data processing into functional steps, but does not tackle the problem of how different modules can share the same data.

```
public abstract class EntryType {
  /** indicates if entry-type is used for initialization */
  boolean initializable;
  /** indicates if entry-type is used by application/network */
  boolean visible;
  /** creation of a new entry */
  Entry createEntry();
}

public abstract class Entry {
  /** indicates if entry-type is used for initialization */
  boolean initializable;
  /** indicates if entry-type is used by application/network */
  boolean visible;
  /** byte-array representation */
  byte[] myData;
  /** object representation */
  Object myObject;

  /** take bytes from an array and represent them as an object */
  int fill(int start, byte[] data);
  /** transform the object into an byte-array */
  byte[] serialize();
  /** allows to access the object */
  void setValue(Object o) {myObject=o;}
  Object getValue() {return myObject;}
}

public class IntegerEntryType extends EntryType {
  /** indicates how many bytes are used to represent an integer */
  int nrBytes;

  /** creates a new IntegerEntry and initializes it */
  public Entry createEntry() {
    IntegerEntry newEntry=new IntegerEntry();
    newEntry.initializable=initializable;
    newEntry.visible=visible;
    newEntry.nrBytes=nrBytes;
    return newEntry;
  }
}

public class IntegerEntry extends Entry {
  /** indicates how many bytes are used to represent an integer */
  int nrBytes;

  /** takes bytes from an array and represents them as an Integer */
  public fill(int s, byte[] data) {
    if (initializable==false)
      return 0;
    //copy ’nrBytes’ bytes into the array ’myData’
    System.arraycopy(data,s,myData,0,nrBytes);
    //transform the ’nrBytes’ bytes starting at position ’s’ into an Integer
    ...//byte[] data -> Object myObject
    return nrBytes; //return the number of bytes "taken"
  }

  /** transforms the Integer into an byte array of length ’nrBytes’
  public byte[] serialize() {
    if (visible==false)
      return new byte[0];
    //transform the object of myObject into an byte-array
    byte[] data=...//Object myObject -> byte[]
    return data;
  }

}
```

Fig. 8. Pseudo Code for an Entry/EntryType implementation



(a) Class diagram (b) Dynamics

Fig. 9. Data Path Partitioning

H. *Discussion*

Data Path Partitioning ensures autonomy and thus re-usability of entry and worker components. It de-couples protocol functions, message parsing from message processing, and input-from output processing.

In contrary to the layered model, protocol functions can share data easily.

At the other hand, this pattern also introduces a processing overhead due to modularization and communication among the various components. A large number of objects also increase the memory footprint of fine-grained structured protocol systems. Flexibility and efficiency must be carefully traded-off when applying *Data-Path Partitioning*.

I. *Resulting context*

While *Data Path Partitioning* provides high autonomy and fine-granularity of the participating protocol components along the data-path, it does not provide any solution for the communication of workers that belong to different orders. The problem of communication between autonomous component is not specific to protocol implementation and is addressed mainly in the context of component-based programming. One way to let workers communicate control-information would be to use the observer pattern [27]. The communication protocol is thereby defined by event types. Component builder tools could generate event-adapters to provide full independence of components. Eskelin [31] proposes a useful set of component interaction patterns.

Experiments with a component-based protocol framework [32] showed us that relations between the various components in the *Data Path Partitioning* pattern can be classified into three groups: in *notification-relations* where one worker notifies another one about a certain event, in *shared-value-relations* where one worker is writing to a state that is read by another worker, and in *new-order-relations* where one worker signals the request of a new order to be executed. Here may be potential of identifying protocol component interaction patterns.

Another subject not addressed by the *Data Path Partitioning* pattern is how to integrate modules in the environment that are not in a data path, e.g. periodical table updates, timers, management functions.

VII. Pattern: Data Path Classification

A. Context

While the three presented patterns assure extensibility, re-usability, and flexibility of protocol components, they are not concerned about how the protocol interacts with the underlying network and the application that uses it.

B. Problem

How should a protocol interact with the underlying network and the application without any knowledge about application or network semantics? How can this be done according to a vertical structure?

C. Forces

- the flexibility of a protocol from the perspective of the application is mainly determined by its interfaces
- there are many different ways applications may like to communicate with a protocol
- network interfaces are generally complicated and platform dependent

D. Solution

Classify orders according to their interaction with application and network by extending the OrderType and the Order class and provide each and provide each classified order-type with an interface object and the needed information to interact with application/network.

OrderType class	**Direction**	**Interface**	**Attributes**
Acceptance	Output	From Application	Write Access Point for the application
Delivery	Input	To Application	Read Access Point for the application
Emission	Output	To Network	Output Port Number
Reception	Input	From Network	Mapping Information, Input Port Number

TABLE I

Classified orders

Acceptance(/-Type): are orders that start processing after an application request.

Emission(/-Type): are orders that send data via the network after processing. Each Emission-Type object hence maintains a reference to a network handle (SAP, see below), which it gives to all its created Emission objects for sending.

Delivery(/-Type): are orders are which deliver data to the application after processing. Each DeliveryType object hence maintains a reference to an application interface (ReadAPI, see below), which it gives to all its created Delivery objects.

Reception(/-Type): are orders that start processing upon reception of data from the network.

SAP: wrapper to a network resource handle (e.g. TCP-Socket, Datagram-Socket) to hide network specifics from the protocol. Each emission-type specifies on which SAP number it sends.

WriteAPI: defines a write interface for the application. A protocol contains as many write-APIs as acceptance-types. Each WriteAPI object maintains a reference to an AcceptanceType object. When the application wants to insert data into the protocol, it gives it to the wished WriteAPI and thus automatically specifies the acceptance order to be processed.

ReadAPI: defines a read interface for the application. Each DeliveryType object maintains a reference to a ReadAPI object and gives it to its Delivery objects, which call the ReadAPIs hook method to deliver its processed data.

Note that the different types of orders may be combined: an order may be accepted and emitted (output order) or received and delivered (input order). Orders that do not interface with application or network are called **internal orders**; those are created during the execution of other orders. An overview of the classification of orders can be seen in Table I.

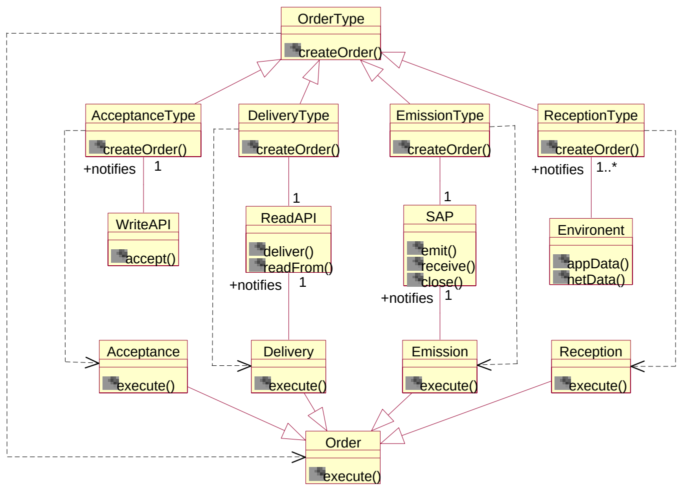


Fig. 10. Class diagram of Data Path Classification

E. Dynamics

Accept: The application gives data to one of its WriteAPI objects by calling `WriteAPI.accept()`. The WriteAPI object calls the `AcceptanceType.createOrder()` of its AcceptanceType object to create a new Acceptance object and executes it calling `Acceptance.execute()`. As a variant, the WriteAPI could delegate acceptance-requests to the environment instead of creating and executing an emission orders itself. This would have the advantage that the environment serves as a central instance that controls all order requests and executions.

Deliver: Delivery orders are created and executed either upon the reception of network data or by another order. The DeliveryType objects give all the Delivery objects they create a reference to a ReadAPI object. Within the `Delivery.execute()` method the `ReadAPI.deliver()` method is called to hand out the processed data, where it is either written to a queue or better directly used by the application.

Receive: The environment receives data from the network, maps it to the right ReceptionType object, creates a new Reception object via `ReceptionType.createOrder()` and starts processing via `Reception.execute()`.

Emit: Emission orders are created and executed either upon the acceptance of application data or by another order. The EmissionType objects give all the Emission objects they create

```
public interface WriteAPI {
  void accept(byte[] data);
}

public class StandardWriteAPI implements WriteAPI {
  /** order-type associated with this interface */
  OrderType myOrderType;
  /** environment */
  Environment muyEnv;

  /** constructor */
  public void StandardInterface(OrderType o, Environment e) {
    myOrderType=o;
    myEnv=e;
  }
  /** gives the accept-request to the environment specifying the order-type */
  void accept(Object[] data) {
    myEnv.accept(myOrderType,data);
  }

}

public interface ReadAPI {
  void deliver(byte[] data);
}

public class QueuingReadAPI {
  /** a queue storing all delivered data chunks */
  private Vector queue=new Vector();

  /** called by the executing delivery-order */
  public void deliver(Object[] data) {
    queue.addElement(data);
  }

  /** method to read data from the queue */
  public Object[] readFrom() {
    Object[] o=(Object[])queue.firstElement();
    queue.removeElementAt(0);
    return o;
  }

}
```

Fig. 11. Pseudo Code for the Interfaces

a reference to a SAP object. Within the `Emission.execute()` method the `SAP.emit()` method is called, which then send the data to the network.

F. Known Uses

Nearly all protocol frameworks follow the notion of input and output direction and the classification of protocol interfaces even when the concrete names are very different. Each layer within the X-Kernel [33] provides the methods *xPushTo* (emit), *xPushFrom* (receive), *xPopTo* (deliver) and *xPopFrom* (accept).

Streams [5] uses the notions *user write* (accept), *user read* (deliver), *device out* (emit), *device in* (receive).

Hüni [34] equips each of his conduits with two bi-directional interfaces.

```
public class Delivery extends Order {
  /** reference to a read api */
  ReadAPI myReadAPI;
  /** number of entry-types that are configured visible */
  int nrDeliveryEntries;
  ...
  /** after execution the data is given to the read-API */
  public void execute() {
    super.execute();
    Object[] delivery=new Object[nrDeliveryEntries];
    for (int i=0; i<delivery.length; i++) {
      delivery[i]=((Entry)entryList.elementAt(i)).getValue();
    }
    myReadAPI.deliver(delivery);
  }
}

public class Emission extends Order {
  /** reference to the SAP */
  SAP mySAP;
  /** number of entry-types that are configured visible */
  int nrEmissionEntries;
  ...
  /** after execution the data is transformed into an byte array and sent */
  public void execute() {
    super.execute();
    byte[] emission=new byte[];
    for (int i=0; i<nrDeliveryEntries) {
      byte[] toAdd=((Entry)entryList.elementAt(i)).serialize();
      emission=ByteArithmetic.append(emission,toAdd);//helper class to append
    }
    mySAP.send(emission);
  }
```

Fig. 12. Pseudo Code for Delivery and Emission

G. *Related Patterns*

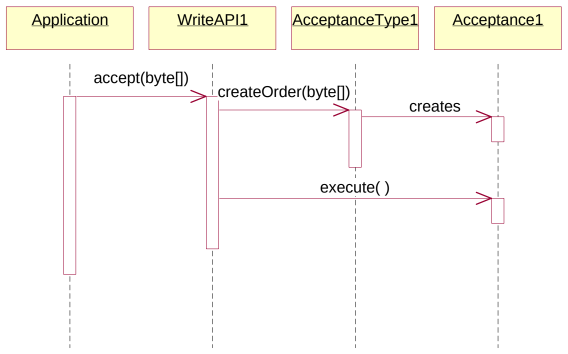
This pattern uses the well known *Adapter* pattern to de-couple the underlying network resource handle (SAP) from the protocol, the Observer pattern to allow the application interfaces to be notified about new data, and the *Bridge* pattern (all in [27]) to achieve flexibility by separating interface and implementation, i.e. Write/ReadAPIs and AcceptanceType/DeliveryType.

VIII. Summary

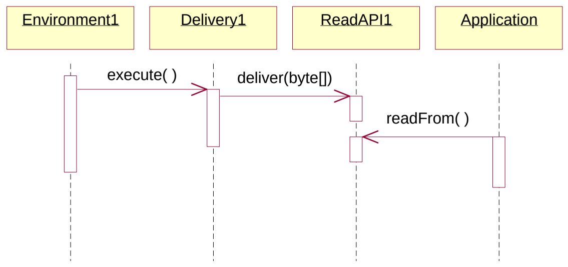
We presented a system of architectural patterns that allow to maximize flexibility and re-usability of application tailored end-to-end protocols. The main idea behind is to introduce vertical structuring instead of fine-grained layering. All patterns have successfully been implemented in a Java framework [32]. Figure 15 illustrates the complete architecture.

We shortly resume how the goals above are achieved:

- The streams of different protocol sessions are decoupled (*Outsourced De-Multiplexing*)
- A protocol can be extended by new services without interfering with existing services (*Data Path Reification*)
- A functional unit does not need to know about what need to be done next (*Data Path Reification* and *Outsourced De-multiplexing*)
- A functional unit does not need any information about where in the message its relevant header information can be found (*Data-Path Partitioning*)
- Header representation units can be re-used in any context and are extremely re-usable (*Data-Path Partitioning*)

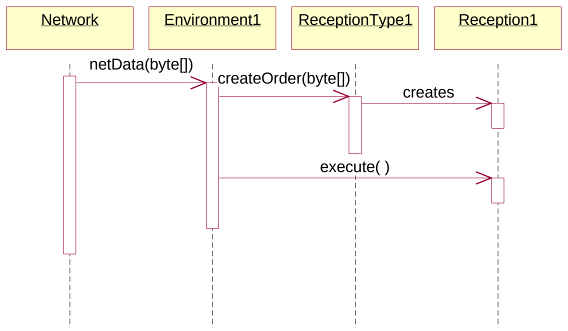


(a) Acceptance

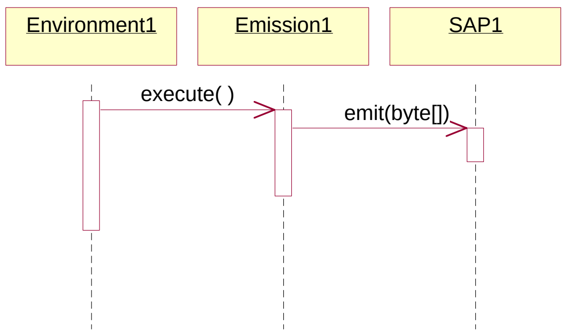


(b) Delivery

Fig. 13. Dynamics of Data Path Classification (Application Interface)



(a) Reception



(b) Emission

Fig. 14. Dynamics of Data Path Classification (Network Interface)

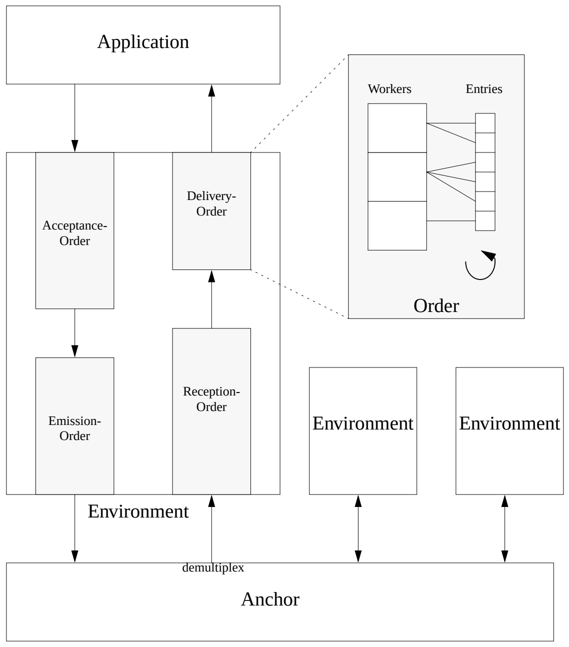


Fig. 15. Architecture implementing Order-Worker-Entry

• A functional unit does not need to know where input information comes from, where output information goes to, who is handling the events he fires, who fired the events he is notified of, and which kind of order he creates and what is happening with this order (Component Interaction Patterns)
• The protocol does not need to know anything about the underlying network or the application that uses it (*Data Path Classification*)

While the described pattern system focuses on the design of application tailored protocols, they may also be considered as a way to structure any protocol layer. It should be seen more a complementary rather than an alternative concept to layering.

ACKNOWLEDGMENTS

I'm very grateful to my shepherd Peter Sommerlad for his valuable comments and suggestions on earlier drafts on this paper that lead to significant improvements.

This work is sponsered by Siemens ZT IK2, Munich.

REFERENCES

[1] J. Postel, "Transmission Control Protocol", *Internet Request for Comments*, RFC 793, Network Information Center, September 1981.
[2] J. Postel, "User Datagram Protocol – Protocol Specification", Request for Comments (Standard) RFC 768, Information Sciences Institute, USC, August 1981.
[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A Systems of Patterns*, John Wiley & Sons, 1996.
[4] V. Roca, T. Braun, and C. Diot, "Demultiplexed Architectures: a Solution for Efficient STREAMS based communication stacks", *IEEE Networks Magazine*, June 1997.
[5] Unix, "STREAMS Programmer's Guide", *Unix System V Release 4*, 1990.
[6] 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.
[7] T. Braun, C. Diot, A. Hoglander, and V. Roca, "An Experimental User Level Implementation of TCP", , INRIA, 1995.
[8] M. Jung, E. Biersack, and A. Pilger, "Implementing Network Protocols in Java - A Framework for Rapid Prototyping", *Proceedings of ICEIS*, Setubal, Portugal, March 1999.
[9] D. C. Schmidt, "Patterns and Frameworks for Concurrent network Programming with ACE", 1999.
[10] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols", *Proc. ACM SIGCOMM 90*, pp. 200–208, Phildelphia, PA, September 1990.
[11] W. Atwood, "Concurrency in Operating Systems", *Computer*, 9(10):18–26, 1976.
[12] M. Ben-Ari, *Principles of Concurrent Programming*, Englewood Cliffs, 1982.
[13] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publ., Reading, MA, 1989.
[14] D. Cheriton, "The V Kernel: A Software Based for Distributed Systems", *IEEE Software*, pp. 19–42, 1984.
[15] A. Birrel, "An Introduction to Programming with Threads", , Digital Systems Research Center, 1989.
[16] Sun Microsystems, "The Java Virtual Machine Specification", , 1995.
[17] D. Mosberger and L. Peterson, "Making Paths Explicit in the Scout Operating System", *Proceedings of OSDI*, pp. 153–168, October 1996.
[18] V. Jacobson, "4BSD TCP Header Prediction", *Computer Communication Review*, 20(2):13–15, April 1990.
[19] T. Braun and C. Diot, "Protocol Implementation Using ILP", *Proceedings of ACM SIGCOMM'95*, pp. 151–161, 1995.
[20] R. Johnson and B. Woolf, "The Type Object Pattern", http://www.ksccary.com/Articles/TypeObjectPattern.html, 1997.
[21] D. Schmidt and F. Buschmann, "The Active Object Pattern", 1998.
[22] M. Patio, F. Ballesteros, R. Jiménez, S. Arévalo, F. Kon, and R. Campbell, "CompositeCalls:A Design Pattern for Efficient and Flexible Client-Server Interaction", *In Proceedings of PloP*, 1999.
[23] H. Schulzrinne, S. Casner, R. Frederic, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", Request for Comments RFC 1889, Internet Engineering Task Force, January 1996.
[24] S. W. O'Malley and L. L. Peterson, "A Dynamic Network Architecture", *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.
[25] T. Plagemann, P. B., M. Vogt, and W. T., "Modules as building blocks for protocol configuration", *Proceedings of the international Conference on Network Protocols (ICNP-93)*, September 1993.
[26] R. van Renesse, "Masking the Overhead of Protocol Layering", *Proceedings of ACM Sigcomm*, September 1996.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing, 1994.

[28] P. Sommerlad and F. Buschmann, "Manager", *PLoP*, 1996.

[29] P. Sommerlad, "Configurability", *EuroPLoP*, 1999.

[30] D.-A. Manolescu, "A Data Flow Pattern Language", *In Proceedings of PLoP*, 1997.

[31] P. Eskelin, "Component Interaction Patterns", *Proceedings of PLoP*, August 1999.

[32] M. Jung and E. Biersack, "A Component-Based Architecture for Software Communication Systems", *Proceedings of IEEE ECBS*, Edinburgh, Scotland, April 2000.

[33] 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.

[34] 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.