PROTOCOL STRUCTURE 2

2.1 INTRODUCTION

In the first chapter we have seen some general examples of the protocol design problem. Having chosen a transmission medium, be it a torch telegraph or an optical fiber, we have to write down a set of rules for its proper use, defining how messages are encoded, how a transmission is initiated and terminated, and so on. Two types of errors are hard to avoid: designing an incomplete set of rules, or designing rules that are contradictory.

In this chapter we look at ways to make sure that the set of rules is both complete and consistent. It requires us to be very precise in specifying *all* the relevant pieces of a protocol. It also requires some discipline in separating orthogonal issues, using modularity and structure.

Let us first look at the general types of services that a computer communications protocol must be able to provide. Assume we have two computers, *A* and *B*. *A* is connected to a file store device *d*, and *B* connects to a printer *p*. We want to send a text file from the file store at *A* to the printer at *B*.

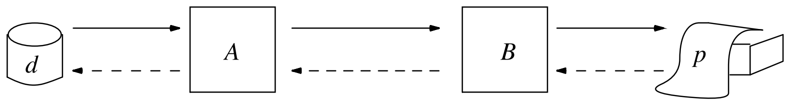

Figure 2.1 — File Server and Print Server

Obviously, to be able to communicate at all, the two machines must use the same physical wires, use compatible character encodings, and transmit and scan the signals on the wires at roughly the same speed. But, assuming that those issues have been resolved, there is still more to the problem than sending signals down a wire.

A must be able to check whether or not the printer is available. It must be able to adapt the rate at which it is sending the characters to the rate at which the printer can handle them. Specifically, the machine must be able to suspend sending when the printer runs out of paper or is switched off line.

It is important to note that, even though the actual data flow in only one direction, from *A* to *B*, we need a *two-way* channel to exchange control information. The two machines must have reached prior agreement on the meaning of control information and on the procedures used to start, suspend, resume, and conclude transmissions. In addition, if transmission errors are possible, control information must be exchanged to guard the transfer of the data. Typical control messages, for instance, are positive and negative acknowledgments that can be used by the receiver to let the sender know whether or not the data were received intact.

All rules, formats, and procedures that have been agreed upon between *A* and *B* are collectively called a *protocol*. In a way, the protocol formalizes the interaction by standardizing the use of a communication channel. The protocol, then, can contain agreements on the methods used for:

- ○ Initiation and termination of data exchanges
- ○ Synchronization of senders and receivers
- ○ Detection and correction of transmission errors
- ○ Formatting and encoding of data

Most of these issues can be defined on more than one level of abstraction (Figure 2.2). At a low level of abstraction, for instance, any synchronization concerns apply to the synchronization of the sender's and receiver's clock that is used to drive or to scan the physical transmission line. At a higher level of abstraction, it is concerned with the synchronization of message transfers (for example, in flow control and rate control methods), and at a still higher level it deals with the synchronization and coordination of the main protocol phases.

At the lowest level a format definition can consist of a method for encoding bits with analog electrical signals. One level up, it may consist of methods for encoding the individual characters of a transmission alphabet into bit patterns. Next, character codes can be grouped into message fields, and message fields into frames or packets, each with a specific meaning and structure.

The error control methods required in a protocol depend on the specific properties of the transmission medium used. This medium may insert, delete, distort, or even duplicate and reorder messages. Depending on the specific behavior, the protocol designer can devise an error control strategy.

The protocol descriptions we have discussed so far have been fairly informal and fragmented. Unfortunately, this is not unusual. It is all too tempting to rely on the goodwill and common sense of the reader (or implementer) to fill in the details that have been omitted, to understand the hidden assumptions, and to disambiguate the language. A first step towards more reliable protocol design is to formalize and to structure the descriptions, to make explicit all assumptions.

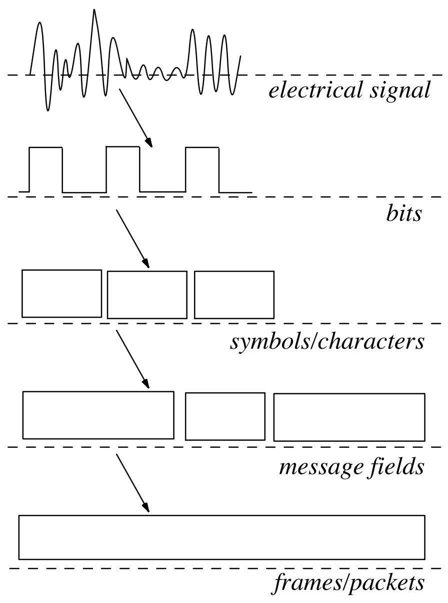

Figure 2.2 — Sample Levels of Abstraction: Formatting

In the next section we begin this process by considering what the essential elements in a protocol definition are.

2.2 THE FIVE ELEMENTS OF A PROTOCOL

A protocol specification consists of five distinct parts. To be complete, each specification should include explicitly:

1. The *service* to be provided by the protocol
2. The *assumptions* about the environment in which the protocol is executed
3. The *vocabulary* of messages used to implement the protocol
4. The *encoding* (format) of each message in the vocabulary
5. The *procedure rules* guarding the consistency of message exchanges

The fifth element of a protocol specification is the most difficult to design and also the hardest to verify. The larger part of this book is therefore devoted to precisely that topic: the design and validation of unambiguous sets of procedure rules.

Each part of the protocol specification can define a hierarchy of elements. The protocol vocabulary, for example, can consist of a hierarchy of message classes. Similarly, the format definition can specify how higher-level messages are constructed from lower-level message elements, and so on.

As noted in Chapter 1, a protocol definition can be compared to a language definition: it contains a *vocabulary* and a *syntax* definition (i.e., the protocol format); the procedure rules collectively define a *grammar*; and the service specification defines the *semantics* of the language.

There are some special requirements we have to impose on this language. Like any computer language the protocol language must be *unambiguous*. Unlike most programming languages, however, the protocol language specifies the behavior of concurrently executing processes. This concurrency creates a new class of subtle problems. We have to deal with, for example, *timing*, *race conditions*, and possible *deadlocks*. Since the precise sequence of events cannot always be predicted, the number of possible orderings of events can be so overwhelming that it defeats any attempt to analyze the protocol by simple manual case analysis.

The next section gives an informal example of the definition of the five protocol elements, and the types of errors that can linger in a design. Following that, we consider each of the five main protocol elements in more detail. The chapter is concluded with a discussion of protocol design techniques that can help to structure a design, so that it ultimately can be implemented efficiently and proven correct with automated tools.

2.3 AN EXAMPLE

The following protocol was described by W.C. Lynch [1968] as

> *''... a reasonable looking but inadequate scheme published by one of the major computer manufacturers in a system information manual.''*

We discuss this protocol here to see how we can identify the basic building blocks in a specification discussed above. Let us first consider the service specification.

SERVICE SPECIFICATION

The purpose of the protocol is to transfer text files as sequences of characters across a telephone line while protecting against transmission errors, assuming that all transmission errors can in fact be detected. The protocol is defined for full-duplex file transfer, that is, it should allow for transfers in two directions simultaneously (see also Appendix A). Positive and negative acknowledgments for traffic from *A* to *B* are sent on the channel from *B* to *A*, and vice versa. Every message contains two parts: a message part, and a control part that applies to traffic on the reverse channel.

ASSUMPTIONS ABOUT THE ENVIRONMENT

The ''environment'' in which the protocol is to be executed consists minimally of two users of the file transfer service and a transmission channel. The users can be assumed to simply submit a request for file transfer and await its completion. The transmission channel is assumed to cause arbitrary message distortions, but not to lose, duplicate, insert, or reorder messages. We will assume here that a lower-level module (see Chapter 3) is used to catch all distortions and change them into undistorted messages of type *err*.

PROTOCOL VOCABULARY

The protocol vocabulary defines three distinct types of messages: *ack* for a message combined with a positive acknowledgment, *nak* for a message combined with a negative acknowledgment, and *err* for a message with a transmission error. The vocabulary can be succinctly expressed as a set:

V = { ack, err, nak }.

Each message type can further be refined into a class of lower-level messages, consisting for instance of one sub-type for each character code to be transmitted.

MESSAGE FORMAT

Each message consists of a control field identifying the message type and a data field with the character code. For the example we assume that the data and control fields are of a fixed size.

The general form of each message can now be represented symbolically as a simple structure of two fields:

{ control tag, data }

which in a C-like specification may be specified in more detail as follows:

```
enum control { ack, nak, err };

struct message {
        enum control    tag;
        unsigned char   data;
};
```

The line starting with the keyword `enum` declares an enumeration type named `control` with three possible values: one for each message type used. The message structure itself contains two fields: a `tag` of type `control`, and a `data` field declared as an unsigned character (one byte).

PROCEDURE RULES

The procedure rules for the protocol were informally described as follows:

''1. If the previous reception was error-free, the next message on the reverse channel will carry a positive acknowledgment; if the reception was in error it will carry a negative acknowledgment.''
''2. If the previous reception carried a negative acknowledgment, or the previous reception was in error, retransmit the old message; otherwise fetch a new message for transmission.''

To formalize these rules, we can use state transition diagrams, flow charts, algebraic expressions, or program-form descriptions. In Chapters 5 and 6 we develop a new language to describe procedure rules like these in protocol validation models. For the time being, though, we can use simple flow charts, such as the one shown in Figure 2.3. An overview of the flow chart language is given in Appendix B.

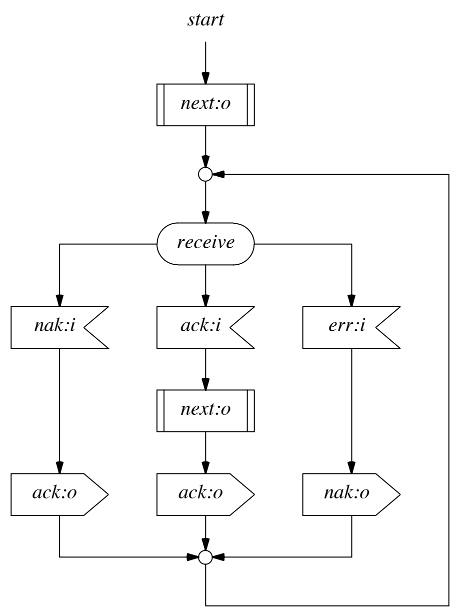

Figure 2.3 — Lynch's Protocol

The box labeled *receive* symbolizes a state in which the reception of a new message from the channel is awaited. Depending on the type of message received, one of three execution paths is then chosen. The dented box represents the recognition of a message of the type that matches its label. The pointed box indicates the transmission of a message with the corresponding type.

The box labeled *next:o* indicates an internal action to obtain the next data item (character) to be transferred. The data item is stored in variable *o*, which is used in the output operations. For instance, *ack:o* sends data item *o* with a positive acknowledgment of the last received message. Incoming data is stored in variable *i*.

As we might expect, there are some problems with this description that need to be considered.

DESIGN FLAWS

First we have the problem that data transfer in one direction can only continue if data transfer in the other direction also takes place. We could try to overcome this problem by having the processes use filler messages whenever no real data are to be transferred.

Another problem that has to be solved before the protocol can be used is to decide how a data transmission is to be initiated or concluded. The two procedure rules specify normal data transfer, but not the setup and termination procedures.

We can try to initiate the data transfer by having one of the two processes send a fake error message. Note that if both parties are allowed to initiate the protocol in this way, it is hard to bring the two processes into phase. To terminate the transfer when the processes have ended up exchanging filler messages, however, requires extra control messages.

A more important deficiency of the protocol is that an essential operation has been omitted from the specification. The receiver has to be able to decide whether or not a data item that was received correctly, and temporarily stored in variable *i*, is to be accepted (and, for instance, saved in a file). Correctly received duplicates of previously received messages should, of course, not be accepted again. This problem seems to have no solution if we are to maintain the two procedure rules listed above.

Consider what can happen if every correctly received message is accepted, that is, data appended to *ack* and *nak* messages is accepted, but data appended to *err* messages is not. The extension looks plausible enough but unfortunately does not solve the problem. The following execution sequence, for instance, leads to the acceptance of a duplicate message. First, process *A* initiates the transfer by sending a deliberate error message to *B*. Assume that *A* attempts to transmit the characters *a* to *z*, and that *B* responds by transmitting the characters in the reverse order, from *z* to *a*. Consider then the sequence of events shown in the time sequence diagram of Figure 2.4. The two solid lines in the figure track the executions of the two processes. The dotted lines show successful message transfers. The dashed lines show message transfers that are distorted by the channel. Two messages are distorted in this manner: a positive acknowledgment from *A* to *B* and a negative acknowledgment from *B* to *A*.

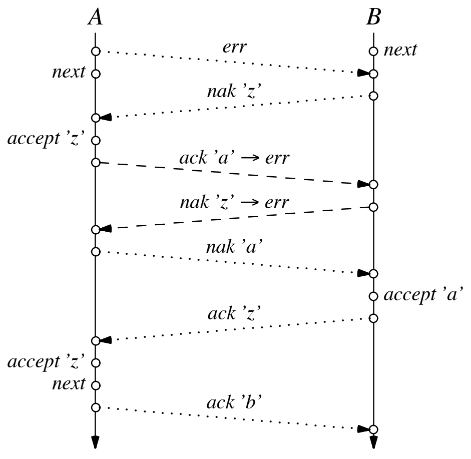

Figure 2.4 — Time Sequence Diagram

At the end of the sequence, when *A* receives the last message from *B*, it cannot tell whether the message is new or an old duplicate. The *nak* message that contained this information was corrupted. In the example sequence, *A* erroneously accepts the message.

It must be noted that, even though the protocol is simple, it is disproportionately hard to discover the error. To assume that the error, if overlooked in the design phase, will sooner or later reveal itself in practice would be naive. The error only occurs in the rare event that two transmission errors occur in sequence. As Lynch observed:

> *''Such errors, while rare, do occur, and their rareness will make it extremely difficult to catch the flaw in the system. This inadequate scheme will work 'almost' all of the time.''*

The example protocol is simple. The informal description is convincing, and based on that description alone few would doubt the protocol's correctness. Yet the specification is incomplete, and any straightforward implementation allows subtle errors during the exchange of the data. If anything, this example should convince us that, even for the simplest of protocols, a good design discipline and effective analytical tools are indispensable.

In the next sections we return to the five elements of a protocol specification defined in Section 2.2, and consider the corresponding structuring methods and design criteria that we could use. First, in Section 2.4, we consider the structuring of service specifications and the explicit assumptions that must be made about a protocol's environment. In Section 2.5, we look at the protocol vocabulary and data format, and in Section 2.6 we talk in more detail about the issues involved in the design of protocol procedure rules.

2.4 SERVICE AND ENVIRONMENT

To accomplish a higher-level task like file transfer, a protocol must perform a range of lower-level functions such as synchronization and error recovery. The specific realization of a service depends on the assumptions that are made about the environment in which the protocol is to be executed. Error recovery, for instance, should correct for the assumed behavior of the transmission medium. Particulars on the types of assumptions one can make about transmission channels are given in Appendix A and in Chapter 3. Here we concentrate on the structure of service specifications proper.

Common sense tells us that if a problem is too large to solve we must partition it into subproblems that are either easier to solve or that have been solved before. Software, and in particular protocol software, is then most conveniently structured in *layers*. More abstract functions are defined and implemented in terms of lower-level constructs, where each layer hides certain undesirable properties of the communication channel and transforms it into a more idealized medium.

As an example, assume we want to implement a data transmission protocol that provides for the encoding of characters into tuples of 7 bits each, and for some rudimentary error detection scheme to protect the bytes against transmission errors, for instance by the addition of one parity bit to each 7-bit byte. This protocol then provides two *services*: encoding and error detection. We can separate these two services into two functional submodules, an encoder and a parity module, and invoke them sequentially. At the other end of the line, there will be a decoder and a parity checker. For full-duplex transmission, we can conveniently combine the function of the encoder and decoder into one module, say P_2, and similarly we can combine the parity adder and checker into a single module P_1.

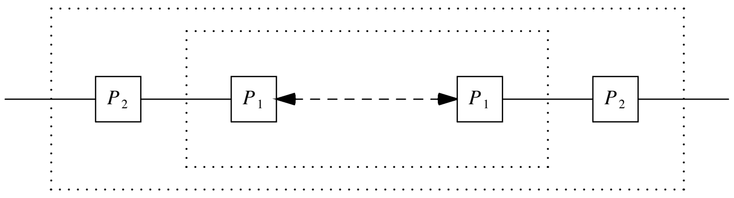

Figure 2.5 — Building a Virtual Channel

Figure 2.5 illustrates the principle. The channel (the dashed line) is wrapped in two layers. In effect, each layer provides a different service and implements a separate protocol. The first layer implements the P_1 protocol; the second layer implements the P_2 protocol. The data format of the P_2 protocol is a 7-bit byte. The data format of the P_1 protocol is an 8-bit byte.

The P_2 protocol does not see and does not know about the eighth bit that is added to its bytes. The only thing it cares about is that the channel its 7-bit bytes travel on is more reliable than the raw channel at the lower level. The P_1 protocol provides a virtual channel for the P_2 protocol, but is transparent to the P_2 protocol. The two keywords are *transparent* and *virtual*. ''Transparent'' is something that exists but seems not to. ''Virtual'' is something that seems to exist but does not.

To the P_1 protocol, any data format that is enforced by the P_2 protocol is invisible (transparent). As far as P_1 is concerned, it is an uninterpreted sequence of data, of which only the length is known. Similarly, neither the P_2 nor the P_1 protocol layer knows anything about the format imposed by possible higher layers in the hierarchy (e.g., a P_0 layer), or lower layers (e.g., P_3).

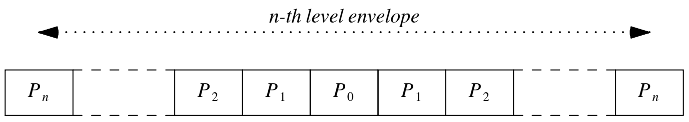

Figure 2.6 — Data Envelopes

As shown in Figure 2.6, each layer can enclose the data to be transmitted in a new data *envelope*, consisting of a header and/or trailer, before passing it to the next layer. The original data format from the upper layers need not even be preserved by the lower layers. The data may well be divided up differently, in larger or in smaller portions, as long as the original format can be restored by the receiving protocol module.

The principle of hierarchical design is well-known in sequential programming, but is relatively new for distributed systems. The advantages are clear:

□ A layered design helps to indicate the logical structure of the protocol by separating higher-level tasks from lower-level details.

□ When the protocol must be extended or changed, it is easier to replace a module than it is to rewrite the whole protocol.

In 1980 the International Standards Organization (ISO) recognized the advantages of standardizing a hierarchy of protocol services as a reference model for protocol designers. The ISO recommendation defines seven layers, as illustrated in Figure 2.7.

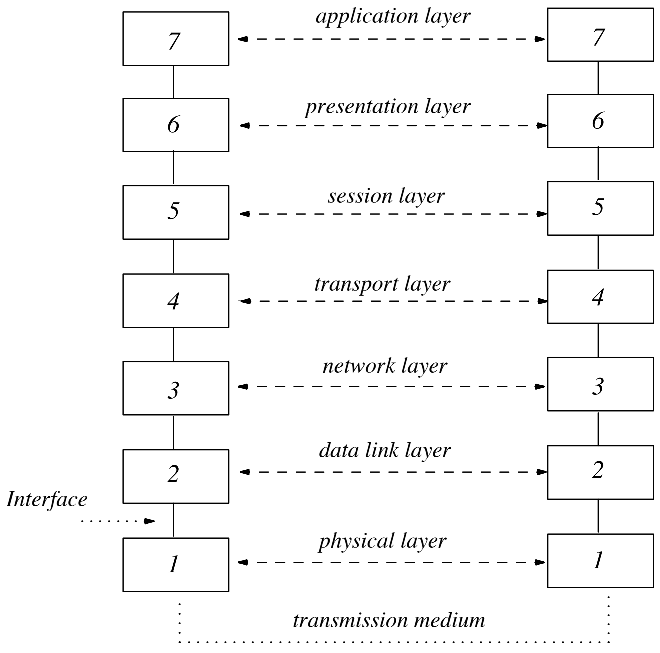

Figure 2.7 — ISO Reference Model for Open Systems Interconnection

The layers are listed below with a short descriptive phrase explaining their place in the hierarchy.

1. *Physical* layer: transmission of bits over a physical circuit
2. *Data link* layer: error detection and recovery
3. *Network* layer: transparent data transfer and routing
4. *Transport* layer: user to user higher-level data transfer
5. *Session* layer: coordination of interactions in user sessions
6. *Presentation* layer: interpretation of user-level syntax
   for instance for encryption or compression of data
7. *Application* layer: entry point for application processes
   such as electronic mail or file transfer demons

The first layer contains all protocol functions that apply to the actual transmission of bits over a physical connection. It specifies, for instance, whether a connection is a copper wire, a coaxial cable, a radio channel, or an optical fiber. The physical medium could be a *point-to-point* channel, dedicated to communication between two specific machines, or it could be a shared *broadcast* channel, such as the University of Hawaii's *Aloha* network, or an Ethernet link. All relevant properties of raw data channels and of the modems that are used to drive them (see Appendix A) are defined here. The first layer also defines the encoding of bits in, for instance, electrical, optical, or radio signals. It also defines and standardizes the mechanical requirements of cables, switches, and connectors, including pin assignments and the like. The physical layer protocols hide all these details from the subsequent layers and transform the physical line into a rudimentary data link.

The next three layers are the most important ones. Their relative function is illustrated in Figure 2.8. The boxes represent network nodes or hosts, the circles represent user-level processes executing at these hosts, and the lines represent the logical connections viewed at three different levels of abstraction.

The data link layer uses the service provided by the physical layer and transforms a raw data link into a reliable one by adding error handling. It connects two hosts, possibly but not necessarily hosts that function as nodes in a network (see Figure 2.8). It transmits the data in blocks (frames) and can provide for the multiplexing of independent data streams over a single data link. It may provide a flow control service to guarantee that frames can only be received from the link in the precise order in which they were sent, despite channel errors. Protocols that operate on the data link level are known as *link-level protocols*.

The network layer takes care of typical network functions, such as the addressing and routing of messages. It can try to avoid bottlenecks in the network by using adaptive routing schemes, or it can try to reduce congestion in the network with rate control methods. The network layer provides the means to set up and release network connections, potentially spanning multiple data links, or hops, through the network, e.g., from node *A* to node *B* in Figure 2.8.

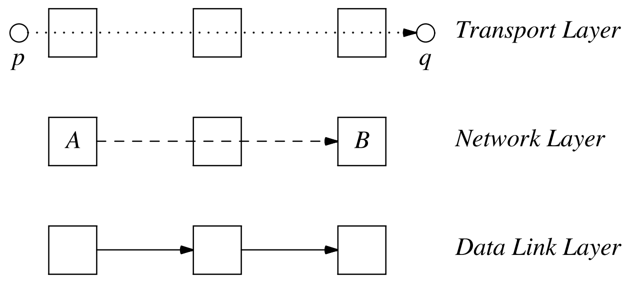

Figure 2.8 — Relative Function of Three Layers

The transport layer connects user-level processes, such as *p* and *q* in Figure 2.8, transparently through a network. Network and transport layer protocols are sometimes called *end-to-end protocols*, and data link protocols are called *hop-by-hop*. Either the network or the transport layer may provide a flow control service, which is now called end-to-end, instead of the hop-by-hop flow control that can be implemented at the data link layer. It can, in fact, make quite a difference which of these two types of flow control is used (see Chapter 4, Rate Control).

Each layer in the hierarchy defines a distinct service and implements a different protocol. The format used by any specific layer is largely independent of the formats used by the other layers. The network layer, for instance, sends data *packets*, the data link layer casts them into *frames*, and the physical layer translates them into *byte* or *bit* streams. The receiver decodes the raw data on layer 1, interprets and deletes the frame structure on layer 2, so that layer 3 can again recognize the packet structure. The format enforced by the lower layers is transparent to the higher layers.

Officially the model sketched above is called the ISO *Reference Model of Open Systems Interconnection*. It has, however, quickly become known as ISO's OSI model.

The first layers of the OSI model are the most frequently used. A layer 1 protocol was standardized by the CCITT[1] as Recommendation X.21. The recommendation for the second layer is largely based on the HDLC protocol we mentioned earlier (see Section 2.5, Bit Stuffing). The first three OSI layers together are defined in CCITT Recommendation X.25. The X.25 protocol defines the interaction of a computer, or DTE for *data terminal equipment* in CCITT terminology, and a network link, or DCE for *data circuit terminating equipment*. Computer-to-computer interaction is not defined until the fourth layer in the OSI reference model: the transport layer. A well-known transport layer protocol is the Transmission Control Protocol (TCP) that was standardized by the U.S. Department of Defense. The corresponding network layer protocol is called the Internet Protocol (IP).

1. Comité Consultatif International Télégraphique et Téléphonique.

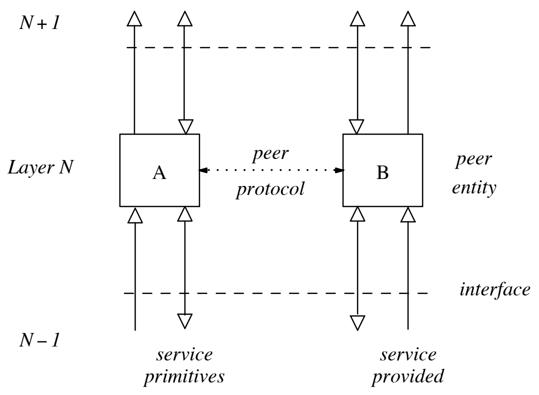

Figure 2.9 — Protocol Layering

The precise functions performed on each layer of the OSI model and the definition of the X.25 protocol are of little interest to us here (see Bibliographic Notes). More important is the structuring method itself. Software layering is a design principle that can be powerful when used properly, but it defeats its purpose when carried to extremes.

- □ A layer defines a level of abstraction in the protocol, grouping closely related functions and separating them from orthogonal ones. By decoupling layers, future changes made in one layer need not affect the design of the other layers. The correct choice of the required levels of abstraction necessarily depend on the specific protocol being designed.
- □ An interface separates distinct levels of abstraction. A correctly placed interface is small and well-defined. A badly placed interface causes unnecessary complexity, it causes code duplication, and it may degrade performance.

Figure 2.9 illustrates the main concepts of a layering technique. The protocol functions on the *N*-th layer form a logical entity. In the model they are referred to as *peer entities*. By convention the vertical boundary between two adjacent layers is called an *interface*, and the horizontal boundary between two entities in different systems is called a *peer protocol*. Since the local implementation details of the layer interfaces can easily be hidden from the environment, only the peer protocols must be standardized among systems.

The interface between two adjacent layers is defined as a collection of *service access points* implemented by the lower layer and available to the higher layer. The information to be exchanged is formatted incrementally by the various layers in *data units* or *data envelopes*. In sequence, the information is passed from the sender down from the highest layer used, to the physical layer, transmitted via the actual physical circuit from system to system, and interpreted step by step while being passed up the protocol hierarchy again to the highest layer used by the receiver.

In this framework, we can recognize the first two elements of the five-part protocol specification discussed in this section

- the *service* to be provided by the protocol, and
- the *assumptions* made about its environment

as formal specifications of the upper and lower interface of a given protocol layer. The service is provided to the upper layer protocols, or to the user at the top layer. The assumptions made are assumptions about the services provided by the lower layer protocols. At the lowest protocol layer these assumptions concern the bare service provided by the physical transmission medium, i.e., an optical fiber, a copper wire, or a torch telegraph.

The protocol hierarchy is an excellent example of the application of design discipline. Design issues are separated from one another and solved independently. The problems of error control, error recovery, addressing and routing, flow control, data encryption etc., can be solved step by step in a disciplined manner. From a designer's point of view, though, it is not predetermined that every design problem is always best subdivided as suggested in Figure 2.7. The specifics of the protocol system and the environment in which it is executed determine how a design problem can best be decomposed into smaller problems.

2.5 VOCABULARY AND FORMAT

We first look, on a fairly low level of abstraction, at some protocol formatting methods. These formats must underly all higher-level structures, for example, the structures that are used to encode the protocol message vocabulary. The three main formatting methods are:

- Bit oriented
- Character oriented
- Byte-count oriented

BIT ORIENTED

A bit-oriented protocol transmits data as a stream of bits. To allow a receiver to recognize where a message (a frame) starts and ends in the bit stream, a small set of unique bit patterns, or *flags*, is used. Of course, these bit patterns can be part of the user data too, so something has to be done to ensure that they are always interpreted properly. If a framing flag, for instance, is defined as a series of six one bits enclosed in zeros, `01111110`, series of six adjacent ones in the user data must be intercepted. This can be done by inserting an extra zero after every series of five ones in the user data.

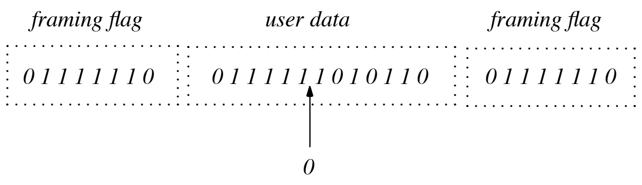

Figure 2.10 — Bit Stuffing

The receiver can now correctly detect the structure enforced by the flags in the bit stream by inspecting the first bit after every series of five ones: if it is a zero it must be deleted, else the pattern being scanned must be part of a true frame delimiter. This *bit stuffing* technique is used in ISO's layer 2 protocol (see Section 2.6) for *High Level Data Link Control*, HDLC, which is in turn based on IBM's *Synchronous Data Link Control* protocol, SDLC. Once the basic low-level flag structure is in place, it can be used to support higher-level structures.

CHARACTER ORIENTED

In a character-oriented protocol some minimal structure is enforced on the bit stream. If the number of bits per character is fixed to *n* bits (typically 7 or 8), all communication takes place in multiples of *n* bits. These data units are then used to encode both user data and control codes. Examples of control codes are the ASCII[2] start of text *STX* and end of text *ETX* messages that can serve as delimiters and can be used to enclose the user data.

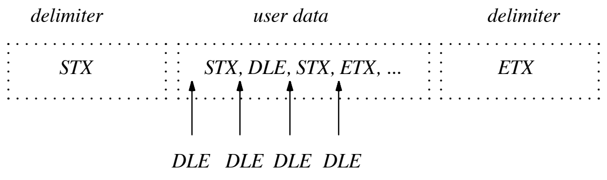

Figure 2.11 — Character Stuffing

Again, if raw data are transmitted (for example, binary object code), care must be taken that the delimiters do not accidentally occur in the user data. In IBM's *Bisync* protocol, for instance, every control character, such as *STX* and *ETX*, is preceded by an extra code, the data link escape character *DLE*. If any control message, such as *STX*, *ETX*, or even *DLE* itself, happens to occur literally in the user data, it is preceded by an extra *DLE* character. The *DLE* code is interpreted by the receiver as a control code that turns off any special meaning of the first character that follows it. The receiver deletes the first *DLE* code that it sees in the character stream, and passes on the following character uninterpreted. Only if the special meaning of an *STX* or *ETX* code is not suppressed by a preceding *DLE* character is it interpreted as a delimiter.

2. American Standard Code for Information Interchange.

The technique is called *character stuffing*.

Figure 2.11 shows where the *DLE* codes would be inserted in a stream that consists of four subsequent control characters in the user data.

BYTE-COUNT ORIENTED

The flags of a bit-oriented protocol and the control characters of a character-oriented protocol are used to structure a raw data stream into larger fragments. One reason for such structuring is to indicate to a receiver where a data stream begins and ends. In byte-count oriented protocols a slightly different method is chosen. In a known place after the *STX* control message, the sender includes the precise number of bytes (characters) that the message contains. An *ETX* message is now superfluous, and techniques such as bit stuffing or character stuffing are no longer needed. Most protocols in use today are of this type. A specific example is DEC's *Digital Data Communication Message Protocol*, DDCMP.

HEADERS AND TRAILERS

With the basic structuring methods we have discussed above, more systematic higher-level data formatting methods can be built. So far we have silently assumed the absence of transmission errors. If a byte-count field is distorted, or a *DLE* character is lost, these techniques fail. In the absence of an error detection and error recovery strategy, therefore, the techniques are of little use.

As we will see in more detail in Chapter 3, error detection schemes require transmission of redundant information, typically in the form of a *checksum*. If flow control techniques are added, for instance to detect loss or reordering of text frames, a *sequence number* field is appended. If more than one type of message is used we further have to include an indication of the type of message being transferred. And then, if we are transmitting redundant information anyway we might as well add other potentially useful data such as the name of the sender or the priority of the message.

All this overhead is most conveniently grouped into separate structures that encapsulate the user data: a mere *STX* control message thus expands into a *header* structure, and similarly the simple *ETX* grows into a composite message *trailer*.

For obvious reasons, byte counts are typically placed in message headers and checksums are placed in the trailer. The message format may then be defined as an ordered set of three elements:

format = { *header, data, trailer* }.

The header and trailer again define ordered subsets of control fields, which may be defined as follows:

header = { *type, destination, sequence number, count* },
trailer = { *checksum, return address* }.

The length of the data field is defined by the last field in the header. The destination and the return address can again be defined by substructures.

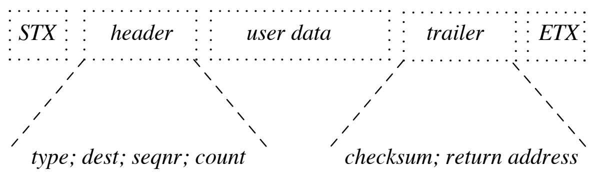

Figure 2.12 — Message Format

The *type* field can be used to identify the messages that make up the protocol vocabulary. Depending on the particular structure of the protocol vocabulary, this field can be refined still further.

2.6 PROCEDURE RULES

Up to this point, we have stressed the similarity of the protocol design task and normal software development. It is time to look at one of the differences. An important aspect of the protocol design problem is that the procedure rules are interpreted concurrently by a number of interacting processes. The effect of each new rule we add to the set is often much larger than can be foreseen. Many different interleavings in time of the interpretation of these rules by the various processes will be possible. Precisely because of this concurrency a protocol behavior is not always reproducible. To convince ourselves of the correctness of a design we need something better than informal reasoning. The most popular tool for reasoning about protocols, unfortunately, is the time sequence diagram, like the one used in Figure 2.4. To be sure, the time sequence diagram is convenient for reporting a single known error. But it is woefully inadequate for reasoning about the working of a protocol in general. To allow this we must, at least, be able to express behavior unambiguously in a convenient formal notation. Transition tables, or formal finite state machines (see Chapter 8), for instance, can be used for this purpose. In addition, we must be able to express arbitrary correctness requirements on the behaviors that we specify (see Chapters 5 and 6).

There is no general methodology that can guarantee *a priori* the design of an unambiguous set of procedure rules (we discuss this in more detail in Chapter 10). There are, however, tools with which we can, even automatically, verify the logical consistency of the rules (see Chapter 11) and the observance of the correctness requirements. And, of course, there is common sense and plain good engineering practice that can help us to keep the protocol rules manageable. We look at some of those issues in the next section.

2.7 STRUCTURED PROTOCOL DESIGN

Protocol design touches on a broad range of issues. Some of these issues are well understood; others we are only beginning to understand. Protocol design is partly an engineering problem that can be addressed by the application of well-known techniques. At the physical layer of the ISO hierarchy, for instance, we know precisely what the characteristic behaviors of different types of information carriers are, how fast we can transmit data on them and what the resulting average bit error rate will be.

There are various techniques for encoding binary data into the analog signals that can be carried by the various media, and there are well-known techniques for synchronizing transmitters and receivers at this level. We do not have to reinvent and revalidate those techniques for each new protocol, and indeed they can be considered so standard that we need not discuss them in this book. For the interested reader, the details are included in Appendix A.

Much higher up in the protocol hierarchy, we face problems of network design: routing data through networks, the precise dimensioning of network structures, the interconnection of multiple networks with gateways, and the development of higher-level disciplines for congestion control and congestion avoidance. In between this high level *network view* and the low level view of transmission codes and data carriers there is a large unknown territory, where there are few techniques that can help us through the design process. There is still a range of well-known error control and flow control techniques that can be used to build reliable data links, but this is only where real protocol design problem begins: the actual problem of devising unambiguous and complete sets of rules for the exchange of information in a distributed system.

Before this ‘‘gray area’’ of protocol design can become a true engineering discipline, it has to be established what the principal design tools are, what rules are to be followed, and what mistakes are to be avoided.

The development of a new engineering discipline often happens in two phases. In the first phase, the new technology is explored, and the designers seek tools that restrict them as little as possible in their exploration of its possibilities. If difficulties are encountered the capability of the tools is *expanded* to allow the user to cope with the growing set of problems. The trend in this first phase, then, is to remove constraints rather than to impose them.

In the second phase, after a better understanding of the nature of the problems develops, a new set of tools appears. These tools deliberately impose a carefully selected set of *constraints* upon the user. These constraints are meant to enforce a design discipline that is based upon the history of mistakes, collectively called ‘‘experience,’’ from the first development phase. In protocol design we are still waiting to make the transition to the second phase of development. Below we discuss some central concepts in the new design discipline for protocols that is emerging.

A designer will adhere to the discipline only if in return, provably and reproducibly, a more reliable product can be obtained. Below we give an overview of what is likely to become part of a general set of principles of sound design, which will allow us to enter the second phase of development in the field of protocol engineering. It is important to recognize that all these notes are variations on two common themes: simplicity and modularity.

SIMPLICITY — THE CASE FOR LIGHT-WEIGHT PROTOCOLS

A well-structured protocol can be built from a small number of well-designed and well-understood pieces. Each piece performs one function and performs it well. To understand the working of the protocol it should suffice to understand the working of the pieces from which it is constructed and the way in which they interact. Protocols that are designed in this way are easier to understand and easier to implement efficiently, and they are more likely to be verifiable and maintainable. A light-weight protocol is simple, robust, and efficient. The case for light-weight protocols directly supports the argument that efficiency and verifiability are not orthogonal, but complementary concerns.

MODULARITY — A HIERARCHY OF FUNCTIONS

A protocol that performs a complex function can be built from smaller pieces that interact in a well-defined and simple way. Each smaller piece is a light-weight protocol that can be separately developed, verified, implemented, and maintained. Orthogonal functions are not mixed; they are designed as independent entities. The individual modules make no assumptions about each other's working, or even presence. Error control and flow control, for instance, are orthogonal functions. They are best solved by separate light-weight modules that are completely unaware of each other's existence. They make no assumptions about the data stream other than what is strictly necessary to perform their function. An error-correction scheme should make no assumptions about the operating system, physical addresses, data encoding methods, line speeds, or time of day. Those concerns, should they exist, are placed in other modules, specifically optimized for that purpose. The resulting protocol structure is open, extendible, and rearrangeable without affecting the proper working of the individual components.

WELL-FORMED PROTOCOLS

A *well-formed* protocol is not *over-specified*, that is, it does not contain any unreachable or unexecutable code. A well-formed protocol is also not *under-specified* or incomplete. An incompletely specified protocol, for instance, may cause *unspecified receptions* during its execution. An unspecified reception occurs if a message arrives when the receiver does not expect it or cannot respond to it.

A well-formed protocol is *bounded*: it cannot overflow known system limits, such as the limited capacity of message queues.

A well-formed protocol is *self-stabilizing*. If a transient error arbitrarily changes the protocol state, a self-stabilizing protocol always returns to a desirable state within a finite number of transitions, and resumes normal operation. Similarly, if such a protocol is started in an arbitrary system state, it always reaches one of the intended states within finite time.

A well-formed protocol, finally, is *self-adapting*. It can adapt, for instance, the rate at which data are sent to the rate at which the data links can transfer them, and to the rate at which the receiver can consume them. A *rate control* method, for instance, can be used to change either the speed of a data transmission or its volume.

ROBUSTNESS

As Polybius (Chapter 1) noted,

> *''it is chiefly unexpected occurrences which require instant consideration and help.''*

It is not difficult to design protocols that work under normal circumstances. It is the unexpected that challenges them. It means that the protocol must be prepared to deal appropriately with every feasible action and with every possible sequence of actions under all possible conditions. The protocol should make only minimal assumptions about its environment to avoid dependencies on particular features that could change. Many link-level protocols that were designed in the 1970s, for instance, no longer work properly if they are used on very high speed data lines (in the Gigabits/sec range). A robust design automatically scales up with new technology, without requiring fundamental changes. The best form of robustness, then, is not *over-design* by adding functionality for anticipated new conditions, but *minimal* design by removing non-essential assumptions that could prevent adaption to unanticipated conditions.

CONSISTENCY

There are some standard and dreaded ways in which protocols can fail. We list three of the more important ones.

- □ *Deadlocks* — states in which no further protocol execution is possible, for instance because all protocol processes are waiting for conditions that can never be fulfilled.
- □ *Livelocks* — execution sequences that can be repeated indefinitely often without ever making effective progress.
- □ *Improper terminations* — the completion of a protocol execution without satisfying the proper termination conditions.

In general, the observance of these criteria cannot be verified by a manual inspection of the protocol specification. More powerful tools are needed to prevent or detect them. Such tools are discussed in Part III.

2.8 TEN RULES OF DESIGN

The principles discussed above lead to ten basic rules of protocol design.

1. Make sure that the problem is well-defined. All design criteria, requirements and constraints, should be enumerated before a design is started.
2. Define the service to be performed at every level of abstraction before deciding which structures should be used to realize these services (*what* comes before *how*).
3. Design *external* functionality before *internal* functionality. First consider the solution as a black-box and decide how it should interact with its environment. Then decide how the black-box can internally be organized. Likely it consists of smaller black-boxes that can be refined in a similar fashion.
4. Keep it simple. Fancy protocols are buggier than simple ones; they are harder to implement, harder to verify, and often less efficient. There are few truly complex

problems in protocol design. Problems that appear complex are often just simple problems huddled together. Our job as designers is to identify the simpler problems, separate them, and then solve them individually.

5. Do not connect what is independent. Separate orthogonal concerns.

6. Do not introduce what is immaterial. Do not restrict what is irrelevant. A good design is ''open-ended,'' i.e., easily extendible. A good design solves a class of problems rather than a single instance.

7. Before implementing a design, build a high-level prototype (Chapters 5 and 6) and verify that the design criteria are met (Chapters 11 to 14).

8. Implement the design, measure its performance, and if necessary, optimize it.

9. Check that the final optimized implementation is equivalent to the high-level design that was verified (Chapter 9).

10. Don't skip Rules 1 to 7.

The most frequently violated rule, clearly, is Rule 10.

2.9 SUMMARY

A protocol includes more than an agreement on the meaning of signals for data. Minimally, the protocol must include agreements on the use of control information, which is needed to standardize the use of the channel itself.

To be complete, the definition of a protocol should include the five main elements listed in Section 2.2. Protocol failures are often caused by hidden assumptions about events or about the possible sequences of events. It is the responsibility of the protocol designer to make these assumptions explicit. Again: it is not sufficient if a correct interpretation of the specification is merely possible. It is required that no incorrect interpretation is possible.

The main protocol structuring techniques are the layering of control software and the structuring of data. The OSI model is given as an example of this approach. Beware, it is not a recipe. Similarly, the ten rules of design are guidelines, not commandments. A structured and sound approach to the design of consistent procedure rules must always be a self-imposed discipline.

In the next two chapters we first cover the basics of protocol design, the known techniques for building reliable channels out of unreliable ones. The remainder of the book is devoted to the study of the protocol design problem itself. It does not discuss network design issues, nor the specific encoding or usage of the protocol standards that are in use today. Instead, our goal is to discuss how protocols can be designed using a simple discipline based on the rules given above.

EXERCISES

2-1. Identify the five protocol elements from Section 2.2 for the torch telegraph of Polybius, discussed in Chapter 1. List at least three cases of incompleteness in the protocol. □

2-2. Give an informal description of the procedure rules of a protocol that manages the data transfer from a file server to a printer (Section 2.1). Make sure that the protocol can recover when the printer runs out of paper or is switched off line. □

2-3. Explain what the equivalents of control and data messages are in a telephone call. Write down a complete (Section 2.2) protocol specification for a phone call, taking into account all possible signals and exception conditions. Consider the case where two people try to call each other simultaneously and consider the best procedure rules for redialing after a busy signal. □

2-4. Extend Lynch's protocol to avoid the duplication error, and show with a rigorous argument that the revised version works. □

2-5. Explain why a byte count is most conveniently placed in a message header (Section 2.5). □

2-6. Explain the difference between bit stuffing and character stuffing. □

2-7. Calculate the optimal length for a framing flag in a bit oriented protocol. Note that a longer series of ones in the framing flag reduces the probability of its occurrence in the user data and thus the overhead in the number of stuffed bits, at the expense of a higher overhead in the framing flag itself. Assume random user data. (See Bertsekas and Gallager [1987, p. 78-79]). □

2-8. In your favorite programming language, write a function that performs *STX — ETX* framing and character stuffing on an arbitrary byte stream. Provide the matching receive function and test it. □

BIBLIOGRAPHIC NOTES

That control messages are vital to a reliable operation of communication lines was already known in the days of the pre-electric telegraphs. Even the torch telegraph had a *start of text* message, and most later systems had at least special control codes for *repeat* and *wait*. The same control signals are defined on nearly every data link in use today. Hubbard [1965], reports yet another type of control message, devised by ''an anonymous French inventor'' for an early electro-static telegraph system. He suggested using the static charge of the telegraph line to ignite a small amount of gunpowder in the receiving station to *wake* a sleeping attendant.

The system described by Marland [1964] wins the prize for the best control messages ever devised. It noted a telegraphic system that was described in the *Mechanics' Magazine* of June 11, 1825 (Vol. IV, p.148). In this system the electro-static shocks are administered directly to the operator. And, if that is not enough, it suggests a most original solution to the problem of a drowsy telegraph operator:

> *''Let the first shock pass through his elbows, then he will be quite awake to attend the second.''*

Excellent introductions to the problems of protocol design can be found in Pouzin and Zimmerman [1978] and in Merlin [1979]. The formalism for describing protocols as an abstract language, with vocabulary, formal grammar, and syntax was introduced in Puzman and Porizek [1980].

Perhaps the greatest importance of the paper by Lynch [1968] is that it sparked a famous paper by Bartlett, Scantlebury, and Wilkinson from the National Physical Laboratory in England, defining one of the simplest and best known protocols in use today: *the alternating bit protocol*. We discuss it in Chapter 4.

The symbols used in the flow chart in Figure 2.3 are from the CCITT specification language SDL. The language is quickly gaining popularity as a specification method for communication protocols. For an overview see, for instance, Rockstrom and Saracco [1982] and SDL [1987]. The official SDL language definition is in CCITT [1988]. The flow charting ''language'' used here is more fully described in Appendix B. The best reference to the C language, referred to briefly in Section 2.3, is Kernighan and Ritchie [1978, 1988].

The principal ideas of structured programming and software layering stem from E.W. Dijkstra [1968a, 1968b, 1969a, 1969b, 1972, 1976] and N. Wirth [1971, 1974]. They are closely related to the technique of design by stepwise refinement Wirth [1971], see also Gouda [1983]. That the principle of stepwise refinement was known long before program design became an issue is illustrated by the following quote from E.F. Moore.

> *''One way of describing what engineers do in designing actual machines is to say that they start with an overall description of a machine and break it down successively into smaller and smaller machines, until the individual relays or vacuum tubes are ultimately reached.''* (Moore [1956])

The ideas on protocol design expressed here are also inspired by discussions with many others, most notably Jon Bentley, John Chaves, Peter van Eijk, Rob Pike, and Chris Vissers. The importance of the service concept in protocol design is eloquently explained in Vissers and Logrippo [1985].

The term *self-stabilization* was also coined by Dijkstra, see for example, [1974, 1986], see also Kruijer [1979]. Lamport discussed self-stabilization in several papers, Lamport [1984, 1986]. Multari wrote his thesis on self-stabilizing protocols, Multari [1989]. Other pioneering work in this area is done at the University of Texas at Austin by M.G. Gouda [1987] and at Cornell University by G.M. Brown [1989].

The study of light-weight protocols was pioneered in the 1970s by a research group at the Computer Laboratory of the University of Cambridge, involved with the design of the Cambridge Ring Network, e.g., Needham and Herbert [1982], and a little later by a group at AT&T Bell Labs, including Sandy Fraser, Greg Chesson, and Bill Marshall, involved with the design of the hardware and software for the Datakit® switch.

The term *light-weight* protocol was coined by the Cambridge group, who also developed the first serious contender in this class: the *byte stream protocol* that is used on the Cambridge Ring. The work at Bell Labs led ultimately to the design of the standard Universal Receiver Protocol (URP), Fraser and Marshall [1989], and its successors the PSP and MSP packet switch protocols.

A complete description of the OSI model can be found in ISO [1979]. The X.25

protocol, finally, is documented in CCITT [1977] and explained in, for instance, Lindgren [1987], and Stallings et al. [1988]. More about data networking problems can be found in Tanenbaum [1981, 1988] or in Stallings [1985].