

2 Protocols

In this chapter we consider how communication services are provided. For this, we look inside the service provider. We introduce the notion of protocol and explain the relations which exist between protocol procedures and the interactions at the service interface. Next we describe the basic elements of communication protocols and show how they can be described. Finally we continue introducing the XDT protocol.

2.1 Principles

Entities

Services are provided by the service provider using interacting entities (see [Figure 2.1/1](#)). **Entities** are active objects of the service provider that communicate with their environment by exchanging messages. The internal structure of the entities is not relevant from the modeling point of view. We only consider their capability to interact with other entities and with the environment.

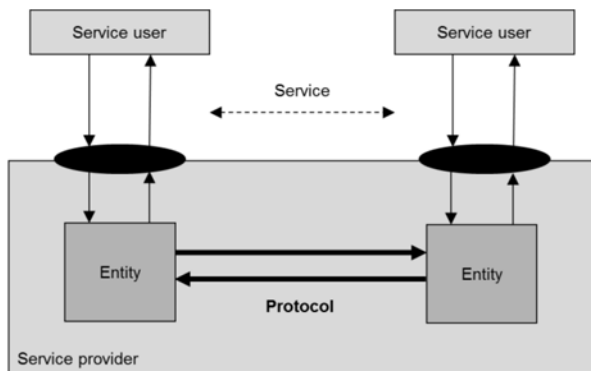


Figure 2.1/1: Service and protocol

The entities serve the service access points. A service access point is always assigned to one entity only, while an entity can simultaneously handle several access points (see [Figure 2.1/2](#)). The entities read the service primitives handed over at the service access points and analyze them. According to the address information transported in the service primitives they start to interact with the entity which serves the related service access point to provide the service. This entity is called a **peer entity**. The working principle of an entity can be compared with a letter distribution centre, which fetches the letters from the mailboxes, reads the addresses,

and forwards the letters to the peer distribution centre, which delivers the letters to the receivers' mailboxes.

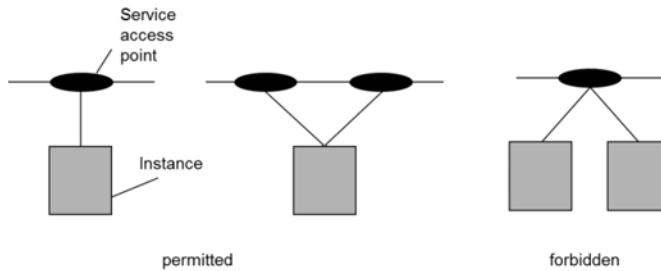


Figure 2.1/2: Relation between service access points and entities

Communication protocol

The interaction between peer entities follows firm rules. These rules are defined by a communication protocol or protocol, for short. A **communication protocol** is a behavior convention that defines the temporal order of the interactions between the peer entities as well as the format (syntax and semantics) of the messages exchanged.

In conversational language the term protocol possesses two meanings. It either denotes the minutes of a meeting, or it describes a firm procedure, i.e., a convention or a ritual. A typical example of the second case is a diplomatic protocol. The term protocol in computer networks corresponds to the latter meaning. It defines a communication procedure that iterates. Note that a protocol is not a dialogue, because a dialogue has mostly an unpredictable course that cannot be repeated.

Two kinds of protocols are distinguished: symmetric and asymmetric ones. A protocol is called **symmetric** if the communication behavior of the two entities is equal. This is given for protocols with duplex data exchange, i.e., a simultaneous data exchange in both directions. Most protocols deployed in computer networks are symmetric. In **asymmetric** protocols the behavior of both entities is different. Protocols that only support data transmission in one direction (simplex transmission) are usually asymmetric ones.

In order to graphically represent protocol interactions, time sequence diagrams are used. In contrast to the description of services, the interaction between the entities is now described by the messages exchanged. Figure 2.1/3 shows as examples the time sequence diagrams for a successful and a rejected connection set up. They include both the service primitives and the exchanged messages.

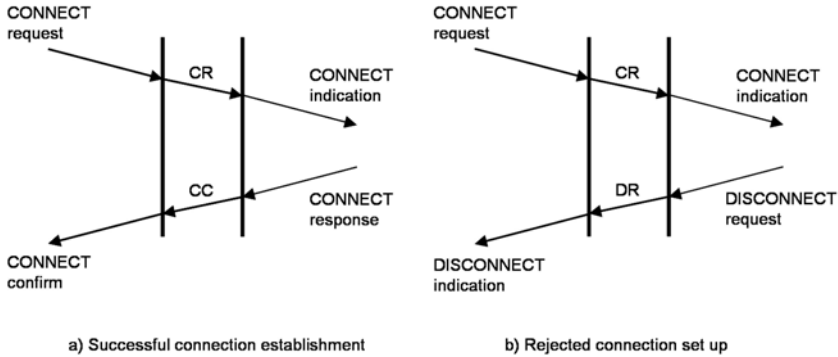


Figure 2.1/3: Protocol sequences as time sequence diagrams

Protocol data units

The messages that are exchanged between peer entities are called **protocol data units (PDUs)**. Their format, i.e., the structure and the semantics of the components, is defined by the protocol, which is known to both entities to ensure that the protocol data units are equally interpreted. A protocol typically uses several protocol data units, e.g., for connection set up, data transfer, and connection release. They are distinguished by their names and different internal code tags.

Principle of transparency

The protocol data units transport the user data which have been handed over to the service provider through service primitives. The user data are not accessible to the service provider, similarly to the context of a letter, which is not accessible to the postman, i.e., these data are transparent for the service provider. They must be delivered unchanged to the receiver. This is called the **principle of transparency**. Of course, the service provider could access these data, but what this principle exactly means is that the service provider should not use these data for controlling the protocol procedure. The principle of transparency forms the basis for the “tunneling” technique applied in the Internet, when protocol data pass through a network with another protocol architecture.

To implement the principle of transparency the user data, called **service data units (SDUs)**, are supplemented by protocol control fields, known as **protocol control information (PCI)**. The protocol control information and the service data unit form the protocol data unit (see Figure 2.1/4). Protocol control fields may precede and/or succeed the service data unit as *PDU header* and/or *trailer*. Most protocols only use a PDU header because it is easier to analyze. Typical protocol control data are, for example, the source and destination address, the length of the PDU, the PDU tag, quality of service (QoS) parameters, and the frame check sequence. The protocol control fields are removed at receiver side before the service data units are delivered to the service user.

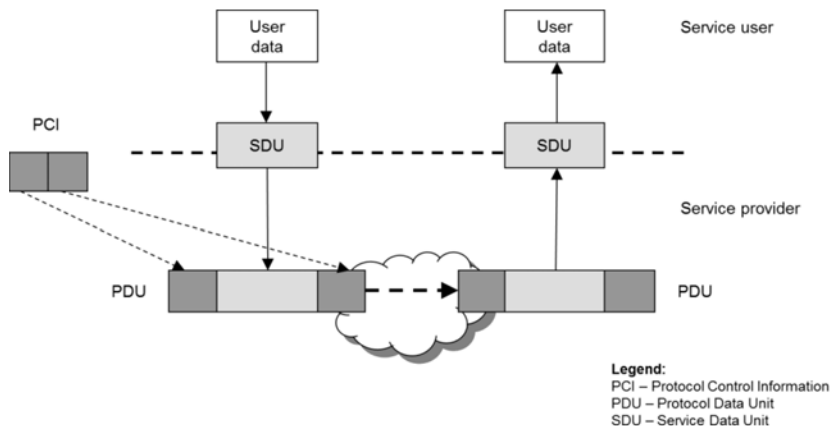


Figure 2.1/4: Principle of transparency

Protocol functions

Protocol entities contain various procedures and mechanisms which are repeatedly used in different protocols. These protocol procedures are called **protocol functions**. A typical protocol function is the error control which defines, for instance, the reactions of the protocol entities when a PDU has not been transmitted successfully. Other examples are the fragmentation of PDUs or the flow control, which controls the speed of the data exchange between the sender and receiver entities. We will comprehensively introduce the most important protocol functions in Chapter 5.

Connections

In connection-oriented protocols the peer entities have to manage the connections that are established between the service users at the service access points. The same protocol runs over each connection. The protocols are executed concurrently and do not influence each other. Connections are identified by **connection references**, which are only used as local context information. The connection references correspond to the connection end points of Section 1.1. The entities use **connection tables** (see Figure 2.1/5) to manage the connections and to store all information about them, e.g., the addresses of the access points, the state of the connection (*non-existent*, *establishing*, *existent*, *releasing*), and others. The connection reference can be used as table index.

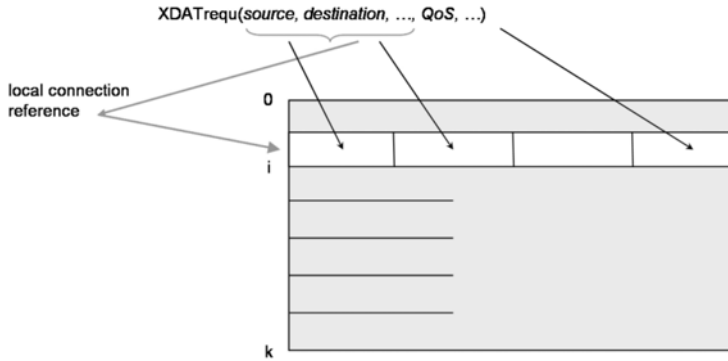


Figure 2.1/5: Connection table

Concurrency and nondeterminism

In the last chapter we discussed that concurrency and nondeterminism may appear at the service interface. Accordingly the protocol entities have to be capable of handling them. The entities must be able to simultaneously react to different communication demands and events. For example, during the coding of a PDU it can be indicated that the respective connection has been interrupted. In this case the entity has to take measures to re-establish the connection. These actions are performed **concurrently**. When several events appear simultaneously to the entity, it cannot be predicted in which order the entity handles them. It is even not prescribed, which event will be handled first and in which order the other events are handled. The selection is **nondeterministic**.

2.2 Description

2.2.1 Protocol specification

The manner in which entities interact with each other in a protocol is defined in the **protocol specification**. It describes the temporal order of interactions between the peer entities and defines the format of the messages exchanged. The protocol specification defines among others how the entities react to service primitives, incoming PDUs, or internal events. The protocol specification is basically the “implementation” of the service specification.

The protocol specification represents the reference document for the validation and the implementation of the protocol. From the service user’s point of view, it is not necessarily required to know the protocol specification. It is relevant for the protocol engineer. A protocol specification is always independent of a concrete implementation, i.e., it only prescribes the interactions between the entities, but it does not prescribe how the protocol is integrated into a certain execution environment, i.e., the operating system. Thus it is guaranteed that implementations for dif-

ferent execution environments can be derived from the same protocol specification.

In former ISO specifications the service and protocol descriptions were strictly separated. This does not apply to the *Request for Comments* (RFCs) of the IETF, in which services are mostly not explicitly described. As with service specifications, informal descriptions also prevail in protocol specifications. They consist of textual descriptions of the protocol procedures supplemented by tables, diagrams, and other semi-formal presentations.

2.2.2 Model language

We again demonstrate the principles of the protocol specification using our model language. As next language level, we now introduce **level P** ($P=Protocol$). It describes the interactions between the entities under the assumption of a virtual communication between them (see Figure 2.2/1). Virtual communication means here that it is not said at this abstraction level how the interaction between the peer entities is implemented.

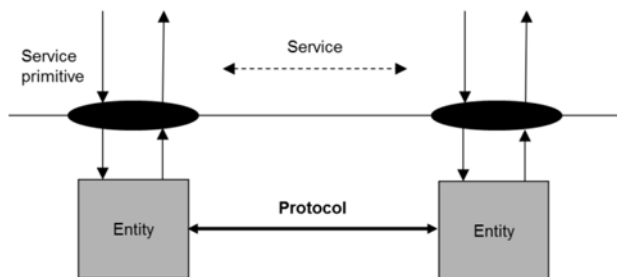


Figure 2.2/1: Description principle of level P

We start with the description of the PDU formats. Thereafter we consider the description of the protocol procedures.

2.2.3 Description of the PDU formats

The data units which are exchanged in a protocol form the “common language” of the communicating entities, which they have to interpret equally. Therefore their formats have to be specified unambiguously. The definition of the PDU formats is independent of the description of the protocol procedures. They are usually described separately.

The specification of the PDU formats is handled differently in the various formal description techniques. One of the earlier techniques, Estelle for instance, used PASCAL data types. This was very convenient for the user because PASCAL was pretty well-known at that time. But it introduced implementation dependencies into the description which might lead to different interpretations of

the data formats. Therefore other languages introduced their own data description. The algebraic language LOTOS incorporated the algebraic data type language ACT ONE. This proved a failure because the data format specification appeared expensive. SDL first defined its own notation based on algebraic representations of abstract data types. Later it was integrated the abstract syntax notation ASN.1, which is preferred for describing the data formats of communication protocols nowadays. We will introduce it in Section 8.5.

In our model language we use for readability reasons elements of modern programming languages for describing the data structures, as we already did for the description of the service primitives at level S. The PDU formats are described in the **message**-specification which follows the **service**-specification of level S:

```
specification name {
    service specification           // unchanged from level S
    message specification
    . . .
}
```

In the **message**-specification all protocol data units that appear in the protocol have to be listed, e.g.,

```
message {
    DT: struct(length: 0..255,           // Data PDU
        code: bits,
        source-addr: address optional,
        dest-addr: address optional,
        conn: integer optional,
        sequ: integer,
        eom: boolean,
        data: [] byte
    )
    ACK: struct (code: bits,             // Acknowledgment DT
        conn: integer,
        sequ: integer
    )
}
```

To describe the access to components of a protocol data unit, we use the dot-notation which is often applied in connection with **struct**- and **record**-constructs in high-level programming languages, e.g., DT.sequ or ACK.conn.

2.2.4 Description of protocol procedures

In general, there are two approaches to formally describe communication protocols: constructive and descriptive techniques (see Section 7.3). *Constructive*

techniques describe the protocol by means of an abstract model. This is in essence a quasi-implementation of the protocol on a more abstract level. *Descriptive techniques*, in contrast, formulate properties usually in logical calculi which the protocol to be designed is to fulfill. In practice most formal description techniques apply the constructive approach. Typical semantic models of constructive description techniques are finite state machines or labeled transition systems (see Section 7.4 and 7.7). Our model language also belongs to this type of description techniques.

Semantic model

For simplicity reasons, we do not introduce a formal semantic model for our model language here. We present the description principle applied to interpret specifications. The semantic model of the level P of our model language is based on extended final state machines (see Section 7.5). It refines the behavior description based on the **par event**-statement used in level S. In contrast to many finite state machine representations, which use a state-oriented description, we apply *a communication-oriented description method* which focuses on the representation of interactions between the peer entities (see Section 7.1). The basic unit of this description principle is the so-called protocol part (see [Figure 2.2/2](#)).

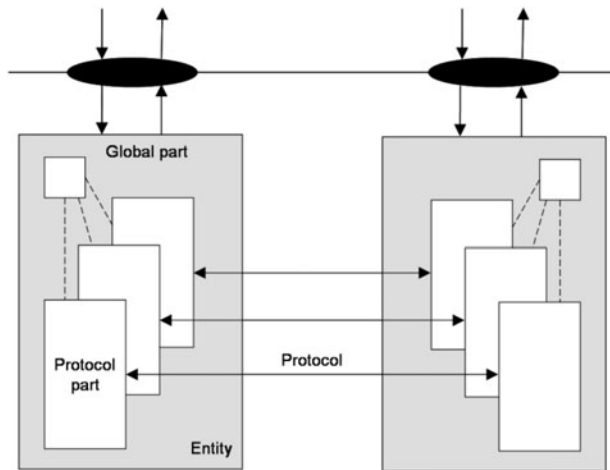


Figure 2.2/2: Protocol parts

A **protocol part** describes a dedicated communication behavior of an entity that is triggered by an external and/or an internal event. The reaction on this event is described sequentially using a procedure-like form. The protocol parts of the peer entities constitute the protocol. To illustrate the principle one can image a conversation. The phrases of each partner are recorded in a protocol part. When they are put together they present the conversation. At best a protocol consist of two protocol parts, those of the sender and receiver entities. In most cases, however, several protocol parts are needed. In symmetric protocols the entities consist of

the same protocol parts; in asymmetric protocols they have different ones. The protocol parts of an entity are executed concurrently. In certain situations coordination between different protocol parts may be required. This is done by exchanging signals.

Interactions between the entities are *asynchronous*. Each entity possesses an event queue in which the incoming events for all protocol parts are stored. Events are service primitives, PDUs, timeouts, signals, and internal events. Figure 2.2/3 depicts the description model of a protocol entity.

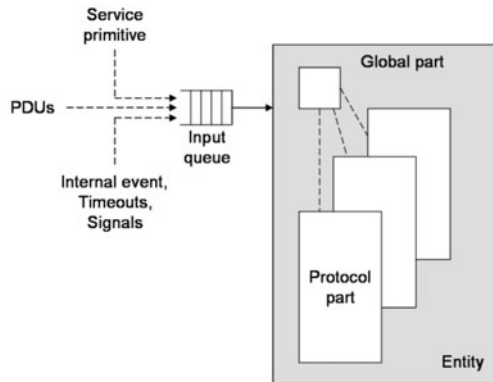


Figure 2.2/3: Description model of a protocol entity

Specifications and statements

In order to describe the protocol behavior we introduce two further specifications: the **protocol**- and the **entity**-specification. A level-P specification now has the following syntactical structure:

```

specification name{
  service specification                               // unchanged from level S
  message specification
  protocol specification
  entity specification1
  . . .
  entity specificationn
}

```

The **protocol-specification** declares which protocol parts constitute a protocol. A protocol may be divided into sub-protocols that run, for instance, in different phases. For each sub-protocol, the respective protocol parts have to be listed. The protocol parts are referred to by the name of the entity and the name of the part, e.g.,

protocol

connection set up: XS.connect_s \leftrightarrow XR.connect_r

data transfer: XS.transfer_s \leftrightarrow XS.ack_handler \leftrightarrow XR.connect_r

The **entity-specification** constitutes the main part of the level-P specification. It describes the communication behavior of the protocol entities. In symmetric protocols only one **entity-specification** is needed; in asymmetric protocols separate **entity-specifications** must be introduced for describing the sender and the receiver entity. The **entity-specification** consists of a specification and declaration part and an action part.

entity name

sap-specification

// Specification and declaration part

signal-declaration

var-declaration

timer-declaration

par event{

// Action part

... .

}.

The *specification and declaration part* begins with the **sap-specification**, which is mandatory. The **sap-specification** specifies the service access points which are served by the entity. The given access points must relate to a service access point of the level-S specification, e.g.,

sap sender.

In the **signal-declaration**, which is optional like the other declarations, the signals must be declared which are exchanged internally between protocol parts, e.g.,

signal break, credit.

The signals indicate the occurrence of certain events in the respective protocol part, e.g., the change of the protocol phase or the arrival of a new credit for flow control. Internal events are also represented by signals. It is not described how these signals are set.

The **var-declaration** is used for introducing variables like in programming languages. Each variable has a data type. Here the same data types are allowed as introduced in the level-S specification. A variable may be initialized by indicating an initial value after **init**, e.g.,

var i, j: integer,

sequ: integer **init**(1),

last: boolean **init**(false)

The declared variables are valid for all protocol parts of the entity. They can be used to exchange data between protocol parts. Access to the variables is assumed to be exclusive, i.e., only one protocol part can read a variable or assign a new value at a given instant.

The **timer declaration** contains the timers used in the entity to detect message losses. The timers are also globally declared for the entity. For each timer, the time interval is specified including the used time unit (*ms*, *s*, *min*, *h*), e.g.,

```
timer t1: 0..100 ms,
      t2: 0..? s.
```

If no upper bound is to be specified at this specification level a question mark is indicated instead. The role of timers in protocols is discussed in detail below.

The *action part* of an entity is described through a **par event**-statement (cp. Section 1.2.4). It describes the parallel execution of the protocol parts. The invocation of a protocol part is described by the triggering event (service primitive, PDU, timeout, signal, and others) followed by the name of the protocol part, e.g.,

```
par event{
  XDATrequ: connect_s ||
  data transfer: transfer_s ||
  run_ah: ack_handler
}
```

The protocol parts are represented by their name which acts like a call. A protocol part may have parameters (see below). For readability reasons, protocol parts are separately described. They follow the main specification **spec**{ ... }.

A protocol part is activated as follows. An entity reads the first event from its input queue and checks whether this event is an expected one. Expected events are those events that are awaited in a **par event**- or **wait event**-statement. If the event is awaited then the respective protocol part or the respective reaction in the **wait event**-statement is executed, otherwise the event is discarded. An event cannot be assigned if the respective protocol part is currently in execution. In that case it is delayed until the protocol part terminates. If the input queue is empty the entity waits for new incoming events.

The description of the **protocol parts** consists like the **entity**-specification of a declaration and an action part:

```
protocol part name (peer entity)
signal-declaration                                     // Declaration part
var-declaration
timer-declaration
event: begin                                           // Action part
      statements
end
```

In the header of the protocol part after the key word **peer** the peer entity (or entities) is specified. This is a formal parameter that represents the communication partner. It must be replaced by the name of the entity when the protocol part is invoked. In symmetric protocols the peer entity can be omitted.

The declaration part corresponds to that of the **entity**-specification. All three declarations are optional. In the **signal**-declaration all signals must be listed that are expected in this protocol part. Variables and timers declared in a protocol part are only valid within that protocol part.

The action part of a protocol part consists of the triggering event and the corresponding reaction. A triggering event may be a service primitive, a PDU, a timeout, or a signal. The event must be the same as specified in the action part of the respective **entity**-specification. The reaction, embedded in **begin ... end**, consists of one or several statements which are executed sequentially. We introduce these statements next. Some of them are already known from the S-specification.

Interactions with the peer entity are described by the **send-statement**, e.g.,

DT → receiver.

It describes the sending of a message to the peer entity. The message must be specified in the **message**-specification, the name of the receiver entity correspondingly in the header of the protocol part. The receiving of a message is correspondingly represented by an arrow in the reverse direction, e.g.,

ACK ← receiver.

The receiving of a PDU represents an event, the so-called *receive event*. It may appear as triggering event of a protocol part or in a **wait event**-statement.

The sending of a service primitive to the service user is described by the **respond-statement**, e.g.,

respond XDATrequ.

The waiting of the entity in a protocol part is described by the **wait event**-statement, which was introduced in Section 1.2.3. Triggering events may be: service primitives, PDUs, time-outs, and signals. As in the service specification, awaited events may be connected to an additional condition to express context dependencies, e.g.,

```
wait event{
    DT ← sender and DT.sequ = 1: . . .
}
```

In this case, the respective alternative is only selected if also the additional condition is fulfilled.

The sending of a signal to another protocol part is described by means of a **set-statement**, e.g.,

set break.

The **set**-statement causes the signal to be written into the event queue like any other event. It has now the value *true*. After the signal has been read out and triggered some reaction, its value is set to *false* again. The change of protocol phases can be handled by signals, if this is taken into account in the specification.

Timers

Timers are used in protocols to monitor communication procedures to avoid waiting indefinitely for certain events. If an event does not occur in a defined time interval, a so-called time-out is triggered. A **time-out** is handled as an event that triggers an alternative reaction to the awaited one, e.g., an error procedure. This prevents the entity from running into a deadlock state.

The representation of time and the declaration of timers are differently handled in formal description techniques. Some languages do not or only partially support time, others do support it. In our model language we assume a continuous progress of time. Timers are declared in the **timer**-declaration introduced in the preceding section, e.g.,

```
timer t1: 0..100 ms,
      t2: 0..? s.
```

The upper value of the time interval defines the time-out. Different time units can be used: *h*, *min*, *s*, and *ms*. Timers that are declared in an **entity**-specification are valid in all protocol parts, whereas timers declared in a protocol part are only valid locally in this part.

Timers are started by means of a **start-statement**, e.g.,

start t.

A running timer may be stopped using the **reset-statement**, e.g.,

reset t.

If a **start**-statement is applied on a running timer, the timer is implicitly reset first. A time-out is indicated by a *timeout* event in a **par** or **wait event**-statement, e.g.,

```
start t
wait event{
    DT ← sender: reset t           // awaited event
    . . . |
    timeout t: reaction           // alternative reaction after time-out
}
```

A time-out causes the *timeout* event to enter the event queue (cp. Figure 2.2/3). When it is an awaited event, it is read-out, otherwise it remains in the input queue and might be selected if the **par** and **wait event**-statements are executed again. To avoid any trouble from this it is recommended to reset timers when an awaited event is processed, as is done in the example above. The **reset**-statement does not just stop the timer. It also removes the *timeout* event from the input queue.

Local actions

Protocol specifications describe the external behavior of communicating entities, i.e., their interactions. To run a protocol actions are also needed which are only of local importance, e.g., the coding/decoding of PDUs, the analysis of the received values, cyclic redundancy checks, start and reset of timers, and others. We call these actions **local actions** in the following. To what extent local actions are considered in a protocol specification depends on the abstraction level of the applied description technique. To represent and to verify the protocol flow they are unlikely to be required because mainly the interactions between the entities are of interest here. Local actions, however, are required for the implementation of the protocol. Therefore, more abstract formal description techniques, such as LOTOS [ISO 8807], scarcely consider them, while less abstract techniques like SDL [ITU-T 100] do.

In our model language we describe local actions. We use statements for the description of the control flow which are known from high-level programming languages, such as the **if**-statement, the **case**-statement, the **loop**-statement, and the empty statement **skip**. Thereby loops are unlimited. They can be unconditionally left by means of **exit** or conditionally by specifying a condition after **exit when**. Increment and decrement statements (**incr**, **decr**) can be used for counting the number of loops. Moreover, we introduce the statement

exit *name*

to prematurely leave a protocol part. In exceptional cases it might be useful to cancel the protocol execution. This is expressed by

cancel protocol

It terminates all protocol parts, resets the variables and timers, and removes all events from the input queues.

Coding and Decoding of PDUs

The coding of PDUs at sender side and their decoding and analysis at receiver side are another example of local actions. They represent a large part of the protocol code and can take a considerable part of the protocol execution time. Coding/decoding is also differently handled in the various formal description techniques. Abstract techniques, such as LOTOS or Petri nets, do not consider them, whereas the coding/decoding can be described in more implementation-oriented techniques like SDL.

In our model language we also forego for readability reasons an explicit description of PDU coding/decoding. Instead we represent them by means of a pre-defined procedure *coding_PDU* using the concrete PDU names for *PDU*, e.g.,

```
coding_DT(data,sequ).
```

Parameters are the data that have to be coded. For simple PDUs, which do not transport user data, the coding is sometimes omitted. The decoding at receiver side is indirectly described by using selections, e.g.,

```
DT.data,
```

when PDU data are used in a **respond**- or another statement.

Informal descriptions

Informal descriptions do not belong in a formal description. Nevertheless they are allowed, for example, in SDL. Informal descriptions may be useful during the design phase to describe protocol steps the formal representation of which would not be useful or needed at this level of abstraction. In this sense informal descriptions are also used in our model language. We mainly describe local actions this way to avoid a too implementation-oriented representation. The same applies to the description of conditions in **if**-, **case**-, and **exit when**-statements.

Instantiations

Up to now we have only considered language features for describing the protocol behavior. Several formal description techniques, e.g., SDL, also provide the possibility to create instances of their description elements (modules, objects, and others). This allows, for example, in addition to the description of the protocol flow to describe the establishment of a connection. This might be useful for the prototyping of the specification. However, it often requires the explicit setting up of communication paths, e.g., channels, between these instances. This requires a lot of additional language features and rules which are actually not related to protocols, but may constitute a large part of the protocol description (see the SDL XDT specification in Section 8.1.4). In our model language we confine ourselves to the pure protocol description.

2.3 Example

We now continue the description of the XDT protocol started in Section 1.3.

2.3.1 XDT protocol

The XDT protocol provides a reliable transmission of a larger, but limited sequence of data units over an unreliable medium preserving the order in which the data units are sent. It uses the *go back N*-principle to retransmit lost PDUs (see Section 5.1). This method retransmits all PDUs starting from the last not confirmed PDU, even if some of them have already been successfully transmitted.

The data units handed over from the *sender* by means of an *XDATrequ* primitive are mapped on *DT*-PDUs (see Figure 2.3/1). Each *DT*-PDU possesses a sequence number. The successful transmission of a *DT*-PDU is confirmed by the receiver entity *XR* with an *ACK*-PDU which contains the sequence number of the PDU increased by 1. A correctly transmitted data unit is delivered to the *receiver* via an *XDATind* primitive. *DT*-PDUs that do not arrive in the correct order are discarded. No confirmation is sent. If the transmission order cannot be re-established within a time interval t the receiver entity *XR* aborts the transmission, sending an *ABO*-PDU to the sender entity *XS*. The service users at both sites are informed about this by an *XABORTind* primitive.

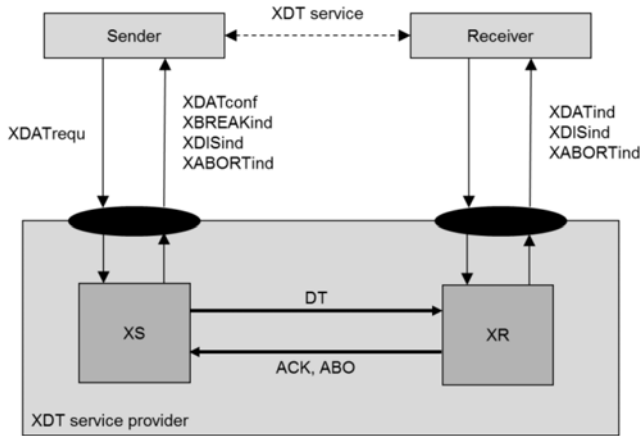


Figure 2.3/1: XDT service and protocol

The first *DT*-PDU also establishes the connection between the communication partners. The *sender* can only continue the data transmission when the connection set up is confirmed by an *XDATconf* primitive (2-way-handshake (see Section 5.2)). For simplicity reasons, we assume that the receiver always accepts a new connection. The connection set up is confirmed by the receiver entity *XR* through an *ACK*-PDU which contains the connection reference *conn* assigned by the receiver entity. This reference is indicated to the *sender* with the first *XDATconf* primitive. If the connection set up is not confirmed within a time period $t1$ it is aborted by an *XABORTind*. The possible protocol procedures for a connection set up are depicted as time sequence diagrams in Figure 2.3/2. The question mark in

Figure b) is to indicate that the reason for the abort remains unknown (loss of the *DT*- or the *ACK*-PDU).

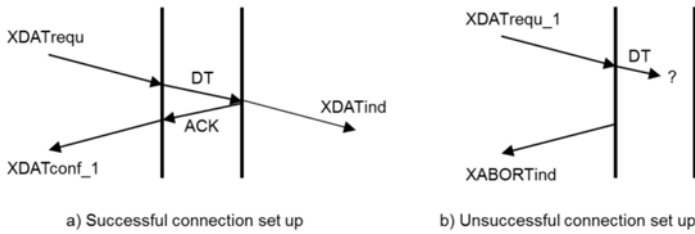


Figure 2.3/2: Possible protocol sequences for XDT connection set up phase

After establishing the connection, the other data units can be sent. They are handed over to the XDT service provider by *XDATrequ* primitives, where they are mapped on *DT*-PDUs. The sending of a *DT*-PDU is indicated to the sender by an *XDATconf*. Only then can the sender hand over a new data unit to the service provider. (Note that for simplicity reasons it is not proved in the protocol that this presumption is fulfilled by the sender.)

After sending the second *DT*-PDU, a timer t_2 is started that monitors the arrival of *ACK*-PDUs. It is restarted each time an *ACK*-PDU is received. When t_2 times out the *go back N*-procedure is called, i.e., all not acknowledged *DT*-PDUs are re-transmitted. During *go back N* no further *XDATrequ* primitives are accepted from the service provider. For the *go back N*-procedure, a copy of each *DT*-PDU is stored in a buffer. It is erased when the respective acknowledgement is received. The size of the buffer, however, is limited. When the buffer is full, *XS* indicates a break to the sender by means of an *XBREAKind* primitive. During this break no further *XDATrequ* primitives are accepted. The break terminates when an *ACK*-PDU of a stored *DT*-PDU arrives and its copy is removed from the buffer. The end of the break is indicated to the sender through the outstanding *XDATconf*.

The sender entity *XS* monitors the activity of the receiver entity *XR* using the activity timer t_1 . If no *ACK*-PDU arrives within a defined time interval the sender entity aborts the transmission and delivers an *XABORTind* primitive to the *sender*. This prevents a deadlock at sender side.

The connection is released after the successful transmission of the last *DT*-PDU. This is indicated to *sender* and *receiver* by an *XDISind* primitive. The last data unit is signaled by setting the parameter *eom true* in the last *XDATrequ* primitive and the last *DT*-PDU. Some of these protocol procedures are depicted in [Figure 2.3/3](#).

Note that the assumptions we made for the use of the XDT service in Section 1.3 also apply to the XDT protocol.

2.3.2 Formal description

The formal description of the XDT protocol at level P of our model language is given below. Since XDT is an asymmetric protocol, the sender entity and the receiver entity both have to be specified. The sender entity *XS* contains three protocol parts: *connect_s*, *transfer_s*, and *ack_handler* that describe the connection set up, the data transmission, and the reception of the acknowledgements, including the *go back N* procedure. The receiver entity *XR* possesses two protocol parts: *connect_r* and *transfer_r* which describe the connection set up and the data reception. The protocol parts *connect_s* and *connect_r* constitute the sub-protocol for the connection establishment, the protocol parts *transfer_s*, *ack_handler* and *connect_r* are the respective sub-protocol for the data transmission.

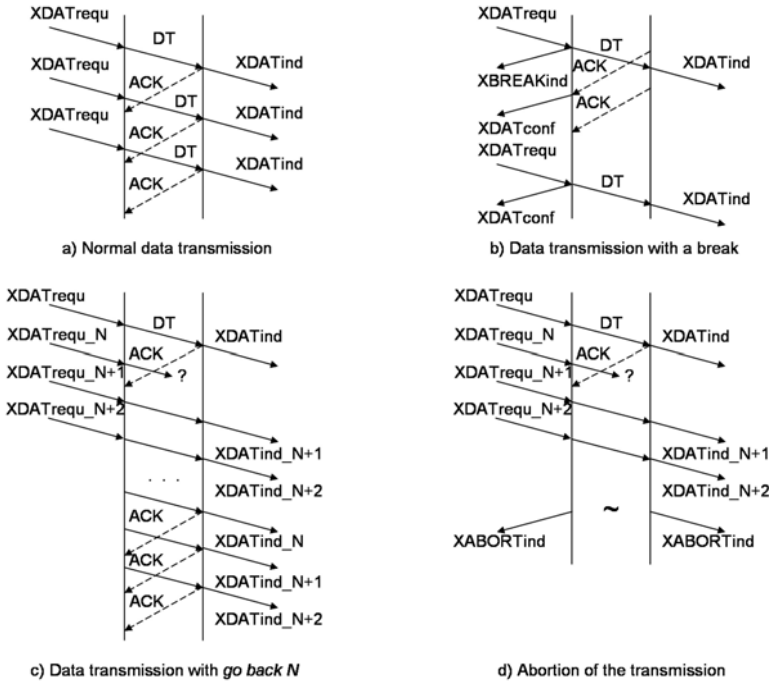


Figure 2.3/3: Possible protocol procedures during the data transmission phase¹

The different number of protocol parts is caused by the fact that the sending of *DT*-PDUs and the reception and analysis of the acknowledgements are activities that are independent of each other and can be executed concurrently. For that reason, they are represented in separate protocol parts.

¹ For clarity reasons, we abstain from representing the *XDATconf* primitive for confirming each *XDATrequ* primitive in figures a), c), and d).

```

specification XDT{
service                                     // XDT-S specification remains preserved
message
    DT: record(length: 0..255,                // Data PDU
               code: bits,
               source-addr: address optional,
               dest-addr: address optional,
               conn: integer optional,
               sequ: integer,
               eom: Boolean,
               data: array [] of byte)
    ACK: record(code: bits,                    // Acknowledgement PDU
               conn: integer,
               sequ: integer)
    ABO: record(code: bits,                    // Abort PDU
               conn: integer)
protocol                                     // Sub-protocols
    connection set up: XS.connect_s  $\leftrightarrow$  XR.connect_r
    data transfer: XS.transfer_s  $\leftrightarrow$  XS.ack_handler  $\leftrightarrow$  XR.connect_r
entity XS                                     // SENDER ENTITY
    sap sender                                 // associated SAP
    signal DATA TRANSFER                     // indicating data transmission
    var conn: integer,                         // Connection reference
        sequ: integer init(1),                // current sequence number
        last: integer init(0),                // Sequence number of last PDU
        buffer: array [1..m] of DT,           // DT-Buffer with upper limit m
        go_back_N: boolean init(false),       // true, if go back N is running
        break: boolean init(false)           // true, if break
    par event{
        XDATrequ
        and XDATrequ.sequ = 1: connect_s ||    // Connection set up
        XDATrequ
        and not go_back_N
        and not break: transfer_s ||           // Data transfer phase
        DATA TRANSFER: ack_handler           // ACK monitoring
    } // XS

entity XR                                     // RECEIVER ENTITY
    sap receiver                               // associated SAP
    signal DATA TRANSFER                     // indicating data transmission
    var conn: integer                         // Connection reference
    par event{
        DT  $\leftarrow$  sender and DT.sequ = 1: connect_r || // Connection set up
        DATA TRANSFER: transfer_r           // Data transfer phase
    }

```

```

    } // XR
} // XDT

```

The protocol parts *connect_s* and *connect_r* describe the connection set up at the sender and receiver side. They are the only protocol parts that are active during the connection set up phase.

```

protocol part connect_s (peer receiver)           // CONNECTION SET UP SENDER
timer t: 0..? ms                                // Timer: ACK monitoring
XDATrequ and XDATrequ.sequ = 1:                  // triggering event
begin
  coding_DT (XDATrequ.source-addr,XDATrequ.dest-addr,1,XDATrequ.data)
  DT → receiver                                  // sending DT_1
  start t                                        // start ACK monitoring
  wait event{                                    // awaiting ACK
    ACK ← receiver and ACK.sequ=1:              // ACK received
      reset t                                    // stop ACK timer
      conn:= ACK.conn
      respond XDATconf(conn,1)                  // connection is set up
      set DATA TRANSFER |                       // change of the phase
    timeout t: respond XABORTind                // abort connection set up
  }
end //connect_s

protocol part connect_r (peer sender)           // CONNECTION SET UP RECEIVER
DT ← sender and DT.sequ = 1:                     // Timer: ACK monitoring
begin
  determine connection reference(conn)
  respond XDATind(conn,1,DT.data,DT.eom)         // deliver XDATind_1
  coding_ACK (conn,1)                            // coding ACK_1
  ACK → sender                                    // sending ACK_1
  set DATA TRANSFER                             // change of the phase
end //connect_r

```

During data transmission the protocol parts *transfer_s* and *ack_handler* are activated at sender side, at receiver side only the protocol part *transfer_r* is activated. The protocol part *transfer_s* maps the data units received from the sender into *DT*-PDUs and sends them to the receiver. A copy of each PDU is stored in a buffer. When the buffer is full, a break is indicated to the sender through an *XBREAKind* and the condition *break* is set. No break is triggered when the *DT*-PDU is transmitted, i.e., *eom* = *true*.

```

protocol part transfer_s (peer receiver)       // DATA TRANSMISSION SENDER
XDATrequ
and not go_back_N and not break:

```

```

begin
  incr sequ                                // next sequence number
  coding_DT (conn,sequ,XDATrequ.data)
  if (DT.eom)                             // last data unit?
    {last:=sequ}                          // number of the last data unit
  copy DT in buffer
  DT → receiver                           // sending DT
  if (buffer is full and not last=sequ)
    {break:=true                          // break
     respond XBREAKind(conn)
    }
    else respond XDATconf(conn,sequ)      // deliver sending confirmation
end //transfer_s

```

The protocol part *ack_handler* monitors the arrival of the acknowledgements from the receiver. It runs in parallel to *transfer_s*. All arriving correct acknowledgements erase the respective *DT*-copy in the buffer. If the entity is in a break state the break is finished. After receiving the acknowledgement of the successful transmission of the last *DT*-PDU the connection is released with an *XDISind* to the *sender*. If no correct *ACK*-PDUs arrive within time interval *t2* the *ack_handler* calls the *go back N*-procedure. Since the *ack_handler* is always active, it also accepts the *ABO*-PDU in case of a connection abandonment and terminates the protocol. Moreover, it monitors the activity of the receiver entity. If no *ACK*-PDU arrives from *XR* within the time period *t1* the *ack_handler* assumes that the *XR* is not active any more and releases the connection at sender side.

```

protocol part ack_handler(peer receiver) // MONITORING ACKNOWLEDGEMENTS
var N: integer init(1),                  // last confirmed PDU
    i: integer                             // auxiliary variable
timer t1: 0..? ms                         // timer: activity receiver
    t2: 0..? ms                             // timer: ACK monitoring
DATA TRANSFER:                             // starting event
  begin
    start t1                               // start activity timer
    loop{
      start t2                             // start ACK monitoring
      wait event{                          // awaiting ACK
        ACK ← receiver:                   // ACK arrival
          reset t1                         // stop activity timer
          reset t2                         // stop ACK timer
          if (ACK.sequ>N)                  // ACK correct?
            {N:=ACK.sequ                  // store sequence number
             erase DT copy
             if (break)
               {respond XDATconf(conn,sequ) // terminate break
                break:=false
               }
            }
      }

```

```

        if (N=last+1)                                // all ACKs received?
        {respond XDISind(conn)                        // connection release
          sequ:=1                                     // reset sequ, last
          last:= 0
          exit ack_handler                            // leaving ack_handler
        }
      }
      start t1 |                                       // restart activity timer
      timeout t2: go_back_N:=true                     // missing ACK
                i:=N+1
      loop{                                           // starting go back N
        coding_DT(buffer[i])
        DT → receiver                                // resending DT[i]
        exit when i=sequ                             // end of retransmission
        incr i
      }
      go_back_N:=false |                               // end go back N
      ABO ← receiver: respond XABORTind(conn) // protocol abortion by receiver
                sequ:=1                               // reset sequ, last
                last:= 0
                exit ack_handler |                   // leaving ack_handler
      timeout t1: respond XABORTind(conn)            // inactive receiver
                sequ:=1                               // reset sequ, last
                last:= 0
                exit ack_handler                     // leaving ack_handler
    }
  }
end //ack_handler

```

The protocol part *transfer_r* describes the reception of the *DT*-PDUs in the receiver entity *XR*. *DT*-PDUs that arrive in correct order are confirmed by an *ACK*. The user data are delivered to the receiver. PDUs out of order are discarded. If the sending order cannot be re-established within the time interval *t transfer_r* aborts the transmission by means of an *ABO*-PDU. The timer *t* also acts as an activity timer to track whether the sender entity *XS* is still active. After receiving the last *DT*-PDU the connection is released at receiver side.

```

protocol part transfer_r (peer sender)           // DATA TRANSFER RECEIVER
var N: integer init(2)                           // awaited sequence number
timer t: 0..? ms                                  // timer: order monitoring
DATA TRANSFER:                                   // triggering event
  begin
    start t                                         // start order monitoring
    loop{
      wait event {                                  // awaiting DT
        DT ← sender: if (DT.sequ = N)              // correct sequence number?
          {reset t                                   // correct order / stop timer t
            respond XDATind(conn,N,DT.data,DT.com) // delivering data to receiver
            ACK(conn,incr N) → sender
          }
        }
    }
  end

```

```

                                // sending ACK
                                // last data unit ?
if (DT.eom)
    {respond XDISind(conn)
                                // connection release
                                // change of phase
    set CONNECT
    exit transfer_r
    }

                                // restart order monitoring
start t
}
else discard DT |
                                // DT out of order
                                // order not re-established
timeout t: respond XABORTind(conn)
ABO(conn) → sender
                                // abort transmission
set CONNECT
                                // change of phase
exit transfer_r
                                // leaving transfer_r
}
}
end //transfer_r

```

Further reading

As for services, introductions to the protocol concept can be also found in all introductory books on computer networks, such as [Kuro 08], [Stal 08], [Pete 07], and [Tane 10].

Exercises

- (1) What are entities? What is their role within the service provider?
- (2) What is a protocol? What is its relation with services? What is the difference between symmetric and asymmetric protocols?
- (3) Why has the structure of protocol data units to be known to both entities?
- (4) Explain the principle of transparency. Why is it needed? Which well-known network method is based on this principle?
- (5) What are typical parameters of the protocol control information?
- (6) How are connections handled in a protocol?
- (7) What is the purpose of the protocol specification?
- (8) What is the task of timers in protocols? What are the basic timer functions?
- (9) Explain the need for the reset function. What could happen if no reset function is available?
- (10) Concurrency is a characteristic feature of protocol entities. Consequently, interleaving is a possible model to describe concurrent behavior. The XDT sender entity executes the sending of data PDUs and the reception of acknowledgements concurrently. Give some possible interleaving sequences.
- (11) Replace the implicit connection acceptance in the XDT protocol according to exercise (11) in Chapter 1 by an explicit one, i.e., the receiver no longer accepts every connection; it can accept or reject a connection.
 - a) Describe this protocol extension by means of time sequence diagrams. Introduce appropriate PDU names.
 - b) Describe the protocol changes for both entities in the model language.

- (12) When the transmission order in XDT cannot be re-established the receiver entity *XR* aborts the transmission with an *ABO*-PDU. The reception of *ABO* at the sender entity *XS* is not monitored by *XR*. How does the protocol behave when *ABO* gets lost?
- (13) In exercise (12) of Chapter 1 we extended the XDT service at the receiver side by a mechanism by which the service user may interrupt the reception of data for a certain time period. The time was given as a parameter to the service provider which controls the break duration and continues to deliver data after this time has elapsed. Extend the XDT receiver entity specification appropriately to handle this extension. Describe the protocol changes in time sequence diagrams and in the model language.
- (14) Replace the implicit connection release in the XDT protocol by an explicit one that is triggered by the receiver through an *XDISrequ* when it has received the last PDU. The sender releases the connection with an *XDISind*, after which also the receiver releases the connection with an *XDISind*.
 - a) Describe this extension by means of time sequence diagrams. Introduce appropriate PDU names.
 - b) Describe the changes for both entities in the model language.



<http://www.springer.com/978-3-642-29144-9>

Protocol Engineering

König, H.

2012, XVI, 528 p., Hardcover

ISBN: 978-3-642-29144-9