A collision problem in OSI standard formal specifications

T. Massart

University of Brussels, Laboratoire d'Informatique Théorique, Bd. du Triomphe, C.P. 212, 1050 Brussels, Belgium

Abstract

A collision problem is presented which can occur between two adjacent protocol entities, a user and its local provider. We consider synchronous and asynchronous communication mechanisms at the Service Access Point between the entities; this is normally an implementation choice. It is shown that even if the problem is limited by using a synchronous communication mechanism, instead of an asynchronous one, it still occurs. We suggest that whenever this case is found, the service provided by the protocol entity must be interpreted differently by its user, ignoring some primitives. When an asynchronous communication mechanism is used, care must be taken to verify that those primitives to be ignored cannot be misinterpreted as new primitives; finally, we point out that the protocol specification could be redesigned to handle these collision cases properly.

Keywords: Protocol specification; Service access point; Synchronous communication; Asynchronous communication.

1. Introduction

In computer communication, to formally specify a service or a protocol, several specification techniques can be used. Such a specification may consist of the set of valid sequences of service primitives or protocol data units, or it may be defined by using models (state machine, Petri net, processes, ...) reflecting the behavior of the service provider or of the protocol entities.

Generally, a protocol specification can be seen as a collection of processes communicating on the one hand with a lower-layer service, which can also be seen as a set of communicating processes, and on the other hand with users for which it provides a service. Those interactions aim at synchronizing and exchanging messages. This communication mechanism must be defined for:

- "internal" communications between processes of a common (sub)specification, or
- "external" communications at Service Access Points between a user and a provider entity of a lower layer.

Synchronous or asynchronous communications can be used for that purpose.

If the "internal" communications are part of the protocol specification, nothing is clearly specified on the communication mechanism that must be used at the Service Access Point between a user entity and a provider entity: this mechanism is generally seen as an implementation choice.

This paper examines both synchronous and asynchronous mechanisms for the communication of two entities at a Service Access Point and exhibits, from an example, a collision problem that can appear with standard protocols; we also suggest a possible solution.

2. Synchronous and asynchronous communications

In synchronous communications, two or several processes of the same layer or two contiguous layers agree to communicate: this communication may be seen as an event or as an atomic action without duration, requiring simultaneous participation of the processes involved. From this communication,

- it results in the knowledge of the moment (point in time) at which the implied processes agreed to communicate (and did it),
- it may result in the exchange of information between the concerned processes.
The moment of the communication is an information in the sense that each entity concerned by the communication knows that at this moment the other processes were at a certain point of their processing, that they have exchanged information, and know the content of this information.

Note that the agreement to communicate can take a while to happen, or even never happen, but when it is reached, the communication is instantaneous: the exchange of information during the communication is defined as not taking any time.

For instance, LOTOS [2] uses synchronous communications between processes. We use this language to formalize our examples.

Remarks:
• Petri net models can also be seen as models using synchronous communications. The basic communication mechanism corresponds to the filling of a place; this is possible if the conditions to fire a transition are fulfilled; those conditions depend on conditions on each participating entity).
• OSI services standards use time-sequence diagrams and queue models. From their design, it appears that those models suppose also a synchronous communication between the provider and each user.

Asynchronous communications occur generally between two processes. During an asynchronous communication, there is no direct agreement between these processes. The communication may be seen as the agreement between one of the processes, generally called sender, and a third party, followed by a second agreement between this latter and the other communicating process, generally called receiver. Each agreement may be seen as a synchronous communication, but no synchronous communication is possible between the two communicating processes.

The third and intermediary party in this communication, may be seen as a particular process or object, playing the role of relay of the “message” in the communication. This entity, generally called queue, may or may not be explicitly expressed in the specification, depending on the model used. It proceeds to the handling of messages in a FIFO order, in one or both ways between the two processes. Note also that generally, the queue is always ready to communicate (accept messages and deliver them to its destination after a variable delay), but a capacity may be associated to it such that if this capacity is reached, no new message can be accepted.

From this communication, it may result in:
• that at a point in time, the receiver knows that the sender has already reached a point in its processing (the moment when the message has been sent),
• the knowledge, by the receiver, of the transmitted information.

Among the specification languages using asynchronous communication, Estelle [3] and SDL [1] are the best known.

It can be noted that Estelle and SDL are based on extended Finite State Machine (FSM) models; however, asynchronism and FSM models are not necessarily associated:
– synchronous communications can be used by some kind of FSM models; this is done for instance in the transition tables used in various OSI protocol standards.
– asynchronous communication can be considered within LOTOS for instance, by using ad-

Thierry Massart studied computer sciences at the University of Brussels. In 1985, he became an assistant professor at this university where he specialized in the formal description of distributed systems. He received a Ph.D. degree in computer science in 1991. His thesis was about the definition of an algebraic calculus and the design, by correctness-preserving transformations, of distributed systems.
ditional objects or processes to model the pairs of FIFO queues at the Service Access Points, or where needed.

In the following, in order to have an homogeneous formalism for both communication mechanisms, LOTOS specifications with queue processes will be used to specify processes using asynchronous communication.

Remarks:

- In the following, we will talk about synchronous and asynchronous models, referring to models using the corresponding communication mechanism.
- Synchronous models can simulate asynchronous communications by the introduction of a queue process between the communicating processes. On the other hand, synchronous communication cannot be simulated by asynchronous models.

3. The collision problem

We have modeled the communication mechanism at a Service Access Point, called g, between an entity of layer \( N + 1 \), called user \( U \), and an entity of layer \( N \), called provide \( P \). We point out that the selection of this communication mechanism, which is generally seen as an implementation choice may be very sensitive in the design of a protocol and the service provided. We illustrate this by a simple example.

The collision happens when two contiguous entities try to make two activities (part of services for instance) which cannot be done simultaneously.

Suppose that, if an event \( \text{ue} \) occurs at the user side, the user must request the provider to realize an action \( \text{pa} \), and if an event \( \text{pe} \) occurs at the provider side, the provider has to request (using an indication) the action \( \text{ua} \) to be realized. Suppose also that the request and the indication (and by consequence both actions \( \text{pa} \) and \( \text{ua} \)) are mutually exclusive by service definition.

This could typically be handled by adding critical sections in the concerned parts of the implementation. Unfortunately, one can see that this type of problems could lead to the introduction of a lot of critical sections in the various layers of the OSI stack, which would require a centralized control between those layers. This could lead to non performant implementations. Another solution must be found.

Let us model the example with both communication mechanisms:

**Synchronous model**

Process \( U[\text{ue}, \text{g}, \text{ua}]: \) exit :=

\[ \text{ue}; \text{g!request}; \text{exit} \]

\[ \square \]

\[ \text{g!indication}; \text{ua}; \text{exit} \]

endproc

Process \( P[\text{pe}, \text{g}, \text{pa}]: \) exit :=

\[ \text{pe}; \text{g!indication}; \text{exit} \]

\[ \square \]

\[ \text{g!request}; \text{pa}; \text{exit} \]

endproc

**Remark.** Generally \( U \) and \( P \) are much more complex; one can imagine that the termination of \( U \) or \( P \) occur only after a long execution (or sequence of transitions).

The parallel composition of \( U \) and \( P \), synchronized on \( g \) leads to a deadlock problem:

Specification \( \text{ex} = \text{sync}[\text{ue}, \text{g}, \text{ua}, \text{pe}, \text{pa}] \) exit

type \( \text{event} \) = \( \text{type} \)

\( \text{sorts} \) \( \text{event} \)

\( \text{opns} \) \( \text{request, indication} \) \( : \rightarrow \) \( \text{event} \)

endtype

behaviour

\( U[\text{ue}, \text{g}, \text{ua}] \ || [g] \ || P[\text{pe}, \text{g}, \text{pa}] \)

where

Process \( U /* * \) see definition above */

Process \( P /* * \) see definition above */

endspec

The global behaviour is equivalent to

\[ (\text{ue}; \text{g!request}; \text{pa}; \text{exit} \]

\[ \square \]

\[ \text{pe}; \text{g!indication}; \text{ua}; \text{exit} \]

\[ \square \]

\[ \text{ue}; \text{pe}; \text{stop} \Box \text{pe}; \text{ue}; \text{stop} \]

This is due to the fact that, by our hypothesis, \( \text{ue} \) and \( \text{pe} \) must be exclusive events.
Asynchronous model

Given the specification of $Q$, a process which simulates a pair of reliable FIFO queues between $U$ and $P$ with gates $\text{gui}$, $\text{guo}$, $\text{gpi}$, $\text{gpo}$, the parallel composition of $U$ and $P$ linked with $Q$ can lead to a situation where "messages" stay in the queue without ever being handled.

Specification ex – async[ue, ua, pe, pa, gui, guo, gpi, gpo]: exit
type event – type
sorts event
opns request, indication: → event
endtype
behaviour
$U[\text{ue}, \text{gui}, \text{guo}, \text{ua}] \parallel [\text{gui}, \text{guo}]$
$Q[\text{gui}, \text{guo}, \text{gpi}, \text{gpo}]
[\text{gpi}, \text{gpo}] P[\text{pe}, \text{gpi}, \text{gpo}, \text{pa}])$

where

Process $U[\text{ue}, \text{gui}, \text{guo}, \text{ua}]: \text{exit} :=$
$\text{ue}: \text{guo}!\text{request}, \text{exit}$
$\square$
$\text{gui}!\text{indication}; \text{ua}; \text{exit}$
endproc

Process $P[\text{pe}, \text{gpi}, \text{gpo}, \text{pa}]: \text{exit} :=$
$\text{pe}; \text{gpo}!\text{indication}; \text{gpi}!\text{request}; \text{exit}$
$\square$
$\text{gpi}!\text{request}; \text{pa}; \text{exit}$
endproc

Process $Q[\text{gui}, \text{guo}, \text{gpi}, \text{gpo}]: \text{exit} :=$
$(\text{guo}!\text{request}; \text{gpi}!\text{request}; \text{exit}$
$\parallel$
$\text{gpo}!\text{indication}; \text{gui}!\text{indication}; \text{exit})$
$\square$
endproc

endspec

The global behaviour is equivalent to

$(\text{ue}; \text{guo}!\text{request}; \text{gpi}!\text{request}; \text{pa}; \text{exit}$
$\square$

\footnote{\text{gui}, \text{guo}, \text{gpi}, \text{gpo} stand for user and provider in and out gates.}

pe; $\text{gpo}!\text{indication}; \text{gui}!\text{indication}; \text{ua}; \text{exit}$
$\square$
$(\text{ue}; \text{guo}!\text{request}; \text{exit} \parallel \text{pe};$
$\text{gpo}!\text{indication}; \text{exit}))$

Here also $\text{pe}$ and $\text{ue}$ should be mutually exclusive events.

Remarks:
- We have seen that with synchronous models this leads directly to a deadlock. With an asynchronous one, various problems may occur:
  (a) deadlock due to the fact that the first communication in the queues may not be taken off during the remaining part of the protocol execution.
  (b) misunderstanding of the communication by one or both entities: they can think that it is a request or an indication which has been sent at a later stage. This can lead to a desynchronization between the user and the provider.
- This collision problem can be more serious with asynchronous models. Indeed, the collision problem comes from the disallowing of the simultaneous occurrence of (or in this model, any interleaving in) events in the sequence $(\text{ue}; \text{guo}!\text{request}; \text{pa})$ with events in the sequence $(\text{pe}; \text{gpo}!\text{indication}; \text{ua})$.

Suppose now that we have in the provider a sequence of events $\text{pe}_i (0 \leq i \leq m)$ leading to the indications asking for the execution of the corresponding $\text{ua}_i$, and that this sequence must be exclusive for an event $\text{ue}$ leading to a user request, asking for an action $\text{pa}$. With an asynchronous model, all the indications can have been sent by the provider into the queue before it has seen that the collision has occurred. Then the model allows the simultaneous occurrence (or in LOTOS, the interleaving) of events in the sequence $(\text{ue}; \text{guo}!\text{request})$ with (some or all) events of the sequence $(\text{pe}_1; \text{gpo}!\text{indication}_1; \text{pe}_2; \text{gpo}!\text{indication}_2; \text{pe}_3; \text{gpo}!\text{indication}_3; \ldots)$

4. Practical example of this problem

If no care is taken, this problem can, for instance, occur between a connection-oriented datalink protocol such as the LLC2 procedures [4] and a network protocol using the service provided.
In a disconnection state, the service provided at the network layer is the following (see Fig. 1): the user may either receive a CONNECT–INDICATION to which it must reply by a CONNECT–RESPONSE or a DISCONNECT–REQUEST or request a connection, by a CONNECT–REQUEST; the answer to this request is either a CONNECT–CONFIRM or a DISCONNECT–INDICATION.

At the protocol level, when an LLC2 entity is disconnected, it may either receive from the MAC, a remote SABME or a local CONNECT–REQUEST. The state tables specify that:
- the reception of the CONNECT–REQUEST leads to the transmission of an SABME (Fig. 2a).
- the reception of an SABME leads, normally, to the indication to its user of the request for the connection (Fig. 2b).

But if the user wants to transmit the CONNECT–REQUEST and simultaneously the LLC2 entity wants to send a CONNECT–INDICATION after the reception of an SABME (Fig. 2c), what happens? Either, in the synchronous case, the user cannot request the connection and must receive the indication, which may be constraining or, in the asynchronous case, the user sends its request and then receives an indication which is not a valid service.

5. Proposed solution

A possible solution is to include the collision case at the specification level, so that whatever communication mechanism is implemented at the Service Access Point, the protocol works properly. This solution arises from the observation that the request (indication) is only a mean to ask for the firing of pa, (ua). We can slightly modify the hypothesis into the mutual exclusion of ua and pa.

A convention must be taken between the user and the provider in order to respect this hypothesis.

A possibility could be that the user U is supposed to be the "master" and the provider P the "slave" (the reverse could also be supposed). In case of collision, on the one hand if the user receives the indication, it ignores it (as if it had never been received), and on the other hand, the provider, which just sent the indication must still be able to receive the request, and handles the collision case; this can lead to the firing of pa or to another sequence of actions (see below pa1; pa2). Let us see the corresponding specifications with both communication mechanisms.

Synchronous model

Process U[ue, g, ua]: exit :=
  ue; g!request; exit
  □
  g!indication; ua; exit
endproc

Process P[pe, g, pa]: exit :=
  g!request; pa; exit
  □
pe; (g!indication; exit □ g!request; pa1; pa2;
exit)
endproc

Asynchronous model

Process U{ue, gui, guo, ua}: exit :=
  ue; guo!request; (exit □ gui!indication; exit)
□
gui!indication; ua; exit
endproc

Process P{pe, gpi, gpo, pa, pa1, pa2}: exit :=
gpi!request; pa; exit
□
pe; gpo!indication; (exit □ gpi!request; pa1;
  pa2; exit)
endproc

Note also that with the synchronous model, the request and indication are always mutually exclusive and therefore the user must never move away an indication after it has sent a request; this mutual exclusion cannot so easily be realized in the case of an asynchronous model without another control between the entities such as centralized control or another appropriate method.

Moreover care must be taken in specifications or the implementations designed with an asynchronous communication such as all cases of collision are properly detected by both user and provider. If there is a situation where a message can be interpreted as the result of a case of collision or as the normal processing of a later stage, another control must be used.

6. Conclusions

We have presented a collision problem which can occur when implementing protocol entities between a user and its local provider. We have shown that even if the problem is limited by using a synchronous communication mechanism between the entities instead of an asynchronous one, the problem still occurs. We have stated that, wherever this case is found, to allow different implementations of the Service Access Points, case can be taken at the protocol specification and service specification level as follows:
1. the service must be interpreted differently by the user, ignoring some primitives.
2. the protocol must be designed to handle this case, and in particular care must be taken to verify that those primitives to be ignored cannot be misinterpreted as new primitives.

Acknowledgments

I am grateful to R. Devillers, P. Van Binst and also to the reviewers E. Andersen and N. Nørup Pedersen for their careful reading and helpful commenting on an earlier version of this paper.

References