Leslie Lamport's Logical Clocks: a tutorial

Rob R. Hoogerwoord

29 january 2002

Contents

Chapter 0

Introduction

In 1978 the American computing scientist Leslie Lamport published a paper [2] in which he introduced so-called *logical clocks* to synchronize processes in a distributed system. These logical clocks can be used to record (information about) the *causal order* in which events in the distributed system take place: some events always "occur before" other ones, independent of the (relative) speeds of the processes. In his paper Lamport used this in a *fair* algorithm for distributed mutual exclusion; here, "fair" means that the order in which requests (to obtain the exclusive right) are granted complies with the causal order of these requests.

This note is mainly written for tutorial reasons, not in the least to develop my own understanding of the subject. A minor reason is my dissatisfaction with the cumbersome notation and nomenclature in the existing presentations, not only in Lamport's original paper but also in recent publications like one by Raynal and Singhal [3]. In addition, this is an exercise in separation of concerns.

This note consists of two parts that can be read independently, as these parts are rather self-contained. In Section 1 we introduce *causal precedence* and Lamport's logical clocks to record information about causal precedence. In Section 2 we present an algorithm for distributed mutual exclusion. This is *not* Lamport's algorithm: to fully understand his algorithm I decided to design it myself, but it so happened that I came up with a simpler, and somewhat more efficient, algorithm. This turned out to be the algorithm designed by Ricart and Agrawala [4] .

The connection between the two stories is rather thin: some variables used in the algorithm for mutual exclusion may be viewed as logical clocks, but to understand (the correctness of) the algorithm one does not need to know this.

Chapter 1

Causal precedence and logical clocks

1.0 Processes, states, and events

An *event* is a change of state in a (single) process. This process can be part of a distributed collection of processes. The adjective "distributed" means that the state of the system, as a whole, is distributed over the individual processes: each process can only change its own state. Moreover, the state of the system consists *exclusively* of the states of its processes.

The processes communicate by exchanging *messages*. We assume that sending a message by one process and receiving it by another is not an indivisible (atomic) action, that is, message transmission is not instantaneous; in this case all messages sent but not yet received are also part of the state of the system. Mathematically, we could model this by introducing additional processes, so-called *channels*, whose states would represent the messages sent but not yet received. This is, however, not necessary, because we can also, for every process proper, view the states of its outgoing channels as part of the state of that process. Phrased differently, if we equip every process with two auxiliary variables, one representing the set of messages sent by that process and one representing the set of messages received by that process, then the differences between the "sent sets" and the "received sets" represent the states of the (directed) channels connecting the processes.

In this way, the above view of the state of a distributed system is equally well applicable to systems with non-instantaneous – also called *asynchronous* – communication.

Because both sending a message and receiving a message affect the state

of the sending or receiving process, both sending a message and receiving a message are events, in the above meaning of the word. Notice that the representation with sent sets and received sets complies with the view that each process only changes its own state: a sending process changes its state by adding the message to its sent set, whereas a receiving process changes its state by adding the message to its received set. Hence, the channel state is distributed over sender and receiver in a way that is consistent with our view of a distributed system.

1.1 Naming conventions, and some more

To distinguish the processes we assume that they have been properly named. In this note we will use variables p, q, r to range over the process names.

In the course of a process the *same* state transition may occur *several* times, but such multiple occurrences of the same state transition are considered *different* events. So, for example, a state transition like $x := x+1$ may occur many times within a process but each individual occurrence of $x := x+1$ is a separate event. To distinguish all events in the system we assume that the events have been properly named, thus making each event unique; in what follows we will use variables e, f, g to denote (names of) events.

An event is a state transition in a *single* process, so each event uniquely identifies the process in which it occurs. This is formally rendered by means of a function pr, from events to processes, with the interpretation:

$$pr{\cdot}e \;=\; \text{“the process in which event } e \text{ occurs”} \quad .$$

Possible events are “sending a message” and “receiving a message”, and for our purposes these are the only events of interest. We shall ignore so-called *internal* events, representing purely local state changes of processes; if so desired, internal events can be treated as send events.

Every message is sent by exactly one process – its sender – and what follows is based upon the assumption that every message is also received by exactly one process – its receiver –. We only make this assumption here to keep the notation as simple as possible, but the assumption is not essential: it is perfectly admissible to have the same message being received by many (or all) processes in the system – so-called *broadcast messages* –.

Every send event and every receive event involves a unique *message*; therefore, these events can be represented by their messages: we also assume that the messages have been properly named. We will use variables l, m, n to denote (names of) messages. So, every message represents two events, namely

the event of sending it and the event of receiving it. To formalize this we introduce, for every message m:

$$\begin{array}{lcl} snd{\cdot}m & = & \text{“the event of sending message } m\text{”} \\ sdr{\cdot}m & = & \text{“the process that sends message } m\text{”} \\ rcv{\cdot}m & = & \text{“the event of receiving message } m\text{”} \\ rvr{\cdot}m & = & \text{“the process that receives message } m\text{”} \end{array}$$

This nomenclature is redundant, but this is an advantage: whenever we wish to take the nature of events into account we use messages to identify events and whenever the nature of events is irrelevant we identify them by their own names. Because of this redundancy we can also formulate a few relations that reflect the above interpretations. Because we consider no other events than sending and receiving, we have, for every event e:

$$(\exists m :: e = snd{\cdot}m \ \vee \ e = rcv{\cdot}m) \quad .$$

Furthermore, functions sdr and rvr are related to snd, rcv, and pr as follows – for all m –:

$$\begin{array}{lcl} sdr{\cdot}m & = & pr{\cdot}(snd{\cdot}m) \\ rvr{\cdot}m & = & pr{\cdot}(rcv{\cdot}m) \end{array}$$

A complete formalisation also requires formulae expressing that all send events differ from all receive events and that different messages represent different events, but for the time being I feel no urge to write down these formulae.

1.2 Sequential precedence

We assume that every single process in the distributed system is *sequential*; a sequential process is one in which any two different events can be ordered: either the one event occurs *before* the other event or it occurs *after* it. For any two events e, f we use $e \rhd f$ to denote *sequential precedence* of e and f.

sequential precedence: for all events e, f:

$$e \rhd f \quad \equiv \quad pr{\cdot}e = pr{\cdot}f \ \wedge \ \text{“}e \text{ occurs before } f\text{”} \quad .$$

□

We shall not further formalize the notion “occurs before”, because we do not need it; in what follows $\rhd$ can be considered as primitive. For every process the relation $\rhd$ is a total order on the events in that process:

$$(\forall e, f : pr{\cdot}e{=}pr{\cdot}f :\ e{=}f \ \vee\ e \rhd f \ \vee\ f \rhd e\ )\quad .$$

The *sequential history* of event f is the set H_f of all events (sequentially) preceding f, that is, for all events e, f we have:

$$e{\in}H_f \ \equiv\ e \rhd f\quad .$$

The history of an event can be viewed as (a representation of) the *state* of the process immediately *before* occurrence of that event. This enables us to formulate *preconditions* for events: a precondition for an event thus is a proposition about its history. Also, we can now formulate that processes, which may be infinite – non-terminating –, have, at any moment in time, been in existence for a finite amount of time only.

postulate: For all f set H_f is finite.
□

aside: A sequential process could also be defined as an infinite sequence $x_i(i : 0{\leq}i)$ of events, such that x_j is the event with exactly j sequentially preceding events; that is, the events would be identified here by their ordinal numbers. Then we would have, for all natural i, j:

$$x_i \rhd x_j \ \equiv\ i < j\quad .$$

In these terms all properties attributed to $\rhd$ could be proved as theorems: the (representation by an) infinite sequence can be viewed as an underlying model. For example, the history of event number j then consists of all events with ordinal numbers less than j, which is definitely finite.

In a system of many sequential processes events cannot so easily be numbered; we could, of course, identify every event by a pair containing the name of its process and its local ordinal number, but this is technically awkward and needlessly overspecific.
□

1.3 Causal precedence

In a concurrent computation the best we can say is that *some* events are *causally* ordered: one event causally precedes another means that, independent of the relative speeds of the processes, the one event certainly occurs before

the other. Not every pair of events can thus be ordered: causal precedence is a partial order.

In its simplest form, event e causally precedes event f either if e and f occur in the same (sequential) process and e occurs before f, or if e is the sending of a message and f is that message's reception: messages are always received after they have been sent. In addition, causal precedence is *transitive*: if e precedes f and f precedes g then it is reasonable to state that e precedes g as well.

Notice that the notion of *time* plays no role here: we are only concerned with the relative order in which events take place, not in the exact moments at which they occur. Of course, we may conclude that one event precedes the other if the one occurs at an earlier time than the other, but in this way time is introduced into the discussion in an artificial way. This observation is particularly relevant for (truly) distributed systems, where it is physically meaningless to compare times in different processes.

Causal precedence can be used to implement distributed arbitration, such as in a mutual exclusion protocol, in a fair way: we now can specify that arbitration requests are granted in an order that respects the causal order of these requests. This may, for instance, be needed to maintain the consistency of a distributed data base.

Causal precedence is the *transitive closure* of a *basic precendence* relation. Event e basically precedes event f, notation $e \mapsto f$, either if e sequentially precedes f or if e and f are the sending and receiving of the same message (or both).

basic precedence: for all events e, f:

$$e \mapsto f \quad \equiv \quad e \triangleright f \ \ \vee \ \ (\exists m :: e = snd{\cdot}m \wedge rcv{\cdot}m = f) \quad .$$

□

This definition can be rewritten into the following equivalent, but more manageable, form; this is the form we will be using.

basic precedence: $\mapsto$ is the strongest relation satisfying:

$$(\forall e, f :: e \triangleright f \Rightarrow e \mapsto f) \ \ \wedge \ \ (\forall m :: snd{\cdot}m \mapsto rcv{\cdot}m) \quad .$$

□

causal precendence: $\rightarrow$ is the transitive closure of $\mapsto$.

□

It may seem somewhat overdone to introduce the basic precedence relation as a separate concept, but, as we will see, it so happens that the discussion can be carried out largely in terms of basic precedence, which is simpler. In particular, the following little lemma is relevant.

lemma: For any *transitive* relation R we have:

$$(\forall e, f :: e \mapsto f \Rightarrow e\,R\,f) \;\Rightarrow\; (\forall e, f :: e \rightarrow f \Rightarrow e\,R\,f) \quad .$$

In words: to prove that causal precedence implies some transitive relation R, it suffices to prove that basic precedence implies R, and the latter is weaker.

□

Causal precedence is a partial order: not every two events need be related. Events that are not causally related are called *concurrent*, notation $e \,||\, f$.

concurrency: for all events e, f:

$$e \,||\, f \quad \equiv \quad \neg(e \rightarrow f) \;\wedge\; \neg(f \rightarrow e) \quad .$$

□

Because causal precedence is not a total order, we do not have:

$$(\forall e, f :: e{=}f \;\vee\; e \rightarrow f \;\vee\; f \rightarrow e)$$

but we do have the weaker relation:

(0) $$(\forall e, f :: e \,||\, f \;\vee\; e \rightarrow f \;\vee\; f \rightarrow e) \quad .$$

As we did with sequential precedence, we can now define the *causal history* of an event f as the set G_f of all events causally preceding it:

$$e{\in}G_f \quad \equiv \quad e \rightarrow f \quad .$$

Just as the sequential history of an event represents the state of the process just before occurrence of the event, we can now view the causal history of an event as (the relevant part of) the state of the whole system just before occurrence of the event. From the axiom that the sequential histories of all events are finite it follows, via the above definitions, that causal histories are finite as well. Moreover, it is now possible to prove that every event $snd{\cdot}m$ is an element of the (causal) history of the event $rcv{\cdot}m$, which is a formal rendering of the property that no message is received before it has been sent. (This also explains the use of the adjective "causal".)

property:

$$(\forall m :: snd{\cdot}m \in G_{rcv{\cdot}m}) \quad .$$

□

1.4 Local and global states

In a (partially or totally) ordered universe we call a subset *closed* if, for every element of that subset, all values smaller than that element are contained in the subset as well.

definition: A subset V (of some universe) is *closed with respect to* relation $<$ (on that universe) means:

$$(\forall x, y :: x < y \wedge y \in V \Rightarrow x \in V) \quad .$$

□

Closedness of sets is connected to transitivity of the relation $<$, in the following way. For every y we can define a set V_y by:

$$x \in V_y \quad \equiv \quad x < y \quad , \text{ for all } x \quad .$$

Then, transitivity of $<$ is equivalent to "V_y is closed, for all y".

Because the sequential precedence relation $\triangleright$ is transitive, the sequential history H_f of an event f is *closed with respect to* $\triangleright$. In Section 1.2 we have called such a set a *state*; in view of the presence of other processes, and of the (yet to be introduced) notion of *global state*, we call such a set a *local state*.

definition: A *local state* is a finite set of events, all part of one and the same process, that is closed with respect to $\triangleright$.

□

In a very similar way, because the causal precedence relation $\rightarrow$ is transitive too, the causal history G_f of every event f is closed with respect to $\rightarrow$. The causal history of an event is (the relevant part of) the *global state* of the system just before that event takes place, and any finite set closed with respect to $\rightarrow$ represents some global system state.

definition: A *global state* is a finite set of events that is closed with respect to $\rightarrow$.

□

Not every finite set of events is a global state: the requirement of closedness is essential. In some of the literature, like [1], the phrase "consistent global state" is used; here the (somewhat redundant) adjective "consistent" just reminds of this closedness requirement.

* * *

We now introduce an (auxiliary) variable Σ_p to represent the local state of process p. Its value is the set of all events that have occurred in process p "thus far". Notice that the order of these events is represented by $\triangleright$, which is why we only need a *set* (instead of a *sequence*). The initial value of Σ_p is ϕ, and the effect on Σ_p of occurrence of event f in process p is:

$$\Sigma_p := \Sigma_p \cup \{ f \} \quad .$$

Set Σ_p contains the whole history of process p and from a mathematical point of view this is convenient; in any practical implementation, however, the state of the process will only contain that part of its history that is relevant for its future. In view of such implementations Σ_p is best considered an auxiliary variable.

If event f "is about to occur" in process p then the state Σ_p contains exactly those events that sequentially precede f. This means that $H_f = \Sigma_p$ is a valid precondition of $\Sigma_p := \Sigma_p \cup \{ f \}$; if so desired, this precondition can be taken as H_f's definition.

The (global) state of the system can now be defined as the union of the (local) states of the individual processes. Calling the global state Γ we obtain:

$$\Gamma \;=\; (\cup p :: \Sigma_p) \quad .$$

In what follows we shall treat Γ (which is redundant), not as an additional variable but as just an abbreviation of the expression $(\cup p :: \Sigma_p)$; thus, we do not have to include manipulations of Γ in the code of our programs.

Just as $H_f = \Sigma_p$ is a valid local precondition of $\Sigma_p := \Sigma_p \cup \{ f \}$, this state change also has as valid *global* precondition:

$$G_f \subseteq \Gamma \quad .$$

This cannot be strengthened into an equality, however, because at the moment event f is about to occur in process p, events may occur concurrently in the other processes too, as a result of which other Σ's and, hence, Γ increase. (Formally, this means that $G_f \subseteq \Gamma$ is globally correct whereas $G_f = \Gamma$ is not.)

1.5 Time stamps

The problem to be solved now is to develop a protocol by means of which information about the causal precedence of events in the system can be collected. More precisely, the problem is to define and implement an integer function T on the set of events, with the following property.

clock condition: for all events e, f :

$$e \rightarrow f \quad \Rightarrow \quad T{\cdot}e < T{\cdot}f \quad .$$

□

Traditionally, the value $T{\cdot}e$ is called the "time stamp" of event e, because one (but only one) of the possible definitions of T is such that $T{\cdot}e =$ "the moment at which event e occurs" ; we shall also use this jargon, but keep in mind that (formally) T has no connection with (physical) time whatsoever.

Because of the implication in the clock condition, function T does not completely represent causal precedence: concurrent events may have different time stamps; yet, for any two non-concurrent events their causal order is represented by their time stamps:

$$\begin{array}{ll} & T{\cdot}e < T{\cdot}f \\ \Rightarrow & \quad \{ < \text{ is antisymmetric } \} \\ & \neg(T{\cdot}f < T{\cdot}e) \\ \Rightarrow & \quad \{ \text{ clock condition } \} \\ & \neg(f \rightarrow e) \\ \Rightarrow & \quad \{ \ (0) \ \} \\ & e \,||\, f \ \vee \ e \rightarrow f \quad . \end{array}$$

So, if $T{\cdot}e < T{\cdot}f$ then either e and f are concurrent, that is, causally unrelated, or $e \rightarrow f$. Similarly, we can derive that:

$$T{\cdot}e = T{\cdot}f \ \Rightarrow \ e \,||\, f \quad .$$

Because T is an integer function and because the integers are totally ordered, function T induces an (almost) total order on the set of events, whereas causal precedence itself is a (truly) partial order; hence, generally no integer function T will also satisfy the so-called *strong clock condition*.

strong clock condition: for all events e, f :

$$e \rightarrow f \quad \equiv \quad T{\cdot}e < T{\cdot}f \quad .$$

□

To satisfy the strong clock condition we need functions to domains that are only partially ordered and not, like the integers, totally ordered. An example of such a domain is the space of N-dimensional integer vectors, where N is the number of processes in the system, but we shall not elaborate upon this here: for some applications the not-so-strong clock condition is perfectly adequate.

The solution to be derived in the next subsection does not depend on specific properties of the integers; therefore, this solution is generally applicable to various kinds of time stamps, such as the ones based upon so-called *vector clocks*, which have been invented by several researchers (see [3] for an overview).

1.6 An implementation

The problem to be solved now is to assign values to T for all events in the system, in such a way that the clock condition is satisfied for all pairs of events in Γ. Formally, this means that T is a global variable now, and that the clock condition will be a system invariant. The solution will be such that $T{\cdot}g$ will be initialised by process $pr{\cdot}g$, just *before* g takes place, for each event g: thus, global variable T is distributed over the processes.

The ordering of the integers is transitive; hence, according to the lemma in Section 1.3, the clock condition is implied by the following condition for $\mapsto$. Thus we obtain a weaker invariant, which can be established more easily.

basic clock invariant (i) :

$$(\forall e,f : e,f{\in}\Gamma \wedge e \mapsto f : \; T{\cdot}e < T{\cdot}f \,) \quad .$$

□

Because Γ is closed with respect to $\rightarrow$ and, hence, also with respect to $\mapsto$, this can be weakened – at least, formally speaking – even further: $e{\in}\Gamma$ is implied by $f{\in}\Gamma$ and $e \mapsto f$.

basic clock invariant (ii) :

$$(\forall e,f : f{\in}\Gamma \wedge e \mapsto f : \; T{\cdot}e < T{\cdot}f \,) \quad .$$

□

By means of the definition of $\mapsto$ this can be reformulated into the following equivalent form, from which $\mapsto$ has been eliminated altogether.

basic clock invariant (iii) :

$$(\forall e,f : f{\in}\Gamma \wedge e \rhd f : T{\cdot}e < T{\cdot}f\,) \;\; \wedge$$
$$(\forall m : rcv{\cdot}m{\in}\Gamma : T{\cdot}(snd{\cdot}m) < T{\cdot}(rcv{\cdot}m)\,) \;\; .$$

□

The two conjuncts of this version can be dealt with in isolation. A viable design strategy is, first, to satisfy the one conjunct by means of an as liberal solution as possible and, second, to restrict this solution to account for the second conjunct as well. That is what we shall attempt.

the first conjunct

The first conjunct from the basic clock invariant, that is:

$$(\forall e,f : f{\in}\Gamma \wedge e \rhd f : T{\cdot}e < T{\cdot}f\,) \;\; ,$$

can be further partitioned, because, by definition, $\Gamma = (\cup p :: \Sigma_p)$ and because for any events e, f we have $e \rhd f \;\Rightarrow\; pr{\cdot}e = pr{\cdot}f$.

As a consequence, the requirement can be meaningfully partitioned into a collection of (disjoint) parts, one per process, thus:

$$(\forall p :: \; (\forall e,f : f{\in}\Sigma_p \wedge e \rhd f : T{\cdot}e < T{\cdot}f\,)\,) \;\; .$$

This is meaningful because $f{\in}\Sigma_p \wedge e \rhd f$ implies $e{\in}\Sigma_p$ as well; therefore, this requirement can now now be satisfied term-wise, by turning it into a local invariant per process; thus, the local invariant for process p becomes:

Q0: $\quad (\forall e,f : f{\in}\Sigma_p \wedge e \rhd f : T{\cdot}e < T{\cdot}f\,) \;\; .$

By nesting dummies we rewrite this as follows:

Q0: $\quad (\forall f : f{\in}\Sigma_p : (\forall e : e \rhd f : T{\cdot}e < T{\cdot}f\,)\,) \;\; .$

The required invariance of Q0 under $\Sigma_p := \Sigma_p \cup \{\, g \,\}$ – "event g occurs" – gives rise to the precondition:

$(*)$: $\quad (\forall e : e \rhd g : T{\cdot}e < T{\cdot}g) \;\; .$

Because $e \triangleright g \equiv e \in H_g$ and because $H_g = \Sigma_p$ is a valid part of the precondition too, condition $(*)$ is equivalent to

$$(\forall e : e \in \Sigma_p : T{\cdot}e < T{\cdot}g) \quad ,$$

which can be established by means of a fresh integer variable c, satisfying:

$$(\forall e : e \in \Sigma_p : T{\cdot}e < c) \quad \wedge \quad c \leq T{\cdot}g \quad .$$

Variable c is a private variable of the process; in Lamport's jargon c is called the "logical clock" of that process; the main technical reason to introduce it is modularisation, so as to isolate Σ_p from $T{\cdot}g$: now Σ_p only occurs in the formula $(\forall e : e \in \Sigma_p : T{\cdot}e < c)$. We turn this formula into an additional invariant, so what remains is the obligation to assign to $T{\cdot}g$ a value that is at least c (and, of course, to establish the invariance of this new invariant).

Q1: $\quad (\forall e : e \in \Sigma_p : T{\cdot}e < c) \quad .$

Initially Q1 holds, because, initially $\Sigma_p = \phi$. (Hence, the initial value of c is irrelevant.) Moreover, Q1 is not violated by *increases* of c, and $\Sigma_p := \Sigma_p \cup \{ g \}$ maintains Q1 under the additional precondition $T{\cdot}g < c$; this can be established by means of a sufficient increase of c.

Hence, with respect to variables T, Σ_p, and c, the occurrence of event g boils down to a program fragment of the following shape:

$$\begin{array}{ll} & T{\cdot}g := \text{ "a value at least } c\text{"} \\ ; & \{ \ c \leq T{\cdot}g \ \} \\ & c := \text{ "a value exceeding } T{\cdot}g\text{"} \\ ; & \{ \ T{\cdot}g < c \ \} \\ & \Sigma_p := \Sigma_p \cup \{ g \} \qquad\qquad (\text{"event } g \text{ occurs in process } p\text{"}) \end{array}$$

The assertions in this fragment specify which expressions are admissible in the assignments to $T{\cdot}g$ and c; the degrees of freedom we have here will be exploited when we deal with the second conjunct of the basic clock condition. (For example, the simplest possible choices meeting all requirements thus far, are $T{\cdot}g := c$ and $c := T{\cdot}g + 1$ respectively.)

the second conjunct

We recall the second conjunct of the basic clock invariant:

Q2: $\quad (\forall m : rcv{\cdot}m \in \Gamma : \ T{\cdot}(snd{\cdot}m) < T{\cdot}(rcv{\cdot}m) \ ) \quad .$

Invariance of Q2 is guaranteed, provided we see to it that every receive event is assigned a sufficiently large time stamp, that is, by seeing to it that $T{\cdot}(rcv{\cdot}m)$ is chosen large enough, for all m. We do so by exploiting the strategic freedom in the previous program fragment, in the obvious way: if g is a receive event we just strengthen the postassertion of the initialisation of $T{\cdot}g$ with $t<T{\cdot}g$, where t is the time stamp of the corresponding send event. Thus we obtain as program fragment for receive events in process p:

$$
\begin{array}{ll}
 & \{\ snd{\cdot}m\in\Gamma\ \}\{\ g=rcv{\cdot}m\ \wedge\ t=T{\cdot}(snd{\cdot}m)\ \} \\
 & T{\cdot}g\ :=\ \text{“a value at least } c \text{ and exceeding } t\text{”} \\
;\ & \{\ c\le T{\cdot}g\ \wedge\ t<T{\cdot}g\ \} \\
 & c\ :=\ \text{“a value exceeding } T{\cdot}g\text{”} \\
;\ & \{\ T{\cdot}g<c\ \} \\
 & \Sigma_p\ :=\ \Sigma_p\cup\{\,g\,\} \qquad\qquad (\text{“message } m \text{ is received in process } p\text{”})
\end{array}
$$

Notice that, because of the preassertion $snd{\cdot}m\in\Gamma$ –why is it valid?–, the value $T{\cdot}(snd{\cdot}m)$ is well defined indeed: it has been initialized in the process that has sent message m. To make this value available in the receiving process, it must be transmitted together with the message proper to the receiver – whence its name “time stamp”–. Also notice that, actually, we only need the weaker preassertion $T{\cdot}(snd{\cdot}m)\le t$ instead of $T{\cdot}(snd{\cdot}m)=t$; occasionally, this may be an advantage.

1.7 Afterthoughts

In the previous two subsections, we have (deliberately) formulated the program fragments in a way that is independent of T's type: its values need not be integers. These program fragments still leave quite some freedom in the choice of expressions. For integer-valued T, here is the *minimal* choice for receive events –for send events the minimal assignment to $T{\cdot}g$ remains $T{\cdot}g:=c$ –:

$$
\begin{array}{ll}
 & \{\ g=rcv{\cdot}m\ \wedge\ t=T{\cdot}(snd{\cdot}m)\ \} \\
 & T{\cdot}g\ :=\ c\ \mathsf{max}\ (t+1) \\
;\ & \{\ c\le T{\cdot}g\ \wedge\ t<T{\cdot}g\ \} \\
 & c\ :=\ T{\cdot}g+1 \\
;\ & \{\ T{\cdot}g<c\ \} \\
 & \Sigma_p\ :=\ \Sigma_p\cup\{\,g\,\}
\end{array}
$$

In an actual implementation the identities of the events and their time stamps need not be recorded. All that matters is that each message sent is given the proper time stamp and that, upon reception of a message, variable

c is properly adjusted. Thus, we obtain the following program fragments for send and receive events, in which *send(t)* and *receive(t)* denote the sending and receiving of a message with time stamp t –annotation omitted–:

send(c)
; $c := c{+}1$

and:

receive(t) (t is a private variable here)
; $c := c \,\mathsf{max}\, (t{+}1) + 1$

Variable c –the process's logical clock– only occurs in invariant Q1, in a formula of the shape $\cdots < c$ only. Therefore, the invariance of Q1 is not violated if we allow, "in between the events" so to speak, occasional additional increases of c. This shows that we may indeed view c as a "clock", which continues ticking even if no events take place. In this view an assignment like $c := c \,\mathsf{max}\, (t{+}1) + 1$ amounts to an "adjustment of the clock" (and a "tick") to information received about the clock of another process.

Finally, notice that the initial value of c is still irrelevant. If we choose, for all processes, $c = 0$ initially, then $T{\cdot}g$ can be interpreted as the length of a longest chain of events causally preceding g, for every event g. In this case variable c is better called an "event counter" rather than a "clock".

Chapter 2

Distributed Mutual Exclusion

In this section we derive a distributed mutual exclusion algorithm. The notion of causal precedence makes it possible to discuss the *fairness* of the solution. Informally, fairness means here that requests are granted in the order in which they are made, where "order" means causal order: if one request causally precedes another, the former request is to be granted before the latter. So, this kind of fairness requirement does not restrict the order in which concurrent – unrelated – requests are granted; nevertheless, the protocol is such that individual progress is guaranteed.

Because mutual exclusion and fairness are different properties, we can (and shall) discuss them in isolation. First, we derive an algorithm that implements mutual exclusion, without any reference to causal precedence. Second, we shall refine this solution so as to obtain fairness.

2.0 Synchronisation

For the purpose of synchronisation we use the following (tentative) construct:

$$\{\bullet \textit{ boolean expression } \bullet\} \quad ,$$

with the operational meaning that its execution terminates only if the value of *boolean expression* is `true`; the net effect on the state of the system is nihil: the synchronisation statement causes no state changes. (A correct but not so efficient implementation is repeated evaluation of *boolean expression* until its value is `true`, also known as *busy waiting*.)

Formally, a synchronisation statement can be used to *strengthen* an assertion in a program. That is, in an application like:

$\{\ Q\ \}$
$\{\bullet\ B\ \bullet\}$
$\{\ Q\ \}\{\ B\ \}$,

the additional post assertion B is *locally* correct. (And, its *global* correctness still requires additional proof, as usual.)

2.1 Mutual exclusion for two processes

We consider a collection of (sequential) processes, each of which contains a so-called *critical section*. Operationally, the requirement of mutual exclusion is that, at any moment in time, at most one of the processes is executing its critical section. This can also be phrased as the requirement that, at any moment in time, *no two* processes are executing their critical sections simultaneously. Thus, mutual exclusion of many processes can be viewed as pair-wise mutual exclusion of any two processes in the system. (It is even possible to solve the problem this way, although too naive an approach introduces the danger of deadlock.)

To keep the nomenclature needed as sober as possible, we begin with a solution for 2 processes, named p and q, only. Next, we generalize this solution to the case for many processes. Also, we use shared variables and an Owicki-Gries style of reasoning to obtain a correct solution; only after having obtained it, we will take into account that the processes are distributed and that, as a consequence, the shared variables must be implemented by means of private variables and message transmissions. So, we use a top-down approach, in which "being distributed" is considered an implementation issue, although some of the design decisions are clearly inspired by the desire to make such a distributed implementation possible.

* * *

The problem is symmetric in the processes and, as usual, we shall retain this symmetry (and exploit it in the presentation). As starting point, we use the following prototype program, in which two boolean variables x_i occur and two assertions R_i, still to be chosen, for $i \in \{p, q\}$. Every assignment and every guard in this program is considered atomic. This prototype program is (by now) well known, and it allows for many elaborations:

initially: $\quad \neg x_p \ \wedge \ \neg x_q$

program p: do forever $\rightarrow$
$\{\ \neg x_p\ \}$
$x_p := \mathsf{true}$
; $\{\ x_p\ \}$
$\{\bullet\ \neg x_q \vee R_p\ \bullet\}$
; $\{\ x_p\ \}\{\ \neg x_q \vee R_p\ \}$
Critical Section p
; $x_p := \mathsf{false}$
od

The predicates R must now be chosen in such a way that they satisfy three requirements, namely:

- The assertions in this program are *correct*, and:
- R_p and R_q are *disjoint*, to guarantee the required mutual exclusion.
- The preassertions of the synchronisation statements (together) imply $R_p \vee R_q$, so as to guarantee absence of deadlock.

example: The choice $R_p \equiv v = p$, where v is a fresh variable introduced for the purpose, gives rise to Peterson's algorithm.
□

As the first step towards the solution we introduce – this is a design decision – an integer function F on the set of process names. Because the integers are totally ordered we have:

$$F_p = F_q \ \vee\ F_p < F_q \ \vee\ F_q < F_p \ ,$$

and, more important, we also have:

$$\neg(F_p < F_q) \ \vee\ \neg(F_q < F_p) \ .$$

This yields the required disjointness, so we choose:

$$R_p \ \equiv\ F_p < F_q \quad \text{and} \quad R_q \ \equiv\ F_q < F_p \ .$$

With this choice the requirement of mutual exclusion is satisfied, so the only problems we are left with are the correctness of the assertions and the possibility that $F_p = F_q$, which gives rise to the danger of deadlock. To exclude the latter we *require* function F to be chosen in such a way that all its values are different:

$$(\forall p, q : p \neq q : F_p \neq F_q) \quad .$$

In Section 2.4 we shall deal with this requirement.

In this design, priority is given to the process with the smaller value of F. In view of the required fairness, F cannot be constant, but must be a variable. Because of the shape of the formula $\neg x_p \vee F_q < F_p$ (in program q), an assignment to F_p is harmless – with respect to the proof obligations – when it has $\neg x_p$ as its precondition. Therefore, we combine the (necessary) assignment to F_p with $x_p := \mathsf{true}$, which already has $\neg x_p$ as precondition. The value to be assigned to F_p is obtained from a private variable c_p, introduced for this purpose – this is a design decision too –; we postpone (for a while) the question how to manipulate c_p. Thus, we obtain a solution with the following structure:

initially: $\quad \neg x_p \ \wedge \ \neg x_q$

program p:
```
do forever →
        { ¬x_p }
        x_p, F_p := true, c_p
    ; { x_p }
        {• ¬x_q ∨ F_p < F_q •}
    ; { x_p } { ¬x_q ∨ F_p < F_q (•0) }
        Critical Section p
    ; x_p := false
od
```

The assertion labelled $(\bullet 0)$ is *locally correct* because of the immediately preceding guard. As for its *global correctness*, the assignment $x_q := \mathsf{false}$ (in program q) is harmless – rule of widening –, whereas the assignment (in q) $x_q, F_q := \mathsf{true}, c_q$ requires as additional precondition: $F_p < c_q$. To incorporate this additional condition we have 3 options, namely:

- make $F_p < c_q$ a system invariant;
- add $F_p < c_q$ as preassertion to $x_q, F_q := \mathsf{true}, c_q$ in program q;
- add $F_p < c_q$ as co-assertion to assertion $(\bullet 0)$ in program p.

The latter option is attractive, because it generates the least amount of new proof obligations; actually, it generates hardly any new proof obligations if we adopt the rule that the variables c will never be decreased. (This is another design decision.)

So, we adopt this rule, and we add $F_p < c_q$ as co-assertion to assertion $(\bullet 0)$ in program p; its local correctness is most easily established by including it into the preceding guard, and its global correctness now follows from the adopted rule that c_q will only be *increased* –rule of widening–.

This yields our first approximation of the solution. This solution is correct, but for the sake of progress we must still see to it that the variables c are increased "every now and then". (This will be taken care of when we construct a distributed implementation.)

initially: $\neg x_p \ \wedge \ \neg x_q$

program p:
```
do forever →
        { ¬x_p }
        x_p, F_p := true, c_p
    ; { x_p }
        {•  (¬x_q ∨ F_p < F_q) ∧ F_p < c_q  •}
    ; { x_p } { ¬x_q ∨ F_p < F_q } { F_p < c_q }
        Critical Section p
    ;  x_p := false
od
```

aside: The decision to obtain the values for F from dedicated *private* variables certainly is inspired by the desire to obtain a solution that can be implemented in a distributed way. Instead of a private variable per process we could also have introduced a single *shared* variable C; this is simpler, but from the point of view of distribution this is less attractive. Had we done so, however, we would have obtained what is known as Lamport's "Bakery Algorithm":

initially: $\neg x_p \ \wedge \ \neg x_q$

program p:
```
do forever →
        { ¬x_p }
        x_p, F_p, C := true, C, C+1
    ; { x_p } { F_p < C }
        {• ¬x_q ∨ F_p < F_q •}
    ; { x_p } { ¬x_q ∨ F_p < F_q } { F_p < C }
        Critical Section p
    ;  x_p := false
od
```

□

2.2 Distributed implementation

Process p contains as guard $(\neg x_q \vee F_p < F_q) \wedge F_p < c_q$; this expression contains one private variable, F_p, and three shared variables, x_q, F_q, and c_q; if processes p and q are to be implemented in different nodes of a distributed system this is awkward. Therefore, we decide to *delegate* the evaluation of this guard to a dedicated new process, called "co-process p,q". The idea is that this co-process will be implemented in the *same* machine as process q; as a result, (we assume that) process q and co-process p,q can share their variables without any difficulty.

This is attractive because the evaluation of p's guard takes place in states where x_p is true, whereas the assignment to F_p has $\neg x_p$ as precondition: so, during the evaluation of p's guard the value of F_p is stable. This is relevant because from the viewpoint of the co-process, F_p is a shared variable now.

Process p and co-process p,q are synchronized by means of message exchanges. First, process p sends a message containing F_p to co-process p,q and then receives a (so-called) acknowledgement, which is just an empty message. Second, after having received F_p co-process p,q establishes the required condition $(\neg x_q \vee F_p < F_q) \wedge F_p < c_q$ and then communicates this fact back to process p by sending an acknowledgement. (This form of message exchanges implements a two-phase handshake.) The conjunct $F_p < c_q$ is established (in co-process p,q) by increasing c_q sufficiently; thus, the obligation to increase the variables c "every now and then" is discharged.

This yields our second version of the solution. The correctness of the new assertions is easily verified; this is left as an exercise.

initially: $\neg x_p \ \wedge \ \neg x_q$

program p:
```
do forever →
        { ¬x_p }
        x_p, F_p := true, c_p
    ; { x_p }
        send F_p to co p,q
    ; { x_p }
        receive-acknowledgement from co p,q
    ; { x_p }{ ¬x_q ∨ F_p < F_q }{ F_p < c_q }
        Critical Section p
    ; x_p := false
od
```

co p,q: do forever $\rightarrow$
receive f from p
; $\{\ x_p\ \}\{\ f = F_p\ \}$
c_q := "a value exceeding f"
; $\{\ x_p\ \}\{\ f = F_p\ \}\{\ f < c_q$, hence: $F_p < c_q\ \}$
$\{\bullet\ \neg x_q \vee f < F_q\ \bullet\}$
; $\{\ x_p\ \}\{\ \neg x_q \vee F_p < F_q\ \}\{\ F_p < c_q\ \}$
send-acknowledgement to p
od

2.3 Many processes

Mutual exclusion of a single process with many other processes amounts to pair-wise mutual exclusion of that single process with each of the other processes in the system. Formally, this means that the conjuncts of the requirement $(\forall q : p \neq q : \neg x_q \vee R_{p,q})$ can be implemented in isolation.

Thus, by generalisation of the previous solution for two processes, we obtain the following solution for many processes. Here, for every process p a co-process *co p,q* is associated with every other process q in the system. So, if the number of processes proper in the system equals N this solution involves $N*(N-1)$ co-processes in total.

It is possible to combine, for every q, all co-processes *co p,q* into a single co-process associated with process q, thus reducing the total number of processes in the system to $2*N$, but we shall not explore this here.

initially: $(\forall q :: \neg x_q)$

program p: do forever $\rightarrow$
$\{\ \neg x_p\ \}$
x_p, F_p := true, c_p
; $\{\ x_p\ \}$
$(\ ||\, q : p \neq q :$ send F_p to *co p,q*
; receive-acknowledgement from *co p,q*
$\{\ \neg x_q \vee F_p < F_q\ \}\{\ F_p < c_q\ \}$
$)$
; $\{\ x_p\ \}\{\ (\forall q : p \neq q : (\neg x_q \vee F_p < F_q) \wedge F_p < c_q)\ \}$
Critical Section p
; x_p := false
od

co p,q: do forever $\rightarrow$
receive f from p
; $\{\ x_p\ \}\{\ f = F_p\ \}$
$c_q :=$ "a value exceeding f"
; $\{\ x_p\ \}\{\ f = F_p\ \}\{\ f < c_q$, hence: $F_p < c_q\ \}$
$\{\bullet\ \neg x_q \vee f < F_q\ \bullet\}$
; $\{\ x_p\ \}\{\ \neg x_q \vee F_p < F_q\ \}\{\ F_p < c_q\ \}$
send-acknowledgement to p
od

2.4 Keeping the values different

For every q, variables F_q and c_q and the assignment $F_q := c_q$ are local to the processes q and *co p,q*. Hence, without additional provisions we cannot guarantee that $(\forall p, q : p \neq q : F_p \neq F_q)$.

One possible way to achieve this is to use *pairs* instead of just integers: the one component of the pair is an integer and the other component is a name of a process. By means of the lexicographic order, the process names act as "tie breakers" whenever the integer components of two pairs happen to be equal. This requires, of course, that the process names are totally ordered, and this destroys the symmetry among the processes (to some extent).

So, instead of $F_q := c_q$ we now use:

$$F_q := \langle c_q, q \rangle \ ,$$

and to evaluate $F_p < F_q$ we define:

$$\langle x, p \rangle < \langle y, q \rangle \quad \equiv \quad x < y \vee (x = y \wedge p < q) \ .$$

2.5 Progress and fairness

To demonstrate progress of process p we must show that it will execute its *Critical Section* within finitely many steps after it has executed its assignment $x_p :=$ true. This boils down to the requirement that process p, within finitely many steps after the assignment $x_p :=$ true, receives an acknowledgement from *co p,q*, for all q.

First, process p sends its F_p to *co p,q*. Assuming that *co p,q* is in its initial state, ready to receive a value from p, and assuming that the communication channel from p to *co p,q* is reliable, the statement receive f from p (in *co p,q*) will terminate. Next, *co p,q* establishes $F_p < c_q$, by means of its

assignment to c_q ; as this is true for any q , we even conclude that, after finitely many steps, we have $(\forall r : p{\neq}r : F_p < c_r)$.

Second, *co p,q* will execute its send-acknowledgement to p (and return to its initial state), provided its synchronization statement terminates; if the communication channel from *co p,q* to p is reliable too, process p will indeed receive an acknowledgement from *co p,q* .

Third, we are left with the obligation to show that the synchronization statement in *co p,q* is guaranteed to terminate, which is the case if its guard $\neg x_q \vee F_p < F_q$ becomes *stably* true within finitely many steps (of the system). Because $\neg x_q \vee F_p < F_q$ is a correct, that is: stable, assertion in the program, we only have to show that it becomes true within finitely many steps. Now we consider set V , defined by:

$$r{\in}V \;\equiv\; x_r \wedge F_r < F_p \quad \text{, for all } r \quad ,$$

and we do so in a state satisfying: $(\forall r : p{\neq}r : F_p < c_r)$. Because of this, no assignment $x_r , F_r := \mathsf{true}, c_r$ adds r to V ; hence, no new members enter V anymore after $(\forall r : p{\neq}r : F_p < c_r)$ has become true . If now, by induction hypothesis, every process r in V terminates its *Critical Section* and, subsequently, completes $x_r := \mathsf{false}$ and, thus, effectively removes itself from V , set V will become empty after finitely many steps. (So, here we use mathematical induction over the values of function F .) Once V is empty we also have what we needed, namely: $\neg x_q \vee F_p < F_q$.

It is important to observe that the *latency* of the communication channel from p to *co p,q* plays a crucial role here: as long as p 's message, carrying F_p , has not yet arrived at *co p,q* , the waiting time for p cannot be bounded. Only after *co p,q* has received p 's request the number of steps taken by the system until p 's request is granted can be meaningfully considered.

Chapter 3

Epilogue

In the examples I have seen thus far, the only events in whose causal relationship we seem to be interested are the send events. It may, therefore, be easier and simpler to define causal precedence as a relation on send events only. Moreover, because of the one-to-one correspondence between send events and messages, we may as well define causal precedence as a relation on messages. So, for messages i, j, we would write $i +\!\!> j$ instead of $snd{\cdot}i \rightarrow snd{\cdot}j$. Similarly, function T could be defined directly on messages, such that $T{\cdot}i$ is the time stamp of message i.

The new causal precedence $+\!\!>$ could be defined as the transitive closure of a basic precedence relation $\cdot\!\!>$, which could be defined as follows.

basic precedence: for all messages i, j:

$$i \cdot\!\!> j \quad \equiv \quad snd{\cdot}i \rhd snd{\cdot}j \;\; \vee \;\; rcv{\cdot}i \rhd snd{\cdot}j \quad .$$

□

Now it should be possible to prove the equivalence of the two forms of causal precedence, that is: $(\forall i, j :: i +\!\!> j \;\equiv\; snd{\cdot}i \rightarrow snd{\cdot}j\,)$.

This approach seems to yield simpler formulae and deserves, therefore, further attention. For instance, the program fragment that corresponds to reception of message i with time stamp t now becomes – here variable M is the set of messages sent or received by the process –:

$$\begin{array}{ll} & \{\; t = T{\cdot}i \;\} \\ & c := c \;\mathsf{max}\; (t{+}1) \\ ;\; & \{\; T{\cdot}i < c \;\} \\ & M := M \cup \{\, i \,\} \end{array}$$

Bibliography

[1] Ö. Babaoğlu, K. Marzullo, *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*, in: S. Mullender (ed.), *Distributed Systems*, Addison-Wesley (1993, 2nd ed.).

[2] L. Lamport, *Time, Clocks, and the Ordering of Events in Distributed Systems*. Comm. ACM 21, 1978, pp. 558-565.

[3] M. Raynal, M. Singhal, *Capturing Causality in Distributed Systems*. IEEE Computer, Febr. 1996, pp. 49-56.

[4] G. Ricart, A.K. Agrawala, *An Optimal Algorithm for Mutual Exclusion*. Comm. ACM 24, 1981, pp. 9-17. [Corrigendum, Comm. ACM 24, p. 578.]