The partition of an information system in several parallel systems

THE PARTITION OF AN INFORMATION SYSTEM IN SEVERAL PARALLEL SYSTEMS

G.J. HOUBEN
J. PAREDAENS
K.M. VAN HEE

86/04

A concurrent system of a number of transaction handlers is considered. Each transaction handler sends actions to different machines and processes their reactions. Several algorithms for a serializable schedule are proposed.

October 1986

1. Introduction

In this paper we consider information systems, to which we give transactions and which partition those transactions into several actions and then order some machines belonging to the system, to take care of these actions. A transaction is considered to be a sequence of actions, where each action can be executed autonomously by a machine belonging to the system.

When we want an information system to execute a transaction, we give one of the transaction handlers of the system the responsibility for the execution of that transaction. The transaction handler therefore sends all the actions of that transaction to machines of the system, asking the machines to execute the action. We suppose that a transaction handler handles at any time at most one transaction. So when a transaction handler is handling a transaction, it can only handle a second transaction after the execution of the first one is completed, that is for all actions the transaction handler has received a reaction (an answer) from a machine.
We suppose that every transaction handler always knows which transaction it has to handle, and for every action it knows which machines are able to execute that action.

Machines can execute actions, since they are able to make computations and to store and update information. When executing a functional action a machine makes computations, which do not depend on the information stored in the machine, and it sends the result of those computations as a reaction to the transaction handler (that can use this information in the next actions).
When executing a view action information is gathered from the information stored in the machine and that information is sent as a reaction to the transaction handler.
When executing an update action the information stored in the machine will be updated. After the execution of an update action a reaction is sent back to the transaction handler. This reaction will contain information concerning the validity of the update.

The scheduler of a system is responsible for the interface between the transaction handlers and the machines. Therefore it executes some schedule, which task is to guarantee a proper information handling.

In section 2 we introduce simple and distributed information systems. In distributed information systems the scheduler has to execute a serializable schedule. In section 3 we present a serializable schedule that is rather obvious and has timestamping as a key issue. Then in section 4 a timestampless serializable schedule is presented. Subsequently the communication involved in the execution of this schedule is examined more closely in section 5.

2. Information systems

Before introducing the notion of distributed information systems, we first turn our attention to simple information systems.

A simple information system is a triple (*th*, *m*, *s*), where *th* is a transaction handler, *m* is a machine and *s* is a scheduler.

At any moment the transaction handler *th* is in one of three states, depending on what it is doing. If *th* is not handling a transaction at all, then *th* is in the state asleep, denoted by $\underline{A}$. If *th* is gathering information needed for the transaction, then *th* is in the state information gathering, denoted by $\underline{VF}$. If *th* is updating information, as a consequence of the transaction, then *th* is in the state updating, denoted by $\underline{U}$. We write *th* is active, if *th* is in $\underline{VF}$ or in $\underline{U}$.
While *th* is in $\underline{VF}$ it can send view actions and functional actions to the machine *m*, but it cannot send update actions to *m*. While *th* is in $\underline{U}$, it can send update actions to *m*, but it cannot send functional or view actions to *m*. While *th* is in $\underline{A}$, it cannot send any action at all.

The machine *m* executes each action *th* sends to *m*. It can execute at most one action at a time. After it finishes the execution of a functional or a view action, it sends to *th* a reaction containing the result of that action, that is the information resulting from computing and retrieving information stored in *m*. After finishing the execution of an update action, that is after modifying the information stored in *m*, it sends to *th* a reaction containing information about the validity of the update.

The scheduler *s* is responsible for the interface between *th* and *m*. It is therefore executing a schedule that controls the state transition of *th* to be $\underline{A} \rightarrow \underline{VF} \rightarrow \underline{U} \rightarrow \underline{A}$ and that allows the information flow between *th* and *m* to be as described above.

Now we turn to distributed information systems. A distributed information system is a triple (*TH*, *M*, *s*), where *TH* is a finite set of transaction handlers, *M* is a finite set of machines and *s* is a scheduler.

At any moment each transaction handler *th* of *TH* is in one of three states, depending on what it is doing. If *th* is not handling a transaction at all, then *th* is in the state asleep, denoted by $\underline{A}$. If *th* is gathering information needed for the transaction, then *th* is in the state information gathering, denoted by $\underline{VF}$. If *th* is updating information, as a consequence of the transaction, then *th* is in the state updating, denoted by $\underline{U}$. We write *th* is active, if *th* is in $\underline{VF}$ or in $\underline{U}$.
While *th* is in $\underline{VF}$, it can send view actions and functional actions to each machine *m* of *M*, but it cannot send update actions to any *m* of *M*. While *th* is in $\underline{U}$, it can send update actions to each machine *m* of *M*, but it cannot send view nor functional actions to any *m* of *M*. While *th* is in $\underline{A}$, it cannot send any action at all.

Each machine *m* of *M* executes each action sent to *m* by a *th* of *TH*. It can execute at most one action at a time. After it finishes the execution of a functional or a view action. it sends back to *th*, a reaction containing the result of that action, that is the information resulting from computing and retrieving information stored in *m*. After finishing the execution of an update action, that is after modifying the information stored in *m*, it sends to *th* a reaction containing information about the validity of the update on *m*.

The scheduler *s* is responsible for the interface between the transaction handlers of *TH* and the machines of *M*. It therefore executes a schedule that controls the state transition of each *th* of *TH* to be $\underline{A} \rightarrow \underline{VF} \rightarrow \underline{U} \rightarrow \underline{A}$ and that allows the information flow between each transaction handler and each machine to be as described above and that is serializable.

We now present a small example, which should illustrate some of the notions mentioned above.

Suppose our information system consists of two transaction handlers, so $TH = \{th_1, th_2\}$, and five machines, so $M = \{m_0, m_1, m_2, m_3, m_4\}$. In our machine m_1 we have the addresses and ages of employees. In m_2 are the medical records of employees, and the salaries are in m_3. Machine m_4 is very good in computing the square of a natural number. Machine m_0 has some special task, which is not important in this example.

In some informal way we now describe three transactions.

Let t_1 be :

> get for every employee his age;
> get for every employee older than 60 his medical record.

When t_1 is handled by th_1, then th_1 has to send a view action to m_1 first. Machine m_1 will send a reaction back to th_1, and a part of the information contained in this reaction will be used by th_1 to initiate a view action at m_2. When m_2 has sent a reaction to th_1 then th_1 has all the information required by t_1.

Let t_2 be :

> get for every employee his address;
> add for every employee living in Eindhoven 1000 to his salary.

When t_2 is handled by th_2, then th_2 has to send a view action to m_1 first. It will get a reaction from m_1 and depending on this reaction it will send an update action to m_3. After m_3 has sent a reaction to th_2, th_2 has done everything required by t_2.

Let t_3 be :

> compute $c = 7^2$;
> add c to every salary.

When t_3 is handled by th_1, first a functional action will be sent by th_1 to m_4. The reaction of m_4 will be used to initiate an update action at m_3. Work on t_3 will be finished after a reaction from m_3 is received by th_1.

3. A Serializable Schedule

From the previous section we know that the schedule we need for a distributed information system must be serializable.
What are serializable schedules ?
First we define serial schedules. When a serial schedule is executed no two transaction handlers are active at the same moment. Hence with a serial schedule there is a function $th : \mathbb{N} \rightarrow TH$ indicating the order in which the transaction handlers of TH are active.
Two schedules are equivalent if and only if they both result in the same information transition, which means that for every possible state of the information stored in the machines of M the final state will be the same. A serializable schedule is a schedule that is equivalent to a serial schedule. Of course, the most easy kind of serializable schedules are the serial schedules.

We now specify a schedule *TSS*, for which we prove that it is equivalent to the serial schedule *TSO* where transactions are handled in, the order of their entrance in the system, thus of their timestamp.

Note that we suppose that from a transaction t the sets of machines, which have to execute functional actions, view actions and update actions respectively in order to execute t, can be computed.

<u>*SCHEDULE*</u> <u>*TSS*</u> :

Suppose <u>*th*</u> is a transaction handler of TH and <u>*t*</u> is a transaction that <u>*th*</u> has to handle. Let $t : \mathbb{N} \rightarrow T$, with T the set of all transactions, indicating the order in which the transactions entered the system.

- When <u>*th*</u> goes, in order to execute <u>*t*</u>, from <u>*A*</u> to <u>*VF*</u>, the scheduler s gives a timestamp, say j, which is one higher than the previous one, so <u>*th*</u> $= th(j)$ and <u>*t*</u> $= t(j)$, and s calculates from $t(j)$ F_j, V_j, U_j :
  $F_j :=$ { machines to which $th(j)$ will send functional actions in order to execute $t(j)$ }
  $V_j :=$ { machines to which $th(j)$ will send view actions in order to execute $t(j)$ }
  $U_j :=$ { machines to which $th(j)$ will send update actions in order to execute $t(j)$ }

- Before $th(j)$ sends, in order to execute $t(j)$, a view action to a machine m, $th(j)$ waits until m does not belong to $\bigcup_{i<j} U_i$.

- When $th(j)$ goes, while executing $t(j)$, from <u>*VF*</u> to <u>*U*</u>, then :
  $F_j := \varnothing$
  $V_j := \varnothing$

- Before $th(j)$ sends, in order to execute $t(j)$, an update action to a machine m, $th(j)$ waits until m does not belong to $\bigcup_{i<j} (V_i \cup U_i)$

- When $th(j)$ goes after executing $t(j)$, from <u>*U*</u> to <u>*A*</u>, then :
  $U_j := \varnothing$

<u>*END*</u> <u>*SCHEDULE*</u> <u>*TSS*</u>

Of course, F_j can be omitted from this schedule.

We will now prove the serializability of *TSS* in showing the equivalence with the (serial) schedule *TSO* in which the transaction handlers execute the transactions in order of their timestamp, that is if $i<j$, then $t(i)$ is executed (by $th(i)$) before $t(j)$ is executed (by $th(j)$), so $th(i)$ is active (with $t(i)$) before

$th(j)$ is active (with $t(j)$).

Let m be a machine of M, and let $i<j$ and $t(i)$ the transaction to be handled by $th(i)$ and $t(j)$ the transaction to be handled by $th(j)$.

- When m is in at most one of the sets V_i, V_j, U_i and U_j, then there is no problem, since it is easy to see that executing first $t(i)$ then $t(j)$ would have the same effect (results in the same state of the information in m).

- Of course, there is no problem either, when m only belongs to both V_i and U_i or only to both V_j and U_j.

- When m only belongs to both V_i and V_j then there is no problem, since $th(i)$ does not change anything in m.

- When m belongs to both V_j and U_i, but not to U_j, then, since m in U_i and $i<j$, $th(j)$ will send a view action to m after $U_i := \varnothing$, so when $th(i)$ is not executing $t(i)$ anymore.

- When m belongs to both V_i and U_j, but not to U_i, then, since m in V_i and $i<j$, $th(j)$ will send an update action to m after $V_i := \varnothing$, so when $th(i)$ is not in <u>*VF*</u> anymore as far as $t(i)$ is concerned.

- When m belongs to both U_i and U_j, but not to V_j, then, since m in U_i and $i<j$, $th(j)$ will send an update action to m after $U_i := \varnothing$ (V_i is already empty at that moment or m was not in V_i), so when $th(i)$ is not executing $t(i)$ anymore.

- When m belongs to both V_j, U_i and U_j, then, since m in U_i and $i<j$, $th(j)$ will send a view action to m after $U_i := \varnothing$ (m is not in V_i (anymore)), so when $th(i)$ is not executing $t(i)$ anymore.

This ends the proof of the serializability.
We therefore have a schedule *TSS* that fulfills the conditions for the schedule of a distributed information system and in which the timestamps are the key issue. In the next section we will present another schedule (that of course fulfills those conditions), that will not make use of timestamps.

4. A Timestampless Serializable Schedule

We will now give another specification of a serializable schedule. The main advantage of this schedule will be the absence of timestamps.
What is this main advantage ? Since at each moment we only have to deal with the transactions being handled in the system at that time, we only have to assign to these transactions some unique number.
In the timestamping approach however, all transactions that ever were handled in the system must have some unique number. Obviously this implies an infinite set of numbers being used. Also, selecting numbers satisfying some condition can be done much easier with some rather small, finite set than with some infinite set of numbers.
Furthermore, in the timestamping approach there must be some central system, that assigns the timestamps, in order to guarantee the global unicity of the timestamps. In the timestampless approach we do not need such a global clock, so the information system consists only of transaction handlers and machines.

First though, we consider a schedule, called *TS*, that uses timestamps. After proving that this schedule is a schedule for a distributed information system, we will show that in this schedule the timestamps are not really needed and can therefore be omitted, thus obtaining a timestampless serializable schedule for a distributed information system. We will call this timestampless schedule *TSL*.

We now specify a schedule *TS* (that uses timestamps) for which we prove that it is a correct schedule for a distributed information system. The serializability of *TS* is proven by showing (indirectly) the equivalence to the schedule *TSO* where transactions are serially handled in the order of their timestamp.

Intuitively V_i will be the set of machines which get a view action belonging to $t(i)$, U_i will be the set of machines which get an update action belonging to $t(i)$, $V(m)$ will be the number of transaction handlers that need to view machine m, $U(m)$ will be the number of transaction handlers that need to update machine m, $UV_i(m)$ will be the number of transaction handlers that need to update m before $th(i)$ can view m, $AU_i(m)$ will be the number of transaction handlers that need to view or update m before $th(i)$ can update m.

<u>*SCHEDULE*</u> <u>*TS*</u> :

Initialize V_i and U_i to be $\varnothing$ for all i, and $V(m)$, $U(m)$, $UV_i(m)$, $AU_i(m)$ to be 0 for all i and all m of M.

Suppose $\underline{th}$ is a transaction handler of *TH* and $\underline{t}$ is a transaction $\underline{th}$ has to handle.

- When $\underline{th}$ goes, in order to execute $\underline{t}$, from $\underline{A}$ to $\underline{VF}$, the scheduler s gives a timestamp, say j, which is one higher than the previous one, so $\underline{th} = th(j)$ and $\underline{t} = t(j)$, and calculates :
  V_j := { machines to which $th(j)$ will send view actions in order to execute $t(j)$ }
  U_j := { machines to which $th(j)$ will send update actions in order to execute $t(j)$ }
  $UV_j(m) := U(m)$ for all m in V_j
  $AU_j(m) := V(m) + U(m)$ for all m in U_j
  $V(m) := V(m) + 1$ for all m in V_j
  $U(m) := U(m) + 1$ for all m in U_j

- Before $th(j)$ sends, in order to execute $t(j)$, a view action to a machine m, $th(j)$ waits until $UV_j(m) = 0$.

- When $th(j)$ goes, while executing $t(j)$, from <u>*VF*</u> to <u>*U*</u>, then :
  $AU_i(m) := AU_i(m) - 1$ if $AU_i(m) > 0$, for all $i \neq j$ and all m of V_j
  $V(m) := V(m) - 1$ for all m of V_j
  $V_j := \varnothing$

- Before $th(j)$ sends, in order to execute $t(j)$, an update action to a machine m, $th(j)$ waits until $AU_j(m) = 0$.

- Before $th(j)$ goes, after executing $t(j)$, from <u>*U*</u> to <u>*A*</u>, then :
  $UV_i(m) := UV_i(m) - 1$ if $UV_i(m) > 0$, for all $i \neq j$ and all m of U_j
  $AU_i(m) := AU_i(m) - 1$ if $AU_i(m) > 0$, for all $i \neq j$ and all m of U_j
  $U(m) := U(m) - 1$ for all m of U_j
  $U_j := \varnothing$

<u>*END*</u> <u>*SCHEDULE*</u> <u>*TS*</u>

It is trivial to prove that at each moment $V(m)$ is the number of transaction handlers $th(i)$ with m in V_i, and $U(m)$ is the number of transaction handlers $th(i)$ with m in U_i.
$UV_j(m)$ is the number of $th(i)$ with $i<j$ and m in both U_i and V_j. $AU_j(m)$ is the sum of the number of $th(i)$ with $i<j$ and m in both V_i and U_j, and the number of $th(i)$ with $i < j$ and m in both U_i and U_j.

To demonstrate the correctness of this schedule *TS*, we consider the next schedule *TSB*, that obviously controls the state transition of each transaction handler to be <u>*A*</u> $\rightarrow$ <u>*VF*</u> $\rightarrow$ <u>*U*</u> $\rightarrow$ <u>*A*</u> and allows the information flow between each transaction handler and each machine to be as described in section 2.

We will use mV_iU_j and mU_iU_j, where mV_iU_j is 1, if m belongs to V_i and U_j, and 0 else, and mU_iU_j is 1, if m belongs to U_i and U_j, and 0 else.

<u>*SCHEDULE*</u> <u>*TSB*</u> :

Initialize V_i and U_i to be empty for all i and mV_iU_j and mU_iU_j to be 0 for all m, i and j.

Suppose <u>*th*</u> is a transaction handler of *TH* and <u>*t*</u> is a transaction <u>*th*</u> has to handle.

- When <u>*th*</u> goes, in order to execute <u>*t*</u>, from <u>*A*</u> to <u>*VF*</u>, the scheduler s gives a timestamp, say j, which is one higher than the previous one, so <u>*th*</u> $= th(j)$ and <u>*t*</u> $= t(j)$, and s calculates from $t(j)$:
  $V_j :=$ { machines to which $th(j)$ sends view actions in order to execute $t(j)$ }
  $U_j :=$ { machines to which $th(j)$ sends update actions in order to execute $t(j)$ }
  for all m of V_j
      $mV_jU_i := 1$ for all i with m in U_i and $i<j$
  for all m of U_j
      $mU_jU_i := mU_iU_j := 1$ for all i with m in U_i and $i<j$
      $mV_iU_j := 1$ for all i with m in V_i and $i<j$

- Before $th(j)$ sends, in order to execute $t(j)$, a view action to a machine m, $th(j)$ waits until $mV_jU_i = 0$ for all $i<j$.

- When $th(j)$ goes, while executing $t(j)$, from VF to U, then :
  $mV_jU_i := 0$ for all i and all m of V_j
  $V_j := \varnothing$

- Before $th(j)$ sends, in order to execute $t(j)$, an update action to a machine m, $th(j)$ waits until $mV_iU_j = mU_iU_j = 0$ for all $i<j$.

- When $th(j)$ goes after executing $t(j)$, from U to A, then :
  $mU_jU_i := mV_iU_j := mU_iU_j := 0$ for all i and m of U_j
  $U_j := \varnothing$

END SCHEDULE TSB

We will now prove the serializability of *TSB* and then by showing the equivalence between *TS* and *TSB*, we will prove the serializability of *TS*.

We will show that *TSB* is equivalent to the schedule *TSO* in which the transaction handlers execute the transactions in order of their timestamp, that is if $i<j$, then $t(i)$ is executed before $t(j)$, so $th(i)$ is active before $th(j)$.

Let m be a machine of M, and let $i<j$ and $t(i)$ the transaction to be handled by $th(i)$ and $t(j)$ the transaction to be handled by $th(j)$.

- When m is in at most one of the sets V_i, V_j, U_i and U_j, then there is no problem, since it is easy to see that executing first $t(i)$ then $t(j)$ would have the same effect.

- Of course, there is no problem either, when m only belongs to both V_i and U_i or only to both V_j and U_j.

- When m only belongs to both V_i and V_j then there is no problem, since $th(i)$ does not change anything in m.

- When m belongs to both V_j and U_i, but not to U_j, then, since m in U_i and $i<j$, $th(j)$ will send a view action to m after $mV_jU_i := 0$, so when $th(i)$ is not executing $t(i)$ anymore.

- When m belongs to both V_i and U_j, but not to U_i, then, since m in V_i and $i<j$, $th(j)$ will send an update action to m after $mV_iU_j := 0$, so when $th(i)$ is not in VF anymore as far as $t(i)$ is concerned.

- When m belongs to both U_i and U_j, but not to V_j, then, since m in U_i and $i<j$, $th(j)$ will send an update action to m after $mU_iU_j := 0$ (mV_iU_j is (already) 0 at that moment), so when $th(i)$ is not executing $t(i)$ anymore.

- When m belongs to both V_j, U_i and U_j, then, since m in U_i and $i<j$, $th(j)$ will send a view action to m after $mV_jU_i := 0$ (then $mU_jU_i = 0$ and $mV_iU_j = 0$), so when $th(i)$ is not executing $t(i)$ anymore.

So we have proven the serializability of *TSB*.

It is rather trivial to prove that the following are invariants :
$mV_iU_j = 1$ iff m in V_i and m in U_j;
$mU_iU_j = 1$ iff m in U_i and m in U_j.

We will show the equivalence between both schedules, by proving Q where Q is : $UV_j(m) = \sum_{i<j} mV_jU_i$ and $AU_j(m) = \sum_{i<j}(mV_iU_j + mU_iU_j)$.

It is trivial that Q holds at initialization.

When $th(j)$ goes from $\underline{A}$ to $\underline{VF}$, Q holds if Q' holds, where Q' stands for :
at the moment $th(j)$ goes from $\underline{A}$ to $\underline{VF}$, $V(m)$ is the number of $th(i)$ with $i<j$ and m in V_i, and $U(m)$ is the number of $th(i)$ with $i<j$ and m in U_i.
It is trivial to prove that Q' holds.

With Q it is clear that the conditions for which $th(j)$ has to wait before sending view or update actions, are the same in both schedules.

When $th(j)$ goes from $\underline{VF}$ to $\underline{U}$ mV_jU_i becomes 0 for all i and m of V_j. mV_jU_i was 1 only if m in V_j and m in U_i and $i>j$. This follows from : mV_jU_i only became 1 if m in both V_j and U_i, and if $i<j$ then mV_jU_i already has become 0, since this was the condition for which $th(j)$ was waiting before sending a view action.
So mV_jU_i changed from 1 to 0 if m in V_j and m in U_i and $i>j$. Therefore for all i and m of V_j, $AU_i(m)$ has to decrease by 1 (if possible), in order to keep Q invariant, since $AU_i(m) = \sum_{l<i}(mV_lU_i + mU_lU_i)$ and in the set of mV_lU_i and mU_lU_i with $l<i$, there is only one mV_lU_i that changed from 1 to 0.

When $th(j)$ goes from $\underline{U}$ to $\underline{A}$ mV_iU_j, mU_iU_j and mU_jU_i become 0 for all i and m of U_j. mV_iU_j was 1 only if m in V_i and m in U_j and $i>j$. Therefore for all i and m of U_j $UV_i(m)$ has to decrease by 1 (if possible), in order to keep Q invariant, since $UV_i(m) = \sum_{l<i} mV_iU_l$ and in the set of mV_iU_l with $l<i$ there is only one mV_iU_l that changed from 1 to 0.
$mU_iU_j = mU_jU_i$ was 1 only if m in U_i and m in U_j and $i>j$. Therefore for all i and m of U_j $AU_i(m)$ has to decrease by 1 (if possible), in order to keep Q invariant, since $AU_i(m) = \sum_{l<i}(mV_lU_i + mU_lU_i)$ and in the set of mV_lU_i and mU_lU_i with $l<i$ there is only one mU_lU_i that changed from 1 to 0.

Therefore Q holds.

The claim was that from the schedule TS, we could derive a schedule TSL, that, in contrast to TS, would certainly not make use of timestamps.

Before specifying TSL, we will define the following :

There is some "super transaction handler" that assigns to each transaction that enters the system a transaction handler on which the transaction will be handled.
TID is the set of transaction id-numbers. We denote the transaction with id-number i by t_i.
$THID$ is the set of transaction handler id-numbers. We denote the transaction handler with id-number i by th_i.
$\theta : TID \rightarrow THID$ assigns to each transaction id-number the id-number of the transaction handler that will handle the transaction with that id-number.
In this information system $TH = \{th_1,..,th_k\}$ and $THID = \{1,..,k\}$.

Now we will specify *TSL*.

<u>*SCHEDULE*</u> <u>*TSL*</u> :

Initialize V_i and U_i to be ∅ for all i of *THID*, and $V(m)$, $U(m)$, $UV_i(m)$ and $AU_i(m)$ to be 0 for all i of *THID* and all m of M.

Suppose th_j has to handle t_l, so $j = \theta(l)$.

- When th_j goes in order to execute t_l from <u>A</u> to <u>VF</u>, then s calculates from t_l :
  V_j := { machines to which th_j sends view actions in order to execute t_l ($j = \theta(l)$) }
  U_j := { machines to which th_j sends update actions in order to execute t_l ($j = \theta(l)$) }
  $UV_j(m) := U(m)$ for all m in V_j
  $AU_j(m) := V(m) + U(m)$ for all m in U_j
  $V(m) := V(m) + 1$ for all m in V_j
  $U(m) := U(m) + 1$ for all m in U_j

- Before th_j sends, in order to execute t_l, a view action to a machine m, th_j waits until $UV_j(m) = 0$.

- When th_j goes, while executing t_l, from <u>VF</u> to <u>U</u>, then :
  $AU_i(m) := AU_i(m) - 1$ if $AU_i(m) > 0$, for all i of $THID \backslash \{j\}$ and all m of V_j
  $V(m) := V(m) - 1$ for all m of V_j
  $V_j := \varnothing$

- Before th_j sends, in order to execute t_l, an update action to a machine m, th_j waits until $AU_j(m) = 0$.

- When th_j goes, after executing t_l, from <u>U</u> to <u>A</u>, then :
  $UV_i(m) := UV_i(m) - 1$ if $UV_i(m) > 0$, for all i of $THID \backslash \{j\}$ and all m of U_j
  $AU_i(m) := AU_i(m) - 1$ if $AU_i(m) > 0$, for all i of $THID \backslash \{j\}$ and all m of U_j
  $U(m) := U(m) - 1$ for all m of U_j
  $U_j := \varnothing$

<u>*END*</u> <u>*SCHEDULE*</u> <u>*TSL*</u>

We now claim that *TS* and *TSL*, as we just specified, are in fact the same schedule, since the only difference between them is the fact that where in *TS* timestamps are mentioned, in *TSL* transaction (handler) id-numbers are mentioned. And when we observe *TS*, we can see that in *TS* we did not use any aspect of the timestamps other than the identification of transactions and transaction handlers. So we can replace the timestamps by id-numbers. Therefore *TS* and *TSL* are quite the same schedule.
So *TSL* is a timestampless serializable schedule for a distributed information system.

5. The Communication in the Timestampless Serializable Schedule

We now consider some aspects of the communication involved in the execution of the timestampless serializable schedule *TSL* of the previous section.

First of all we consider an obvious problem. When a transaction handler wants to send an action to a machine, it is possible that that machine is at that moment busy executing some other action (of another transaction handler).

Therefore we define for every machine m a boolean $busy(m)$, that of course will denote whether m is busy at that moment or not. Then we want a transaction handler th only to send an action to m (whatever type of action it is) if m is busy with th. This means that, th found, as it inspected $busy(m)$ exclusively, $busy(m)$ to be false and then set it to true itself.
Later on we will show how $busy(m)$ should be used in detail.

Now we turn to the communication (and the computations implied) needed when executing the schedule.

Suppose we have a transaction handler th_j, that has to handle t_l, with $j = \theta(l)$. When th_j goes from $\underline{A}$ to $\underline{VF}$, first V_j and U_j have to be computed. For these computations only t_l, the transaction that th_j has to handle, is needed, since from t_l th_j can learn which machines can get what kind of actions. Furthermore, $U(m)$ for m in $V_j \cup U_j$ and $V(m)$ for m in $V_j \cup U_j$ are needed in order to be able to compute $UV_j(m)$ for all m, $AU_j(m)$ for all m, $V(m)$ for all m in V_j and $U(m)$ for all m in U_j.

When th_j is in $\underline{VF}$ and it wants to send a functional action to a machine m, it has to know whether m is busy or not, therefore it needs $busy(m)$. (It may need $busy(m)$ even several times, when it finds $busy(m)$ to be true for a number of times, since we decide to retry when we find m not to be ready for th_j.)

When th_j is in $\underline{VF}$ and it wants to send a view action to a machine m, it has first of all to know whether $UV_j(m) = 0$, so $UV_j(m)$ is needed. When it finds out that $UV_j(m) = 0$ holds, and it therefore decides that the action can be sent, it has to wait perhaps until m is not busy anymore, so $busy(m)$ is also needed (perhaps several times).

When th_j goes from $\underline{VF}$ to $\underline{U}$, $AU_i(m)$ for m in V_j and $i \neq j$, and $V(m)$ for m in V_j will get new values based on the old values. Further, V_j gets a new value, but since that value is a trivial one, we do not need any value from outside.

When th_j is in $\underline{U}$ and it wants to send an update action to a machine m, it has first of all to know whether $AU_j(m) = 0$, so $AU_j(m)$ is needed. When it finds out that $AU_j(m) = 0$ holds, and it therefore decides that the action can be sent, it has to wait perhaps until m is not busy anymore, so $busy(m)$ is also needed (perhaps several times).

When th_j goes from $\underline{U}$ to $\underline{A}$, $UV_i(m)$ for m in U_j and $i \neq j$, $AU_i(m)$ for m in U_j and $i \neq j$, and $U(m)$ for m in U_j will get new values based on the old values. Further, U_j gets a new value, but this is a trivial value : $\varnothing$.

So we have :

Input for th_j	Output for th_j
t_l	V_j
t_l	U_j
$U(m)$ (m in V_j)	$UV_j(m)$ (all m)
$V(m)$ and $U(m)$ (m in U_j)	$AU_j(m)$ (all m)
$V(m)$ (m in V_j)	$V(m)$ (m in V_j)
$U(m)$ (m in U_j)	$U(m)$ (m in U_j)
$busy(m)$	$busy(m)$
$UV_j(m)$	
$AU_i(m)$ (m in V_j, $i \neq j$)	$AU_i(m)$ (m in V_j, $i \neq j$)
$V(m)$ (m in V_j)	$V(m)$ (m in V_j)
	V_j
$AU_j(m)$	
$UV_i(m)$ (m in U_j, $i \neq j$)	$UV_i(m)$ (m in U_j, $i \neq j$)
$AU_i(m)$ (m in U_j, $i \neq j$)	$AU_i(m)$ (m in U_j, $i \neq j$)
$U(m)$ (m in U_j)	$U(m)$ (m in U_j)
	U_j

From this we can conclude that t_l is only read by th_j, therefore we can easy store the transaction at the transaction handler that is handling it. So we can see the transaction inside the transaction handler as some plan as what to do next.

Further V_j and U_j are also only read by th_j, whereas $UV_j(m)$ and $AU_j(m)$ can be read and modified by any transaction handler th_p. Therefore it is convenient to store at least $UV_j(m)$ and $AU_j(m)$ in a place where we can give every th_p easy access, but also where we are able to guarantee mutual exclusive access. Because we do not want to leave the burden of this for the machine m, we suppose a central information unit where all $UV_j(m)$ and $AU_j(m)$ are stored. Therefore we extend our information system with a machine m_0 and m_0 will serve as this central information unit.
As far as V_j and U_j are concerned, for reasons of efficiency, we will also store them at m_0. More about (using) m_0 later.

From the table above, we can also conclude that $V(m)$ and $U(m)$ can be read and modified by any transaction handler, and for the same reasons as hold for $UV_j(m)$ and $AU_j(m)$ we will store all $V(m)$ and $U(m)$ at m_0.

Of course, $busy(m)$ can also be read and modified by any transaction handler th_p, but since we want to give machine m the possibility of mingling with $busy(m)$, we prefer to store $busy(m)$ at the machine m, that is at a place where m can easy access $busy(m)$. We then have the burden of explicitly taking care of mutual exclusion (as we will see later).

Before we explicitly describe what a transaction handler has to do, we turn our attention to what a machine has to do as far as communication is concerned.
For instance, what does a machine m have to do after executing an action ?
After executing any action for th_j, m must send a reaction back to th_j. Before it can really send the reaction to th_j it must be sure that th_j is able to receive the reaction, since th_j could be busy in for example the sending of another action. Therefore we suppose a boolean $busy(th_j)$ that, analogous to $busy(m)$, denotes whether th_j is busy or not. So m must wait until $busy(th_j)$ is set to true by m itself before sending the reaction to th_j.
Furthermore after the reaction has been sent to th_j, m has to set $busy(m)$ to false again to be able to accept another action. (Of course, this changing of $busy(m)$ has to happen mutually exclusive.)

As we have just seen we need something like *busy*(th_j) at th_j.
Surely *busy*(th_j) must be true whenever th_j is doing anything for which it has mutually exclusive access of some resource (a machine or itself). It may be not busy, that is able to receive a reaction if it is just working "internally" or is perhaps waiting for a machine to become not busy. (When a reaction is received while doing "internal" work, the transaction handler has to know what it was doing to be able to continue that internal work after the reception of the reaction. We will come back to this later.)

We will now describe what the transaction handlers and the machines have to do in order to guarantee a proper information flow, as is described. We do this in some program-like notation in which we use claim and release to guarantee mutual exclusion :

claim(x): of all processes that want exclusive control of x, the one that has waited the longest gets control when that is possible: "wait until it is your turn".

release(x): give up the control of x you have, and thus give another process the possibility of completing its claim(x), that is getting the control of x.

At th_j the boolean *arrived−from*(m) will be true if and only if m has sent a reaction to th_j, which th_j has not yet accepted.

The procedure there-is-a-problem will signal that some update action took place, but something has gone wrong (the constraints are not satisfied anymore, for instance).

When a reaction r is sent back to th_j, we suppose that in *r.info* the information of the reaction is contained. The information of the reaction is either the result (answer) of the action if it was a functional or view action, or the message saying that the update did or did not take place successfully if the action was an update action.

The rest of the notations will be rather straightforward.

First we introduce two procedures action-send and reaction-receive, which only use is to simplify the program texts.

```
procedure action-send(th,m,a);
var ready, sent : boolean;
begin ready:=false;
      while not ready
      do claim(th);
         if not busy(th)
         then busy(th):=true;
              release(th);
              sent:=false;
              while not sent
              do claim(m);
                 if not busy(m)
                 then busy(m):=true;
                      release(m);
                      send(m,a);
                      sent:=true;
                 else release(m)
                 fi
              od;
              ready:=true;
              claim(th);
              busy(th):=false;
```

```
            release(th)
        else release(th)
        fi
    od
end
```

So action-send(th, m, a) will result in action a being sent to machine m by transaction handler th.

```
procedure reaction-receive(th,m,r,x);
var received : boolean;
begin while not arrived-from(m)
      do skip
      od;
      received:=false;
      while not received
      do claim(th);
         if not busy(th)
         then busy(th):=true;
              release(th);
              receive(m,r);
              arrived-from(m):=false;
              received:=true
         else release(th)
         fi
      od;
      x:=r.info;
      claim(th);
      busy(th):=false;
      release(th)
end
```

So reaction-receive(th, m, r, x) will result in th accepting reaction r from m in such a way that x will contain the information from r.

Now we describe what th_j has to do.

When th_j goes from <u>A</u> to <u>VF</u>, then it has to perform :

```
action-send(th_j,m_0,u_0);
reaction-receive(th_j,m_0,r,x);
if x≠OK
then there-is-a-problem
fi

where u_0 could be specified as :
compute(V_j);
compute(U_j);
for m in V_j
do  UV_j(m) := U(m);
    V(m) := V(m) + 1
od;
for m in U_j
do  AU_j(m) := V(m) + U(m);
    U(m) := U(m) + 1
```

```
od
```

When th_j wants to send a functional action f to a machine m, then it has to perform :

```
action-send(th_j, m, f);
reaction-receive(th_j, m, r, x)
% x contains the answer %
```

When th_j wants to send a view action v to a machine m, then it has to perform :

```
condition := false;
while not condition
do action-send(th_j, m_0, v_0);
   reaction-receive(th_j, m_0, r, x);
   if x
   then condition := true
   fi
od;
action-send(th_j, m, v);
reaction-receive(th_j, m, r, x)
% x contains the answer %

where v_0 could be specified as :
UV_j(m) = 0 ?
```

When th_j goes from $\underline{VF}$ to $\underline{U}$, then it has to perform :

```
action-send(th_j, m_0, u_1);
reaction-receive(th_j, m_0, r, x);
if x≠OK
then there-is-a-problem
fi

where u_1 could be specified as :
for m in V_j and i≠j
do if AU_i(m) > 0
   then AU_i(m) := AU_i(m) - 1
   fi
od;
for m in V_j
do V(m) := V(m) - 1
od
```

When th_j wants to send an update action u to a machine m, then it has to perform :

```
condition := false;
while not condition
do action-send(th_j, m_0, v_1);
   reaction-receive(th_j, m_0, r, x);
   if x
   then condition := true
   fi
od;
action-send(th_j, m, u);
```

```
reaction-receive(th_j,m,r,x);
if x≠OK
then there-is-a-problem
fi

where v_1 could be specified as :
AU_j(m) = 0 ?
```

When th_j goes from $\underline{U}$ to $\underline{A}$, then it has to perform :

```
action-send(th_j,m_0,u_2);
reaction-receive(th_j,m_0,r,x);
if x≠OK
then there-is-a-problem
fi

where u_2 could be specified as :
for m in U_j and i≠j
do if AU_i(m) > 0
   then AU_i(m) := AU_i(m) - 1
   fi;
   if UV_i(m) > 0
   then UV_i(m) := UV_i(m) - 1
   fi
od;
for m in U_j
do U(m) := U(m) - 1
od
```

What does m have to do as its part of the communication ?
After the execution of an action for th_j, m has to send the reaction r back to th_j, when th_j is busy with m, and it has to set $busy(m)$ to false again afterwards. Of course, m has to notify th_j that a reaction has arrived in setting $arrived-from(m)$ to true. Therefore it has to perform :

```
sent := false;
while not sent
do claim(th_j);
   if not busy(th_j)
   then busy(th_j) := true;
        release(th_j);
        send(th_j,r);
        sent := true;
        claim(th_j);
        busy(th_j) := false;
        arrived-from(m) := true;
        release(th_j)
   else release(th_j)
   fi
od;
claim(m);
busy(m) := false;
release(m)
```

Of course, in order to be able to execute such programs as above, all resources have of course to do

some internal work. However that internal work will not affect the communication.
Here we wanted to abstract from internal work and to consider just the communication between transaction handlers and machines. We briefly mention one of the aspects of internal work. As we have seen before, a transaction handler is able to receive a reaction while doing some (less important) work internally. We want to help machines getting free again as soon as possible.
Therefore we propose some buffer IN at th_j, where th_j can store some reaction, when it receives a reaction, and we then need to remember what th_j was doing just before it decided to receive the reaction, in order to be able to resume that work. So we will need some other buffer OUT in which is stored what to do next.
Here we do not pay further attention to this internal work.

6. Conclusions on the Communication

We can conclude that the communication directly resulting from the schedule is a rather small part of all the communication involved.
We can see that the main part of the programs described in this section is dealing with the exclusive access to several resources, that is claiming and releasing. Only a small part is acting on the variables of the schedule like UV_j and AU_j. Furthermore of this claiming and releasing the majority is aimed at waiting until the other, the receiver of the communication, is ready.

Later, in a next paper, we more formally define what transaction handlers and machines are. Then we see that, because of those definitions, the work involved in the communication between transaction handlers and machines can be less, mainly because the problem of guaranteeing mutual exclusion is tackled differently.

Until now we only considered one level of transaction handlers and one level of machines. We want to extend the model in defining systems to consist of transaction machines. A transaction machine will consist of a transaction handler and a machine. Information systems can then consist of transaction machines, which each are able to communicate with every other transaction machine. A transaction machine can, when handling a transaction, send transactions, which will be subtransactions of the transactions it is handling, to other transaction machines.
In that way we are approaching distributed databases, since transaction machines are in fact databases (machines), which have the possibility of handling transactions (transaction handlers) and thus of communicating with other databases.
Until here we did consider the partition of both information and work on a flat level. When we have the concept of transaction machines, we have the possibility of partitioning information and work in various other ways. That implies the usage of this model to describe many other systems.

References

Hansdah, R.C. and L.M. Patnaik, Update Serializability in Locking, to appear in the proceedings of ICDT, Rome,September 1986.
Houben, G.J. and J. Paredaens, A Formal Model for Distributed Information Systems, Eindhoven University of Technology, in preparation.
Schlageter, G., Process Synchronization in Database Systems, ACM TODS,Vol.3,No.3,September 1978,Pages 248-271.

COMPUTING SCIENCE NOTES

In this series appeared :

85/01	R.H. Mak	The Formal Specification and Derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a Computer for Evaluation of Flow Films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive Directed Trace Structures Satisfy the Foam Rubber Wrapper Postulate
86/01	R. Koymans	Specifying Message Passing and Real-Time Systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A Language for Formal Specifications of Information Systems
86/03	Rob Hoogerwoord	Some Reflections on the Implementation of Trace Structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The Partition of an Information System in Several Parallel Systems