Figure 30.2: A distributed computation
$P_i$:  
\textbf{type} entry = (integer ver, integer ts); // version, timestamp  
\textbf{var} clock : array [1..N] of entry initially  
\hspace{1em} \forall j : clock[j].ver = 0 ;  
\hspace{1em} \forall j : j \neq i : clock[j].ts = 0; clock[i].ts = 1 ;  

To send message :  
\hspace{1em} send \ (data, clock) ;  
\hspace{1em} \text{clock}[i].ts := \text{clock}[i].ts + 1 ;  

Upon receive of a message (data, m\text{clock}) :  
\hspace{1em} // \ P_i \text{ receives vector clock 'm\text{clock}' in incoming message }  
\hspace{1em} \forall j : \text{clock}[j] = \max(\text{clock}[j], m\text{clock}[j]) ;  
\hspace{1em} \text{clock}[i].ts := \text{clock}[i].ts + 1 ;  

Upon Restart (state s restored) :  
\hspace{1em} \text{clock} = s.\text{clock} ;  
\hspace{1em} \text{clock}[i].ver := \text{clock}[i].ver + 1 ;  
\hspace{1em} \text{clock}[i].ts = 0 ;  

Upon Rollback (state s restored) :  
\hspace{1em} \text{clock} = s.\text{clock} ;

Figure 30.3: Formal description of the fault-tolerant vector clock
$P_i$: 

Receive\_message (data, mclock) :

// Check whether message is obsolete
\forall j: \text{if } ((mclock[j].ver, t) \in vtable[j]) \text{ and } (t < mclock[j].ts) \text{ then}
discard message;

if \exists j, l \text{ s.t. } l < mclock[j].ver \land P_i \text{ has no token about } P_j \text{ then}
postpone the delivery of the message until that token arrives;

Restart (after failure):

restore last checkpoint;
replay all the logged messages that follow the restored state;
insert(vtable[i], (v, clock[i].ts));
broadcast\_token(clock[i]);

Receive\_token (v,t) from $P_j$:
synchronously log the token to the stable storage;
if ((mes, v, t') \in vtable[j]) then
  if (t < t') then Rollback;
// Regardless of rollback, following actions are taken
update vtable;
deliver messages that were held for this token;

Rollback (due to token (v,t) from $P_j$):
log all the unlogged messages to the stable storage;
restore the maximum checkpoint such that
either no record (v, t') \in vtable[j] or (t' < t) ..(l)
discard the checkpoints that follow;
replay the messages logged after this checkpoint
until condition (l) holds;
discard the logged messages that follow;

Figure 30.5: An optimistic protocol for asynchronous recovery
1. On a failure, a process loses information about the messages that it received but did not log before the failure. These messages are lost forever, unless $P_i$ also broadcasts its clock with the token and other processes resend all the messages that they sent to $P_i$ (only those messages need to be retransmitted whose send states were concurrent with the token’s state). This means that processes have to keep a send version end table. Observe that no retransmission of messages is required during rollback of a process that has not failed but has become orphan because of a failure of some other process. Before rolling back, it can log all the messages, and so no message is lost.

2. Some form of garbage collection is also required for reclaiming space. Before committing an output to the environment, a process must make sure that it will never roll back the current state or lose it in a failure.

### 30.5.1 An Example

In Figure 30.6, $c_i$ is the checkpoint of process $P_i$. The value of the FTVC and the version end table is also shown for some of the states. The FTVC is shown in a box. The row $i$ of the FTVC and the version end table corresponds to $P_i$. Some of the state transitions are not shown to avoid cluttering of the figure. The process $P_1$ fails in state $f_{10}$. It