

Elastic Vector Time

Christof FETZER* Michel RAYNAL†

* AT&T, 180 Park Ave, Florham Park, NJ 07932, USA

† IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

christof@research.att.com raynal@irisa.fr*

Abstract

In recent years there has been an increasing demand to build "soft" real-time applications on top of asynchronous distributed systems. Designing and implementing such applications is a non-trivial task and application designers are often faced with the need to circumvent impossibility results. In this paper we discuss how to ensure that actions are executed in the correct order even in the face of failures. We propose a novel time base and a new synchronization mechanism for the design of distributed "soft" real-time applications. We demonstrate (1) how this time base can be used to enforce an externally consistent ordering, and (2) how it permits to circumvent impossibility results by sketching how to solve the leader election and perfect failure detection problem.

Keywords: *Elastic vector time, external consistency, logical time, leader election, failure detection.*

1 Introduction

Problem Description Most distributed systems are asynchronous in the sense that there exists no upper bound on the transmission delay of messages and there exists no guaranteed maximum scheduling delay for actions. Designing and implementing distributed applications for asynchronous systems is a difficult task. First, many application designs require solutions to basic problems like the leader election and the perfect failure detection problem – which are impossible to solve in purely asynchronous system models. In contrast, most existing distributed systems seem to have sufficient synchrony to solve these basic problems. Second, many distributed applications control an external system that require that actions – as seen by a global observer – are taken in a certain order. Most group communication systems for asynchronous systems only permit to enforce an

order consistent for internal observers. Third, the utility of "soft" real-time applications depends on how close actions are executed to their scheduled time. However, the safety of a "soft" real-time application is by definition not violated due to the late execution of scheduled actions. (Note that the order between actions might need to be preserved in the face of late executions or crashed processes).

To simplify the design and implementation of such distributed applications, we need abstractions that 1) provide a proper amount of synchrony (i.e., they permit to solve the same set of problems as in existing distributed systems), 2) simplify maintaining an externally consistent order, and 3) permit to execute an action at a certain time.

Content of the paper This paper proposes a new time base (that we call *elastic vector time*) and an associated synchronization mechanism (*wait-delivered*) to help simplifying the task of designing and implementing distributed applications. We define the properties of elastic vector time such that elastic vector clocks can be used in a very similar way as real-time clocks in synchronous systems, e.g., one can use them for failure detection and communication by time. We show how elastic vector clocks can be used to 1) solve the perfect failure detection and the leader election problem, 2) ensure an externally consistent order, and 3) schedule an action at a certain time. Elastic vector time is kept as close as possible in sync with real-time. While most of our focus is on "soft" real-time systems, the order preserving properties of elastic vector time has also benefits for distributed, non real-time applications.

The elastic vector time combines real-time with logical time [9]. The intuition is that as long as the underlying system is sufficiently fast, elastic vector time is kept in sync with real-time. During periods in which the underlying system is too slow, elastic vector time is slowed down too. A synchronization mechanism that permits application programmers to take advantage of elastic vector time is introduced. More precisely, applications can be programmed similar to those in synchronous systems. It is also shown how one can use elastic vector time to implement leader election and a perfect failure detector.

*This paper will appear in the Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS2003).

Organization of the paper Section 2 introduces the concept of elastic vector time. We sketch how to implement elastic vector time in timed asynchronous systems in Section 3. Section 4 shows how to implement leader election and a perfect failure detector with elastic vector time. Section 5 discusses related work. Finally, Section 6 conclude the paper.

2 Elastic Vector Time

Distributed real-time applications need to schedule actions to be executed at certain times. Scheduling actions imposes a partial order on all actions. In [6] we introduced a scalar time base that enforces the execution of actions to be “internally consistent” with this partial order, i.e., an internal observer cannot prove any violation of the partial order. In this paper, we introduce the elastic vector time base that permits the execution of actions to be “externally consistent” with this partial order, i.e., an external observer cannot prove any violation of the partial order.

We assume that processes can fail by crashing. Crashed processes do not recover. Messages are delivered via reliable channels.

2.1 External vs Internal Consistency

Elastic vector time provides an algorithm with the functionality to specify during run-time that an action A has to be executed at some future time T . Let $ST(A)$ denote the *scheduled time* T of an action A , i.e., in this case $ST(A) = T$. If an action A is not scheduled, we assume that $ST(A)$ is undefined, i.e., $ST(A) \uparrow$. We call an action A for which $ST(A)$ is defined, a *scheduled action*.

Scheduling actions introduces a partial order $<_S$ that constrains the order in which actions are executed. Naively, we could define $<_S$ as follows: for any two scheduled actions A_1 and A_2 : (N) $A_1 <_S A_2$ iff $ST(A_1) < ST(A_2)$. Such a definition would introduce an overhead for each action. Instead we permit a programmer to define a partial order between actions. We give a definition of $<_S$ in Section 2.6 that permits to relax the order between actions while still providing the possibility to define the same “strong” ordering between actions as (N).

The goal is that the system executes scheduled actions in an order that is consistent with the partial order $<_S$. We say that an execution is *internally consistent* iff the following condition holds: if an action A_1 is scheduled to be executed before an action A_2 (i.e., $A_1 <_S A_2$), then A_2 does not happen before A_1 . We use notation \rightarrow_H to refer to the *happened before* relation [9]. Formally, we express this condition (IC) as follows:

$$(IC) \forall A_1, A_2 : A_1 <_S A_2 \Rightarrow A_2 \not\rightarrow_H A_1.$$

Informally, internal consistency means that if A_1 is before A_2 , then one cannot find a sequence of successive actions that proves that A_2 happened before A_1 .

Let $e(A)$ denote the point in real-time when action A is executed. We say that an execution is *externally consistent* iff the following condition holds: if an action A_1 is scheduled to be executed on process p before an action A_2 on process q (i.e., $A_1 <_S A_2$), then A_1 is executed before A_2 unless not both actions are executed due to a crash of p or q . More precisely, if A_1 is not executed because p crashes, we require that A_2 is executed after p has crashed. We define that predicate $crashed_p^t$ is true iff process p is crashed at real-time t . Formally, we express this condition (EC) as follows:

$$(EC) \forall A_1, A_2 : A_1 <_S A_2 \Rightarrow [(e(A_1) < e(A_2)) \vee (e(A_2) < \infty \Rightarrow crashed_p^{e(A_2)})].$$

Informally, external consistency means that if A_1 is scheduled before A_2 then A_2 cannot be executed before A_1 . External consistency implies internal consistency. This can be shown using the fact that the transmission delay of messages and the execution of actions takes a positive amount of time.

Note that internal consistency does not imply external consistency. If A_1 and A_2 are concurrent with respect to the happened before relation, i.e., $A_2 \not\rightarrow_H A_1$ and $A_1 \not\rightarrow_H A_2$, then the system is permitted to execute A_2 before A_1 (i.e., $e(A_2) < e(A_1)$) even if A_1 is scheduled before A_2 (i.e., $A_1 <_S A_2$).

2.2 An Application Example

Internal consistency is often sufficient to guarantee properties defined over the global state of a distributed system. However, if a distributed system controls or interacts with the external world, one might need to enforce external consistency. To explain this, let us consider the following small application example.

Our application consists of a water tank with 3 valves (see Figure 1): an inflow, outflow, and overflow valve. Two valves (outflow and overflow) are controlled by a process p running on one computer, and the third valve is controlled by a process q running on a different computer. To ensure the safety of the system, the overflow valve must be open whenever the inflow valve is open. To optimize the utility of the system, the inflow valve has to be open in a given time interval $[T, U]$.

In a completely synchronous system without crash failures and with a sparse time base, i.e., all actions are executed at their scheduled time [8], one can solve this problem as depicted in figure 2. Terms R_p and R_q denote the synchronized real-time clocks of processes p and q , respectively. The timing of the program is depicted in Figure 3.

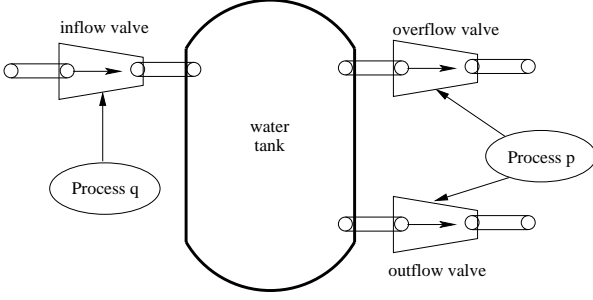


Figure 1. Controlled system with one safety property (the overflow valve has to be open whenever the inflow valve is open)

```

process p:
Action A1: when  $R_p = T - 2$  do closeOutflowValve(); enddo
Action A2: when  $R_p = T - 1$  do openOverflowValve(); enddo

process q:
Action A3: when  $R_q = T$  do openInflowValve(); enddo
Action A4: when  $R_q = U$  do closeInflowValve(); enddo

```

Figure 2. Application Program: Sketch

As long as all actions are executed in their scheduled order (see Figure 2), the safety of the system is ensured. That means that as long as external consistency (EC) is guaranteed, the system is safe. Note that internal consistency is not sufficient to ensure the safety of the system because the actions on process p and q are concurrent with respect to the happened before relation. Also, the closer the actions are executed to their scheduled time, the higher is the utility of the system.

2.3 Definition

The goal of elastic vector time is as follows: 1) permit the enforcement of external consistency, and 2) attempt to keep elastic vector time in sync with real-time whenever possible.

To define elastic vector time, we assume that each process p has access to a lower level clock R_p . The exact properties of this clock depends on the underlying system model. The goal is that R_p is (as close as possible) synchronized with real-time. The maximum external deviation and drift of the clocks is not needed in the definition of elastic time. However, a better synchronization will increase the utility of the system (as defined by the cost function defined below). We assume that R_p is strictly monotonic and unbounded.

Elastic vector time requires that each process p has a local clock V_p . The value of clock V_p is a vector that contains an entry for each process. We denote the value of V_p at

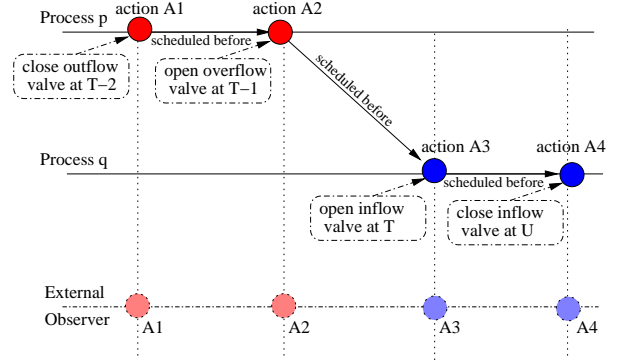


Figure 3. In a sparse time base, an external observer sees the state changes in the order they are scheduled.

real-time t by V_p^t and the entry of process q by $V_p^t(q)$. Entry $V_p^t(p)$ is the *local time* of process p . The intuition is that $V_p^t(p) = R_p^t$ whenever possible. However, the clock can be slowed down, i.e., $V_p^t(p) < R_p^t$, if the process cannot keep up processing events.

We assume that the execution of an action A is atomic in the sense that there exists exactly one point in real-time t at which A is executed. Recall that the execution time with respect to real-time is denoted by $e(A)$. The execution time with respect to elastic vector time is denoted by $E(A)$, i.e., $E(A) = V_p^{e(A)}(p)$. We denote the set of all actions in an execution by H .

Elastic vector time is defined by five properties (synchronized, scheduling, estimate, unbounded, and logical time consistency). The performance of an elastic vector time implementation is determined by a cost function CF .

To simplify the definitions of the properties, we assume that all clocks, scheduling times, execution times, and variables of a crashed process have by definition value ∞ , e.g., when a process crashes all its scheduled actions are rescheduled for time ∞ . In this way all properties are by definition true for crashed processes.

We want to avoid that elastic vector time can be slowed down forever. For example, if a process could send an unlimited number of messages per clock tick, the process could slow down time forever. We therefore need to ensure that the work per time unit is bounded. Hence, we restrict the number of actions a process can schedule per time unit and the number of message it sends per time unit. For simplicity, let us assume a process can send at most one message per time unit and the distance between any two scheduled actions is at least one time unit.

The elastic vector clock V_p of a process p must not be faster than its local clock R_p :

(synchronized) $\forall p, t : V_p^t(p) \leq R_p^t.$

Each scheduled action is executed at its scheduled time:

$$\text{(scheduling)} \quad \forall A : E(A) = ST(A).$$

A vector clock provides a process p with an estimate of the other clocks. This estimate can be lower than the actual value but must never be greater than the actual clock value unless p is crashed (when the estimate is by definition ∞).

$$\text{(estimate)} \quad \forall p, q, t : \neg \text{crashed}_p^t \Rightarrow V_p^t(q) \leq V_q^t(q).$$

The values of each elastic vector clock are unbounded:

$$\text{(unbounded)} \quad \forall p, q, s, \exists t : V_p^t(q) \geq s.$$

Elastic vector clocks are logical clocks [9]:

(logical time consistency)

$$\forall p, A_1, A_2 : A_1 \rightarrow_H A_2 \Rightarrow E(A_1) < E(A_2).$$

The logical time consistency property implies that an elastic vector clock is strictly monotonic with respect to the local times at which an action is executed. To make sure that elastic vector clocks are monotonic with respect to all real-time instances, we require the following:

$$\text{(monotonicity)} \quad \forall p, q, s, t : s \leq t \Rightarrow V_p^s(q) \leq V_p^t(q).$$

Recall that we assume that crashed processes do not recover. Permitting processes to recover would violate the monotonicity requirement since we assume that the clock value of a crashed process is ∞ .

An implementation of elastic vector time is supposed to keep elastic vector time as closely as possible to the time base represented by R_p^t . To express this goal formally, we define a cost function CF . The goal of an implementation of elastic vector time is to minimize the cost function CF :

$$CF := \sum_{A \in H} (e(A) - ST(A)).$$

Note that the cost function defines a partial order on implementations of elastic vector time. It permits us to distinguish between better and worse implementations.

2.4 Synchronization

We do not require that in an elastic vector time base all actions are executed externally consistent because this could be too costly. Instead, we introduce two primitives (**wait** and **wait-delivered**; see Section 2.5) that give an algorithm designer a finer control of which actions need to be executed externally consistent.

The **wait** primitive schedules the execution of an action A on a process p such that each remote estimate in p 's elastic vector clock V_p shows a value greater than the corresponding entry given in a *threshold vector* W (see Figure 4). The scheduling time of action A is $W(p)$, i.e., $ST(A) =$

$W(p)$. Note that the action has to occur *at* $W(p)$ and *after* every other $W(q)$. The execution of an action satisfies the following property (see also Figure 5):

$$\text{(delay)} \quad \forall q \neq p : W(q) < V_p^{e(A)}(q).$$

process p :

timeVector $W=(...);$

Action: **wait** $V_p > W$ **do** STATEMENTS; **enddo**

Figure 4. The wait primitive ensures that $V_p > W$ before executing the action. Note that $V_p(p)$ does not advance to $W(p)$ until the action is executed.

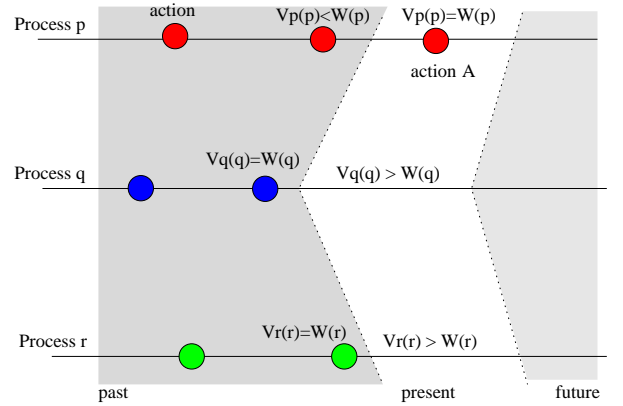


Figure 5. The threshold W of an action A defines which actions must be executed before A is executed at time $W(p)$.

The **scheduling** property together with the **monotonicity** property make sure that p 's local time (i.e., $V_p(p)$) cannot advance beyond $W(p)$ until A is executed. Therefore, p 's local time might have to be slowed down if the other processes are lagging too far behind, i.e., have not yet advanced beyond the time specified in the threshold vector.

We have to constrain the threshold vector of an action to avoid deadlocks. For example, a deadlock would occur if two actions can wait for each other to be executed first. To avoid deadlocks, we require that an action to be executed on a process p can only be waiting for actions that are scheduled beforehand, i.e.:

$$\text{(no-circularity)} \quad \forall q : p \neq q \Rightarrow W(q) < W(p).$$

2.5 Wait-Deliver Primitive

As the **wait** primitive, the **wait-delivered** primitive satisfies the **(delay)** property. In addition, it defines a *send-time*

vector X which enforces that a process p delivers certain messages before executing some action A : each message sent by a process q to p no later than time $X[q]$ has to be delivered before A is executed (see Figure 6).

Like the threshold vector, we also have to constrain X to avoid deadlocks and livelocks. We assume that the system defines a constant $\Delta > 0$: a message m has to be in transit for at least Δ time units before the system slows down time to wait for m . A livelock (i.e., a violation of property **(unbounded)**) could occur if processes could send an infinite number of messages to a process p that all have to be delivered before the execution of some action A at time T . Consider that we would permit to specify that all messages sent by p before T have to be delivered before A is executed at time T . Then, an action B on p that receives a message m_i before T and sends another message m_{i+1} to p will result in a livelock since m_{i+1} has to be delivered before T and sends another message m_{i+2} and so on. By requiring that messages must be in transit for at least Δ before an action can wait for them, avoids this scenario.

| |
|--|
| process p: timeVector $W=(...), X=(...);$ Action: wait $V_p > W$ delivered X do STATEMENTS; enddo |
|--|

Figure 6. The wait-delivered primitive ensures that $V_p > W$ and that all messages sent by a process q to p no later than $X(q)$ have been delivered to q

We denote the local send-time of a message m by $XT(m)$, the local delivery time of m by $DT(m)$, the sender of m by $S(m)$, and the destination process by $D(m)$. For example, if action A_1 on process p sends a message m to process q and m is delivered to q in an event A_2 , then $XT(m) = E(A_1)$, $DT(m) = E(A_2)$, $S(m) = p$, and $D(m) = q$. If a message m is never delivered, then $DT(m) \uparrow$ is true, otherwise $DT(m) \downarrow$ is true.

Consider that process p executes action A and that A is guarded by a threshold vector W and send-time vector X . Action A is executed at $ST(A) = W(p)$, all clocks except $V_p(p)$ moved beyond the threshold, i.e., $\forall q \neq p : W(q) < V_p(q) \leq V_q(q)$ (**delay** property). No message sent before or at X to p is delivered at or later than $V_p(p)$. Only if process q has crashed by the time p executes A , p is not required to be delivering m . Formally, we express this as follows:

| |
|--|
| (stability) $\forall p, q, m : S(m)=q \wedge D(m)=p \wedge XT(m) \leq X(q) \Rightarrow [DT(m) \downarrow \Rightarrow DT(m) < W(p)] \wedge [DT(m) \uparrow \Rightarrow crashed_q^{e(A)}]$ |
|--|

| |
|--|
| (minimum transmission time) $\forall q : X(q) + \Delta \leq W(p).$ |
|--|

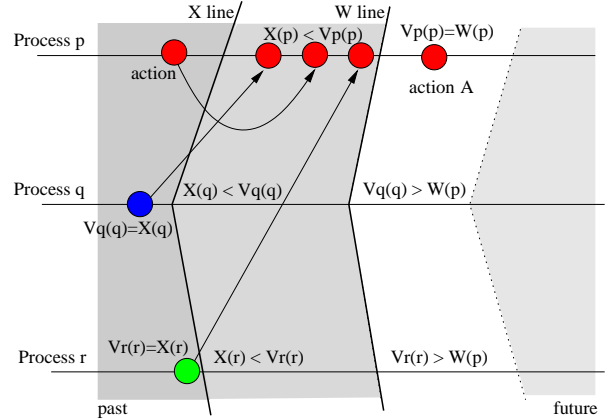


Figure 7. The send-time vector X of action A delays the execution of A until no message sent to p can cross both the X and W lines.

We assume that time starts at time 1 and that all clocks have values greater than or equal to 1, i.e., $\forall p, q, t : R_p^t \geq 1 \wedge V_p^t(q) \geq 1$. Scheduling an action without waiting for other processes to reach a certain time can therefore be done by setting the entries of the threshold vector to 0. The **wait** primitive can be viewed as syntactic sugar: it is the **wait-delivered** primitive with the send-time vector set to the null vector.

2.6 External Consistency

We show that elastic vector time enforces external consistency (see Section 2.1). The **wait-delivered** primitive together with the happened before relation introduces a relation on actions which we refer to by $<_S$. For any two actions such the $A_1 <_S A_2$ and both actions are executed, we show that A_1 is executed before A_2 . If A_2 is executed but A_1 is not executed, we have to show that A_2 is executed after the process p that was scheduled to execute A_1 has crashed.

Let A_1 and A_2 be two scheduled actions. We define that $A_1 <_S A_2$ iff one of the following conditions holds:

- (1) action A_1 happened before A_2 , i.e., $A_1 \rightarrow_H A_2$,
- (2) action A_2 has a threshold of W and action A_1 is scheduled on another process q at $ST(A_1)$ such that $ST(A_1) \leq W(q)$, or
- (3) there exists an action A_3 such that $A_1 <_S A_3 <_S A_2$.

With this definition of $<_S$, we can show that external consistency (see Section 2.1) is enforced.

Theorem 1: *Elastic vector time guarantees external consistency.*

Proof: A) We have to show that if $A_1 <_S A_2$ and both A_1 and A_2 are executed, then A_1 is executed before A_2 . First, consider that action A_1 happened before A_2 , i.e., $A_1 \rightarrow_H A_2$. We show this case by contradiction: assume $e(A_2) < e(A_1)$. Because $A_1 \rightarrow_H A_2$, we can find a sequence of actions $B_1 = A_1, B_2, \dots, B_N = A_2$ such that B_i precedes B_{i+1} on the same process or B_i sends a message that is received by B_{i+1} . This means that $e(B_i) < e(B_{i+1})$ for all $1 \leq i \leq N - 1$ and hence, $e(A_1) = e(B_1) < e(B_N) = e(A_2)$. This is a contradiction to the assumption that $e(A_2) < e(A_1)$.

Second, let us consider the case that A_1 is executed on process p and A_2 on a process $q \neq p$ with a threshold W such that $ST(A_1) \leq W(p)$. To derive a contradiction, let us assume that $e(A_2) \leq e(A_1)$. At $e(A_2)$ the **delay** together with the **estimate** property ensure that $W(p) < V_q^{e(A_2)}(p) \leq V_p^{e(A_2)}(p)$. The **scheduling** property together with the precondition for this case, imply that $V_p^{e(A_1)}(p) = ST(A_1) \leq W(p)$. Therefore, $V_p^{e(A_1)}(p) < V_p^{e(A_2)}(p)$ which is a contradiction to the **monotonicity** property because we assumed that $e(A_2) \leq e(A_1)$.

Third, consider that $A_1 <_S A_2$ and there exists an A_3 such that $A_1 <_S A_3 <_S A_2$. Assume we know that $e(A_1) < e(A_3)$ and $e(A_3) < e(A_2)$, then we also know that $e(A_1) < e(A_2)$.

B) If $A_1 <_S A_2$ but only A_2 is executed, we have to show that at the time A_2 is executed, the process p that was scheduled to execute A_1 at $ST(A_1)$ has crashed at time $e(A_2)$. As above in the second case, we can assume that $ST(A_1) \leq W(p) < V_p^{e(A_2)}(p)$ by the definition of $<_S$ and by the delay requirement. Since A_1 is not executed, we know from the scheduling property that p cannot have reached $ST(A_1)$ before it crashed – otherwise A_1 would have been executed at $ST(A_1)$. Hence, for $W(p) < V_p^{e(A_2)}(p)$ to be true, $V_p^{e(A_2)}(p)$ has to be ∞ . This means that p is crashed at time $e(A_2)$. \square

2.7 An Application Example (Continued)

We are now showing how to solve the water tank application described in Section 2.2 using elastic vector time. To ensure the safety of the water tank application, we ensure the following execution order for actions A_1, A_2, A_3 , and A_4 : $e(A_1) < e(A_2) < e(A_3) < e(A_4)$. If we can exclude crash failures, this can be ensured by making sure that $A_1 <_S A_2 <_S A_3 <_S A_4$ (see Theorem 1 in Section 2.6). To do this, we require that action A_{i+1} be waiting for A_i to be executed first (see Figure 8).

3 Implementation In Timed Asynchronous Systems

We sketch how to implement elastic vector time in timed asynchronous systems [4] with hardware watchdogs and an externally synchronized clocks (e.g., see [3]). The hardware watchdogs permit the implementation of a perfect fail-

```

process p:
Action A1: wait  $V_p > (T - 2, 0)$  do closeOutflowValve(); enddo
Action A2: wait  $V_p > (T - 1, 0)$  do openOverflowValve(); enddo

process q:
Action A3: wait  $V_q > (T - 1, T)$  do openInflowValve(); enddo
Action A4: wait  $V_q > (0, U)$  do closeInflowValve(); enddo

```

Figure 8. Application program based on elastic vector time

ure detector [5], i.e., all crashed processes are eventually suspected and no non-crashed process is suspected. Our implementation is based on such a perfect failure detector.

Actions are locally executed in their scheduled order. Before an action A is executed, the executing process p makes sure that 1) p has delivered all messages matched by the send-time vector X of A , and b) that its approximation of the remote clocks has advanced beyond the threshold vector W of A . To ensure this, all messages are sent via reliable FIFO channels and each message is appended by the local time of the sender. In case the two conditions are not satisfied, process p polls each process q of which p does not know that p has delivered all messages from q that must be delivered before A or from which p 's approximation of q 's clock has not sufficiently advanced yet.

A remote process replies to a poll as soon as the condition specified in the poll are satisfied. To avoid the situation that a process p waits forever for a reply from a crash process q , we use a perfect failure detector. As soon as a process q is detected to have crashed, all wait and deliver conditions for q are satisfied. A message that arrives from q after p has detected q 's crash is only delivered if it does not violate the stability requirement.

4 Examples

4.1 Leader Election

The leader election problem states that at any point in real-time there is at most one leader and infinitely often there is a leader (unless all processes crash). In a completely synchronous system, an easy way to solve the leader election problem is to define infinitely many non-overlapping time slots and assign statically each time-slot a unique leader in a round-robin fashion. The non-overlapping of the time-slots and having one agreed upon process per time-slot ensures that at any time there is at most one leader. Assigning each process infinitely many time-slots ensures that infinitely often there is a leader unless all processes crash.

Elastic vector time permits to implement this algorithm very much the same way as in a synchronous system (see Figure 9). Each process p schedules an action (A_i) at the

start of its next time-slot and the end of its next time-slot (A_2). At the start of p 's time-slot it waits for all other processes to have reached the end of the time-slot of the previous leader before it enters its own time-slot by setting a flag (`leader`) to true indicating it is the new leader. If needed, p could additionally wait to get all messages from the previous leader. At the end of its time-slot, p waits to make sure that all processes are close to the end of the time-slot (e.g., this permits the process to continue to reply to requests it might get from slower processes) and then it resets its leader flag.

Because the new leader waits for all previous leaders to first leave their previous time-slots, Theorem 1 implies that the time-slots of the processes are non-overlapping. Note that for this leader election algorithm, elastic vector time is sufficiently strong such that the algorithm does not even have to detect that or which processes have crashed.

```

const const time P = ... ; % length of time-slot
const const int N = ... ; % number of processes
Process p: bool leader = false; % true iff process is leader
const int myrank = ... ; % rank of process  $\in \{1, \dots, N\}$ 
timeVector F = <P*myrank-1>, F[myrank]=P*myrank;
timeVector U = <P*(myrank+1)-2>, U[myrank]=P*(myrank+1)-1;

Action A1: wait  $V_p > F$  do
  leader  $\leftarrow$  true; F  $\leftarrow$  F + <P*N> enddo

Action A2: wait  $V_p > U$  do
  leader  $\leftarrow$  false; U  $\leftarrow$  U + <P*N> enddo

```

Figure 9. Time-Slotted Leader Election

4.2 Perfect Failure Detector

A perfect failure detector [2] ensures that eventually each crashed process is permanently suspected by any non-crashed process and that if a process p is suspected to have crashed, p is indeed crashed. Implementing a perfect failure detector in a completely synchronous system is straightforward. A process p sends periodically (say every π) an 'alive' message and if a process q has not received an alive message within some δ of the scheduled send time, it knows that p has crashed (because processes and messages are timely).

This simple perfect failure detector protocol can be implemented in a very similar way using elastic vector time (see Figure 10). Process p schedules the transmission of an alive message to process q every π time units. To make sure that p does not advance to far ahead of q , it waits that q has received its last alive message before it sends a new one. In other words, we use the threshold vector to enforce some basic flow control.

Process q schedules an action A_2 that is scheduled to be executed after the next alive message of p is delivered ($X(p)$

```

const time  $\pi = \dots, \delta = \dots$ ;
process p:
  timeVector W = < 0 >, W[p] =  $\pi$ ;
Action A1: wait  $V_p > W$  do
  send ("alive",  $V_p[p]$ ),  $q$ ;
  W[q] = W[p] +  $\delta + 1$ ; W[p] = W[p] +  $\pi$ ;
enddo
process q:
  boolean crashed_p = false;
  time AT = 0;
  timeVector T = < 0 >, T[q] =  $\pi + \delta + 1$ ;
  timeVector X = < 0 >, X[p] =  $\pi$ ;
Action A2: wait  $V_q > T$  delivered X do
  if (AT < X[p]) then
    crashed_p = true;
    T[q] =  $\infty$ ;
  else
    T[q] = T[q] +  $\pi$ ; X[p] = X[p] +  $\pi$ ;
  endif
enddo
Action A2: deliver ("alive", T) from q do
  if (T > AT) AT = T; endif
enddo;

```

Figure 10. Perfect failure detector: process q can detect the crash failure of process q .

contains the send time of the next alive message that q expects). The **stability** property states that the alive messages has to be delivered unless p has crashed by the time A_2 is executed. This means q can safely assume that if it has not gotten the last expected alive message when executing A_2 that p has crashed.

Consider that process p crashes and hence, does not send its next alive message scheduled for local time x . Action A_2 of process q waits for this message to be delivered before time $x + \delta + 1$. Property **unbounded** states that $V_q(q)$ is unbounded, i.e., eventually has to go beyond $x + \delta + 1$. Hence, q has to executed action A_2 eventually. This action will detect that the alive message was not delivered and hence, suspects process p .

5 Related Work

Verissimo and Casimiro introduced the time-elastic application class [11]. Informally, that means that all actions of an application permit some change of their scheduling time while still ensuring the application properties. In our approach we permit the time to be slowed down to ensure that all actions are executed at their scheduled time instead of changing the scheduled time. This is sufficient for the applications which can guarantee their properties based on the external consistency (EC) property.

We introduced a scalar version of elastic time in [6]. The emphasis of the scalar version was internal consistency (IC) while the vector version together with the new *wait-*

delivered primitive permits to enforce external consistency (EC). Elastic vector time is more powerful, e.g., it permits to implement a Perfect Failure Detector while the scalar version does not. In [6] we introduced a “suspension” mechanism that suspends the execution of an action if it were to be executed outside of a specified real-time interval. In some applications (e.g., leader election) this can be used to enforce external consistency because the application permits the system to omit the execution of certain actions. In [6] we also introduced an adaptation mechanism that permits to rescheduled actions in case slowing down time increased the difference to real-time beyond some threshold. Adding the suspension and adaptation mechanisms to the elastic vector time base could be advantageous for certain applications.

Birman’s virtual synchrony [1] is related to elastic vector time in the sense that it is an approach to provide synchronous system guarantees in an asynchronous system. The emphasis of virtual synchrony is on multicast communication and internal consistency. The focus of elastic vector time is instead on unicast communication and external consistency. Virtual synchrony provides the processes with a *view* of processes that are currently considered to be up. All processes installing a new view agree on the messages delivered in the previous view. The corresponding mechanism in elastic vector time is the explicit specification of a send-time vector by the application. That means that synchronization only happens when the application requests it and with the set of processes requested by the application.

Lamport introduced logical time [9]. Extensions of logical time to logical vector time were proposed independently by Fidge [7] and Mattern [10]. Logical vector time LV tracks exactly the happened before relation, namely, $A_1 \rightarrow_H A_2 \Leftrightarrow LV(A_1) < LV(A_2)$. We decided not to make this property part of elastic vector time and instead making elastic vector time only consistent with (scalar) logical time (see **logical time consistency** property). Our reasoning is that elastic vector time permits processes to “communicate by time” (e.g., a process p learns that process q ’s lease has expired when p learns that q has reached some time T). Hence, in our system, being able to track the happened before perfectly does not mean tracking causality perfectly.

6 Conclusion

We introduced elastic vector time. This new time notion can be viewed as a combination of logical time and real-time. This new time base permits to schedule the execution of actions at points in real-time and to define an external order on actions. Elastic vector time is slowed down if the underlying system is not sufficiently fast to execute the actions at their scheduled time. However, even if time is slowed down, the global order of actions will be enforced.

Elastic vector time has several noteworthy properties: (1) it is powerful, e.g., it permits to implement perfect failure detectors, (2) it provides processes with a notion of time, e.g., one can implement a leases mechanism, (3) it can help to simplify the design of distributed algorithm by restricting the number of possible global states, (4) it increases the portability in the sense that it slows down automatically for slow systems, (5) it can enforce external consistency, which might be needed in systems interacting with the external world. Elastic vector time can be extended by an adaptation mechanism in a similar way as described in [6] to permit the rescheduling of actions in case the deviation between elastic vector time and real-time becomes too large.

References

- [1] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37-53, 1993.
- [2] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, 1996.
- [3] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146-158, 1989.
- [4] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, 1999.
- [5] C. Fetzer. Enforcing perfect failure detection. *IEEE Transactions of Computers*, to appear.
- [6] C. Fetzer and M. Raynal. Approximate real-time clocks for scheduled events. *Proc. 5th IEEE Int. Symposium on Object-oriented Real-time distributed Computing (ISORC’02)*, IEEE Computer Press, pp.54-61, Washington DC, 2002.
- [7] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computer Science Conf.*, 1988.
- [8] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proc. 12th Int. IEEE Conference on Distributed Computing Systems*, IEEE Computer Press, pp.460-467, 1992.
- [9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558-565, 1978.
- [10] F. Mattern. Time and global states in distributed system. In *Proc. Int. Workshop on Parallel and Distributed Algorithms*, North-Holland, pp.215-226, 1989.

- [11] P. Verissimo and A. Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):881-882, 2002.