INFORMATICA, 1996, Vol. 7, No. 3, 371–388

STABILIZING PROTOCOL FOR A NETWORK OF PROCESSORS

Leo SINTONEN

Tampere University of Technology, Software Systems Laboratory
POB 553, 33101 Tampere, Finland
E-mail: lsi@cs.tut.fi

Abstract. It is desirable in many applications, that a set of cooperating processes do not loose their coordination due to failures. This paper presents a fault-tolerant protocol for a network of processors, which form a logical ring on a physical broadcast medium. The presented protocol makes possible for a set of processes to reestablish their normal operation after transient or permanent process failure or transient communication failures. The protocol is described, model for the system is developed, and in the framework of this model it is proved, that the system reaches a stable and correct configuration in finitely many steps after failure.

Key words: cooperating processes, fault-tolerant computing, distributed algorithms, self-stabilization, token bus protocol, event driven protocol.

1. Introduction. The increasing use of distributed computing in safety-critical applications has made fault-tolerance a desirable feature. Distributed systems that tolerate up to k failing processes and still perform correctly are called *k-resilient* (Rangarajan *et al.*, 1995). A system of processor is self-stabilizing, when started from an illegal initial state is guaranteed to converge to a legal state in finite number of steps (Dijkstra, 1974). In this paper we considered applications, in which a set of processors cooperate by communicating and can loose the coordination due to a processor failure or a communication failure. Class of applications considered in this context include, e.g., applications of mobile Local Area Networks in automation, when communication is done through some kind of shared broadcast media like radiowaves.

The problem of failing processors is discussed broadly in distributed systems textbooks. The problem is usually treated in terms of reliability and availability obtainable through replication and resilient protocols (Rangarajan *et al.*, 1995).

In this paper we do not define fault tolerance in terms of reliability. Instead we define fault tolerant protocol as an algorithm having stabilizability properties.

The concept of self-stabilization was introduced by Dijkstra (1974). He considered a set of $n+1$ state machines connected to a ring. An important issue in the verification of such systems is the number of states of each processor. In (Burns and Pachl, 1989) the authors show, that there is a self-stabilizing system with no distinguished processor, if the size of the ring is prime. Their protocol uses $O(n^2)$ states at each processor, where n is the number of processors. Rings have also been studied in (Flatebo and Datta, 1994). The authors represent a self-stabilizing algorithm for mutual exclusion, which requires two states in each machine in the network.

Application of self-stabilization to fault-tolerance has been studied in Arora and Gouda (1993), Laranjeira *et al.* (1993). These generalize the notion of fault-tolerance to variety of state systems. The desired properties of the system are described by invariant predicates identifying the legal states related to fault-tolerance. Verification of the system is then based on the notion of convergence towards legal state set. Stabilizability and discrete event dynamic systems are discussed in Sobh (1991). The automaton model is used to develope techniques to describe the behaviour of such systems. Stabilizability is defined as the property of system to return to a "good" state by control of events.

Communication and communication protocols are an important part of distributed computation, and contribute also essentially to the fault-tolerance. Self-stabilizing communication protocols have been studied (Gouda and Multari, 1991). In Gouda and Multari (1991) there are two processes involved in communication. Authors represent a formal method for this case.

The protocol presented in this paper can be regarded as an access protocol with safety features. The protocol was originally designed for applications, where reliability and fault-tolerance are required (Sintonen, 1990). The access method is event-driven multiple access. Communication is through shared medium, but the order of using the medium is ring. Events are used to control the operation sequence of the systems. Processors cooperate by observing event sequence and can deduce station failures. Based on the observation processors, called stations here, can modify their view of the configuration. Transient errors in communication lead the system into a state, from which the normal operation can be restored. Thus, the part of the algorithm implementing reliability fea-

tures can be regarded as a separate layer. This is a new approach to reliability and it allows to combine resiliency and self-stabilation.

Model is represented to prove the reliability properties of the protocol. The model is based on the abstraction of *global events*, which cause *transitions*. Transitions a local and are locally guarded. Transitions cause processors to modify their view of the configuration. This allows us to describe the coordination amog distributed processors.

This paper is organized as follows. In Section 1 general issues related to this work are discussed. The protocol is described in Section 2. In Section 3 the computational model for the system is developed. In Section 4 the static station failure properties are proved. Effect of communication and timing errors is discussed in Section 5. Concluding remarks are made in Section 6.

2. Description of the protocol

A. Topology. The topology considered in this paper is a bus. Stations on the bus form a logical ring according some list of addresses. The addresses are numbers $0, \ldots, N-1$. There is a station which controls the operation of the bus. The address of this station is initially 0. It can be any station, but there is only one control station at a time.

Every station maintains a *list* of the actual configuration. Stations are also assumed to know their initial configuration. The address of the control station is in local variable ctrl_station.

A station has registers to hold the following parameters: PREDECESSOR register contains PREVIOUS_ADDRESS, which is the address of the predecessor station on the logical ring. The DESTINATION_ADDRESS and SOURCE_ADDRESS registers hold the destination and source addresses of the last message on the bus. Station also has register holding it's own address value. How the registers are implemented is not relevant here.

Stations communicate by sending a frame to a shared medium. Frame contains the address of the receiver station, the address of the sender station, data and possibly other relevant fields.

Stations have circuit to recognize the events FRAME_ENDED, circuits to detect colllision and to generate the event COLLISION_ENDED, and to detect the start of a sending on the bus. They also have counter to give time marks D and T, explained later.

B. Protocol. The protocol is based on the observable events on the bus.

Every station listens the bus and receives both the destination address and the source address and stores them in the registers respectively.

Receiving. When a station notices it's own address in the DESTINATION_ADDRESS register, it receives the frame.

Sending. When a station has a frame to send, it waits until it receives the address of it's predecessor in the logical ring into the SOURCE_ADDRESS register.

Then it waits for the event FRAME_ENDED of it's predecessor station. After that it sends it's frame. After the end of the frame it waits time for a time delay D to hear the next station to begin sending (event S). When this happens, the sending phase is ended. Delay D is equal or greater than $2 \cdot \tau$, where τ is the round trip delay.

The event starting the delay counter D is the frame ended-event FE. Counter D is cleared by the begun-sending-event S when next station begins sending, or after period D.

After the event FE stations execute the following program:

ProgramSend:

```
IF source_address = previous_address
      THEN Send
FI
```

If a station has no data to send, it sends a token frame in order to give to the next station turn to send.

Ring initialization. The control station initializes the ring. It has initially address 0. The address of the control station is held in the variable control_station_address, local at every station. Control station also restarts the ring after failure.

When starting, all stations compute:

ProgramInit:

```
WHENEVER
      IF myaddress = control_station address AND BUS_EMPTY
            THEN my_turn_to_send FI
```

The event BUS_EMPTY is signalled by the counter T. Thus the station 0 always takes the initiative when there has been no signal on the bus for period T.

The T counter is started by the START – event from outside, or by the frame-ended or collision ended-events. It is also started, after execution of

ProgramModify below, when a station has taken the role of control station. It is cleared when a station begins to send, or after time period T.

Station failure. After sending a frame, if the bus remains empty after delay D, the next station has not begun to send, which creates the event NS.

Station then sends a broadcast management frame indicating the failed station. The address of the failed station is in the variable failed_station. Stations remove the failed station from the configuration list.

Event NS causes stations to execute ProgramModify:

ProgramModify:

```
Remove(failed_station_address)
  IF failed_station_address = PREVIOUS_ADDRESS
    THEN PREVIOUS_ADDRESS := PREVIOUS_ADDRESS – 1
  FI
  IF failed_station_address = control_station_address
    THEN control_station_address := control_station_address +1
            IF control_station_address = my_address
                THEN become_control_station
            FI
  FI
```

Thus, if a station fails, it is automatically removed from the list by all other stations. If control station failed, next station in the configuration list is selected to control station.

Re-entering station. A previously failed station wishing to rejoin the ring resets it's PREDECESSOR register value to the initial value, and restores the configuration list. It then listens the bus, and starts sending some defined signal after noticing any FRAME_ENDED on the bus, in order to cause a collision. All stations notice the collision, and reset their PREDECESSOR register to the initial configuration value, and restore the configuration list to the initial configuration.

ProgramReset:

```
BEGIN
    Reset
END
```

After this, the bus remains empty, until the control station restarts the operation. The operation continues with the new station added. Because the register

values at the stations correspond to the initial configuration, the protocol notices the non-existing stations as failed stations. The system reorders the configuration list in successive computations by ProgramModify, and sets the value of the PREVIOUS_ADDRESS register to the correct value.

Changing the control station. Due to ProgramModify, fail-stop of the cotrol processor causes next operating station to become control station.

C. Events and timeouts. Events related to the operation of the protocol are summarized below. Events and their timing relationships are depicted in Fig. 1.

FE = frame ended.

CE = collision ended.

S = next station starts sending a frame.

The events above are derived by circuits.

BE = BUS_EMPTY, expiration of the timer T.

NS = next station is not sending. Expiration of timer D at sending phase.

System has two timers:

T = bus activity timer,

D = interframe gap timer.

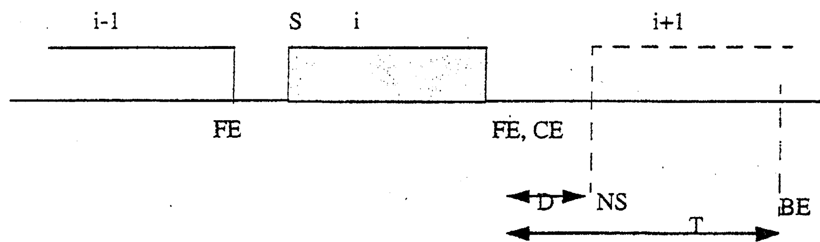

Fig. 1. Protocol event sequence.

3. The model

A. General framework. To prove the stabilizability properties of the protocol the following model is developed. The process at each station has the form:

Process

[ ] <event> ; <transition>
[ ] <event> ; <transition>
—

endproc

Each transition is a program of the form:

<guard> ⟶ < sequence of local statements >

Events are global. Happening of an event selects a transition. Transition is a program to be excuted. In the semantics of this model "transition becomes true" means, that the corresponding program is executed. Execution of a transition can cause events. When the guard becomes true, the sequence of local statements is executed. The guard is a boolean expression of local variables. Event variables have values derived from the physical events and timeouts by circuits. Local statements act only on local variables.

B. Configuration. The global state of the system is denoted by s. Computation is a sequence of states $\{s_0, s_1, \ldots, s_i, s_{i+1}, \ldots\}$, where s_{i+1} is the next state to s_i.

System starts in state s_0, when "nothing happens". In this state the predicate "BUS_EMPTY" is true. This predicate can also be true elsewhere during the computation, let's say in some intermediate state s'_0. This means, that the operating cycle is started from beginning by the responsible station.

In the system under consideration the computation is infinite.

The topology is a logical ring. There are N stations in the ring, numbered $0, \ldots, N-1$.

Notation. View is the partial state of system described by the predecessor registers of stations. The *state* of the system is denoted by (s, w), where w is the list describing the *view* of the system, and s is the rest of the state of the system.

Let $pre(i)$ be the variable denoting the predecessor-register of station i. The value of $pre(i)$ is the address of the predecessor of i.

The configuration C of the ring is the list of station identifiers or addresses

$$C = \{0, 1, 2, \ldots, i, k, \ldots, m\}.$$

List C describing the configuration is ordered circularry by the relation $<$.

The initial configuration is denoted by $C_{0},$

$$C_{0}, = \{0, 1, 2, \ldots, N-1\},$$

where N is the number of stations in the ring.

The list $W = \{pre(0),\ pre(1), \ldots,\ pre(n-1)\}$, $pre(i) \in C$ describes the partial *state* of the ring, called view.

The values of $pre(i)$ in a configuration are from the set of numbers $0, \ldots, N-1$.

The initial configuration of the ring can change during the operation due to the deletion and addition of stations. The actual configuration is denoted by C in the following.

DEFINITION 1. *Two adjacent stations i and j, $i < j$, in a configuration C are connected, iff $pre(j) = i$.*

DEFINITION 2. *Configuration C is connected, iff all pairs of adjacent stations are connected.*

In the initial configuration the system is assumed to be connected.

The operation of the ring is controlled by the transition $T0$:

$T0: \ <FE> \longrightarrow <\text{ProgramSend}>$.

Modification of the configuration is controlled byt guard $T1$.

$T1: \ <NS> \longrightarrow <\text{ProgramModify}>$.

Restart of the ring is done by guard $T2$.

$T2: \ <\text{BUS_EMPTY}> \longrightarrow <\text{ProgramInit}>$.

Reset of local configuration variables is controlled by $T3$.

$T3: \ <CE> \longrightarrow <\text{ProgramReset}>$.

Transition $T0$ becomes true after event FE (frame ended). Transition $T1$ becomes true when control frame is detected. $T2$ becomes true when counter T expires. $T3$ becomes true when end of collision is detected.

From the description of the protocol the following specification regarding transitions T can be written.

$S1$: Normal operation: $T2; T0; T0; \ldots$.

$S2$: Transitions: transition diagram is depicted in Fig. 2.

In the diagram the possible next events and transitions after a completed transition are shown. Labels on arcs denote the event causing the transition.

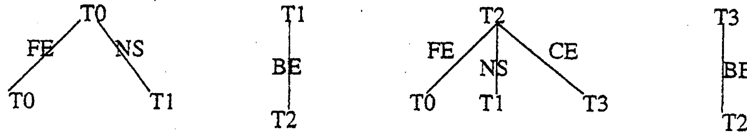

Fig. 2. Transition diagram.

The model is summarized in Table 1 below.

Table 1. Components of the model

Event	Transition	Guards	Guard expression
FE	$T0$, ProgramSend	CondSend	source_address = = previous_address
NS	$T1$, ProgramModify	CondModifyConf	source_address = = previous_address
		CondModifyCtrl	failed_station = = ctrl_station
		CondNewCtrl	my_address = = ctrl_station+1
T	$T2$, ProgramInit	CondInit	my_address = = ctrl_station
CE	$T3$, ProgramReset	CondTrue	true

4. Static failure modes

A. Station Failure. Stations are assumed to operate in the fail-stop-mode. Failure mode: station does not begin to send on it's turn. When a station failure occurs, station is totally stopped regarding the operation of the protocol. It is assumed that there are no other errors.

We adopt the following notation of variables:

x_1^i denotes the predecessor-register,

x_2^i denotes the own-address register of station i and has always value i.

Then the partial state at station i is described by the pair x_1^i x_2^i. Configuration C can thus be described as list of x_j^i values for every station i.

The computation of the protocol reaches the point, where the failed station should begin to send. The next possible event after this is event NS.

According to the description of the station failure procedure of the protocol, transition $T1$ becomes true.

Let $T1$ be true. Let the connected state W before the failure be

$$\dots\ x_1^{k-1}x_2^{k-1}\quad x_1^k x_2^k\quad x_1^{k+1}x_2^{k+1}\dots,$$

where $k-1$, k and $k+1$ are assumed to be adjacent in W. Let station k be the failing station.

After the computation of ProgramModify, the state becomes:

$$\dots\ x_1^{k-1}x_2^{k-1}\quad x_1^k x_2^{k+1}\dots,$$

where stations $k-1$ and $k+1$ are conneted. Because no other alterations to W were made, the new state is connected.

Computation of ProgramModify in the case of failed station $k-1$ in W makes substitutions x_1^{k+1}/x_2^{k-1} into neighbouring stations of k, and thus connects $k-1$ and $k+1$.

If the failing station k is a control station, then the computation of ProgramModify changes the control to the next station $k+1$.

Theorem 1. *In a ring of N stations there can be $n = N-2$ station failures.*

Proof. The proof follows from the repeated application of the procedure above. By definition here, at least two stations are needed to form a ring.

Aftter $T1$ transition $T2$ becomes true, and the control station starts the operation from the beginning.

$T2$ becomes true also when there are no stations able to send. Regarding to the assumed failure mode case this means that there are < 2 stations operable. In this case $T2$ = true will cause the control station ececute cycle $T1$; $T2$ forever.

If the failed station is the control station, then due to ProgramModify, control has changed to station $k+1$.

The failure mode when station fails during sending causes by assumption station to detect frame ended, FE. This causes $T0$ become true, and next station to begin sending. If failure was transient, operation continues normally. If failure is permanent, station is removed during the next cycle.

B. Insertion of stations. A station previously failed is inserted into the ring. New stations with new addresses can not be inserted. Failure mode: there is an operating station in the ring which has the same value in the PREDECESSOR register as the inserted station. Let the value be j.

The computation of the protocol reaches a point where the station with value j in the predecessor register should start sending. Two stations start sending, and collision occurs. The transitions that can become true are:

$T3: \ < CE > \longrightarrow <$ ProgramReset $>$.

ProgramReset resets the state from W to W'. After reset the bus is detected empty, counter T expires. Transition $T2$ becomes true, and station execute ProgramInit, which causes the control station to restart the ring.

After reset the system is in a state W', where the value of the PREDECESSOR at each station is the same as in the initial (full) configuration. If there were n missing stations in state W, then there will be $n-1$ missing stations in W'.

Because all the register values point to the stations in the original configuration, there are values that point to a missing station. The system then proceeds in the following steps:

Step 1. After $T2$ becomes true, the control station tries to start the operation of the actual ring by executing ProgramInit.

Step 2. Then, some station $i-1$ notices, that the next station i does not start sending. This causes transition $T1$ to become true. Thus, the system repeats the same sequence as in the station failure case, treating the missing station as a failed station. This results in a state W'' with neighbours $i-1$ and $i+1$ connected, as proved earlier.

If there are no other missing stations, the the state W'' is connected, and the normal operation can continue.

If there are other missing stations, then the system repeats the sequence $T1$, $T2$ for all missing stations. Thus the final state becomes connected. There are three possible subcases.

Let the state sequence corresponding to the local views after reset be W' (with k inserted).

Case 1. Station k is inserted to the tail of some connected subsequence (i, j) of W', after position j. There is no station existing immediately after k.

Let's denote the resulting sequence by $(\ldots, i, j, k, k+m, \ldots)$. Stations i

and $k+m$ are connected, and there are $m-1$ missing stations between k and $k+m$. In the connected sequence W before reset:

$$x_3^j = k + m, \quad x_1^{k+m} = j.$$

After reset and the execution of ProgramReset the state W' is:

$$x_3^j = k,\ x_1^k = j,\ x_1^{k+m} = k + m - 1,\ x_1^k = j,\ x_3^k = k + 1.$$

From the values above we see stations j and k become connected after reset by their initial values. Station k is not connected to any other station, so that the state W' is not connected.

The values of the variables of the $m-1$ stations between k and $k+m$ are their initial values. This means, that some stations have in their PREDECESSOR registers address values of stations, which do not exist in actual ring. Thus, some station next in turn will not start sending. Let this station be i, which is the next station from the control station, which has wrong predecessor address, denoted as wrong view.

The transition $T1$ will be true.

$< NS > \longrightarrow <$ ProgramModify $>$.

ProgramModify makes the substitution x_1^{k+2}/k at the station $k+2$. But the station $k+2$ does not exist, and therefore there is no program execution at $k+2$.

The next possible transition to become true is $T2$:

$< S,$ BUS_EMPTY $> \longrightarrow <$ ProgramInit $>$.

The sequence of transitions $T2,\ T1$ will continue, until for the station $k+m$ ProgramModify is executed, and the value substituted is x_1^{k+m}/k. Thus, the state of the ring becomes eventually connected. If there are M stations in the new ring with the added station included, then the ring is connected after $N-M$ restarts. In this case it was assumed, that stations i and $k+m$ are connected. Relaxing this assumption, however, will not essentially change the proof. Steps 1 and 2 connect also all missing stations, which are not part of the sequence $(k, k+m)$.

Case 2. Station k is inserted to the head of a connected subsequence $(k+1, k+j)$ of W' to position k. There is no station existing immediately before k.

Let's denote the resulting subsequence by $(\ldots, l, m, \ldots, k, k+1, k+2, k+j \ldots)$. There are $n-1$ deleted stations between m and k. Stations l and $k+j$ are connected.

The original value in W is

$$x_1^{k+1} = m.$$

After reset the state W' is:

$$x_1^k = k - 1, \quad x_1^{k+1} = k.$$

Thus k and $k+1$ become connected. The configuration is not connected, however, because m and k are not connected. There are $n-1$ missing stations between m and k. Then Steps 1 and 2 are repeated for all $n-1$ missing stations between m and k, starting from m. Eventually the system becomes connected. This requires $N - M$ restarts. In the same way the Steps 1 and 2 connect all other missing stations, which are not part of the sequence (m, k).

Case 3. The inserted station is k. There is no station existing immediately before k or after k in W'.

Let's denote the resulting configuration by $(\ldots, k-n-1, k-n, \ldots, k, \ldots, k+m, k+m+1, \ldots)$. There are $n+m-2$ missing stations. The proof is easy by combination of Case 2 and Case 1, in that order.

Case 4. Multiple stations moved, multiple added. It can be deduced from the definition of the protocol, that when two or more stations reenter simultaneously, the effect on the local views of the stations is the same as when one station reentering. Simultaneouly means, that the reentering stations all enter within the same collision interval. Because the addition algorithm adds stations in numbering order and independent of the higher number stations, this case reduces to the Case 2 above.

Multiple entering stations can cause diferent collision intervals. If a collision interval occurs before the execution of the algorithm caused by the previous enter has ended, then the protocol will force the system to repeat the addition algorithm. Because the algorithm adds stations in numerical order and independent of higher order stations this case also reduces to the Case 2 above, and eventually all stations will be added.

C. Example. The Case 3 is described by an example below. In this example there are six stations. Numbers 1, 3 and 2 fail in that order. Station number 3 will recover and is reentered into the ring. Missing stations are underlined.

Stations	0	1	2	3	4	5
	Initial global view:					
	5 0	0 1	1 2	2 3	3 4	4 5
	1 fails , - -, ProgramModify.					
	5 0	- -	0 2	2 3	3 4	4 5
	3 fails, -					
	5 0	- -	0 2	- -	2 4	4 5
	2 fails , -					
	5 0	- -	- -	- -	0 4	4 5
	2 coming up.					
	5 0	- -	0 2	- -	0 4	4 5
	Insert 2, ProgramReset, 1 not sending.					
	5 0	0 1	1 2	2 3	3 4	4 5
	Insert 2, ProgramInit, by 0, ProgramModify.					
	5 0	- -	0 2	2 3	3 4	4 5
	Insert 2, - -, ProgramModify.					
	5 0	- -	0 2	- -	2 4	4 5

5. Communication errors

A. Event failure. Event FE is the only event broadcasted. We consider the case, where the event FE is not properly interpreted.

Sender i sends a frame, station j misses the event Frame_Ended. In the case that j is not the next station to send, the failure has no effect, since this event causes no transition in other stations than $i+1$. If station $i+1$ misses the event FE, then event NS will be true at all stations except $i+1$. After D, transition $T1$ occurs at every station except $i+1$. This causes all station except $i+1$ to modify their configuration list. After the modification there are station $i+1$ and $i+2$ pointing to i as their predecessor station. The next event, that can be true is expiration of timer T. After this control station makes transition $T'2$, and the system restarts. But because there are two stations after i, there will be collisison. Thus, transition $T3$ will be executed by all stations after CE, and all stations have the initial configuration. The operation will return to normal through a procedure similar to that explained in chapter 3.2 before.

B. Transmission errors. There can be transient error in the addresses, and we consider the following cases: there is a wrong or not-existing sender, the

next address is wrong, or there is wrong or not-existing receiver address. We consider each case separately. The address of the sending station of the frame cur rently on the bus is kept in the SOURCE_ADDRESS register of each station.

Case 1. Wrong sending address. The effect is, that some station j, which is not next in turn, will notice the correct event FE, and will start sending. Operation continues correctly, but stations between the expected sender $i+1$ and j will miss the turn at this cycle.

Case 2. Not-a-sender address. The effect is, that no station will notice event FE from its predecessor, and no station will start sending. The effect is, that eventually transition $T1$ will become true, and station $i+1$ is removed from the onfiguration. Then both station $i+2$ and $i+1$ will have their predecessor value $x_1 = i$.

This causes collision after next FE from station i. The recovery will take place through transition $T3$. After this, transition $T2$ will be true. Thus the control station will execute ProgramInit and will restart the operation.

Case 3. Receiver is not existing or wrong. Message will not reach it's destination. Because the receiving process will not cause any events related to the operation, no transitions will be affected and thus the operation will continue normally.

C. Timing errors. Delay in computation can cause timing errors. We consider the case when computation at station is temporarily delayed. Sender is station i.

– When a station $i+1$ has failed and is not sending, stations will notice the event NS.

– Transition $T1$ will cause station i to execute ProgramModify and have CondModifyConf true. No station has CondSend true.

– Eventually counter T expires, condition CondInit will be true at the control station, and the operation restarts. Program at $i+2$ shoud change the value of the x_1^{i+2} register before time $(i+2)\Delta$ from the restart in order to continue sending at this cycle. Δ is the time to pass the sending turn from a station to another. Total time to make the computation is thus $T - D + (i+2)\Delta$.

By assumption the delay in program execution was transient, so that operation will eventually continue normally.

Timing errors also arise, when the timers expire too early. Avoiding this kind of errors is a practical matter of setting timer values according to the actual situation.

If timer T expires erroneously before D, condition CondInit will be true at the control station, and control station starts sending. This clears timer D. Operation continues normally. Other conditions for time delay T are discussed below.

D. Changing the control station. Let's assume, that control station fails during the operation and not before initialisation. Then after the sending of station $N - 1$ transition $T1$ eventually becomes true. Station ctrl_station+1 executing ProgramModify will have CondModifyCtrl and CondNewCtrl true and becomes control station, and all other stations will also execute ProgramModify and have CondModifyCtrl true and compute the change. After this transition $T2$ eventually becomes true, and CondInit becomes true at the new control station which restarts the operation. The operation of the system requires, that all computations are made before timer T expires. The total time available for computation is $T - D$.

From the discussion above in paragraphs C and D it can be concluded, that the max. time for making transitions at station is $T - D$ for transistion $T1$ when control station is changed, and $T - D + (i + 2)\Delta$ otherwise for the same transition. Other transitions have no time limits set by timers.

6. Conclusion. A stabilizing protocol for an event driven token bus network is presented. The protocol is applicable to systems using broadcast-type media as the communication media.

The system is self-configurating. After processor failures it can reach a stable configuration where the operation continues normally. This is done in finitely many computation steps. Faulty processor is identified. Processor failure can also be transient in the sense that the processor can later recover and join the system. Fault-tolarance is achieved by the use of timeouts and a distinguished control processor. The protocol could be, however, to be distributed in the sense that in the case of a failure of the control processor some other processor takes the role.

Model for the protocol was developed, and prove of the stabilizing properties was carried out. The system will tolerate multiple station failures. Stations can be inserted. Operation of the protocol does not assume reliable communication.

System can recover from broadcast and communication address errors in finite steps. It was also proved, that the system will recover from transient timing errors of the protocol timer, and delays in the program execution.

REFERENCES

Arora, A., and M. Gouda (1993). Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering,* **19**(11), 1015–1027.

Brown, G.M., M.G. Gouda and C.-L. Wu (1989). Token systems that self-stabilize. *IEEE Tr. on Computers,* **38**(6), 845–852.

Burns, J.E., and J. Pachl (1989). Uniform self-stabilizing rings. *ACM Transactions on Prog. Lang. and Syst.,* **11**(2), 330–344.

Dijktsra, E.W. (1974). Self-stabilizing system in spite of distributed control. *Comm. ACM,* **17**(11), 643–644.

Flatebo, M., and A.K. Datta (1994). Two-state self-stabililizing algorithms for token rings. *IEEE Transactions on Software Engineering,* **20**(6), 500–504.

Gouda, M.G., and N.J. Multari (1991). Stabilizing communication protocols. *IEEE Trans. Comp.,* **40**(4), 448–458.

Laranjeira, L.A., M. Malek and R. Jenevein (1993). Nest: a nested-predicate scheme for fault tolerance. *IEEE Transactions on Computers,* **42**(11), 1303–1324.

Rangarajan, S., Y. Huang and S.K. Tripathi (1995). Computing reliability intervals for k-resilient protocols. *IEEE Transactions on Computers,* **44**(3), 462–466.

Sintonen, L. (1990). Event driven bus architecture for bounded area networks. *Proc. IECON'90,* Vol. I. pp. 539–541.

Sobh, T.M. (1991). Discrete-event dynamic systems. *Rep. GRASP LAB,* Univ. of Pennsylvania. 35 pp.

Received March 1996

L. Sintonen is an associate professor at the Software Systems Laboratory at the Department of Informatics, Tampere University of Technology. He received his Dr. Degree in Computer Science in 1980, from Tampere University of Technology. His current research interests are modelling and analysis of distributed systems and protocols.

PROCESŲ TINKLO PROTOKOLO STABILIZAVIMAS

Leo SINTONEN

Daugeliui dalykinių sričių pageidautina, kad trikių atvejais kooperuojančių procesų koordinavimas nesutriktų. Šiame darbe aprašomas trikius toleruojantis protokolas tokiam procesiorių tinklui, kuris organizuotas kaip loginis žiedas, sukurtas naudojant kokią nors fizinę vienpusio pranešimų perdavimų terpę. Aprašomasis protokolas atkuria normalų procesų vyksmą po laikino kurio nors proceso trikio, po nepašalinimo kurio nors proceso trikio ir po laikino ryšio trikio. Darbe nagrinėjamas protokolas bei jį naudojančių sistemų modelis ir įrodoma, kad po trikių per baigtinį žingsnių skaičių tokiose sistemose bus atkurta stabili būsena, kurioje sistema turės korektišką konfiguraciją.