Distrib Comput (1996) 9: 157–171

DISTRIBUTED COMPUTING

Design and analysis of dynamic leader election protocols in broadcast networks

Jacob Brunekreef[1], Joost-Pieter Katoen[2], Ron Koymans[3], Sjouke Mauw[4]

[1] Programming Research Group, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
[2] Department of Computing Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
[3] Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
[4] Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Received: September 1993 / Revised: September 1995

Summary. The well-known problem of leader election in distributed systems is considered in a dynamic context where processes may participate and crash spontaneously. Processes communicate by means of buffered broadcasting as opposed to usual point-to-point communication. In this paper we design a leader election protocol in such a dynamic context. As the problem at hand is considerably complex we develop the protocol in three steps. In the initial design processes are considered to be perfect and a leader is assumed to be present initially. In the second protocol, the assumption of an initial leader is dropped. This leads to a symmetric protocol which uses an (abstract) timeout mechanism to detect the absence of a leader. Finally, in the last step of the design processes may crash without giving any notification of other processes. The worst case message complexity of all protocols is addressed. A formal approach to the specification and verification of the leader election protocols is adopted. The requirements are specified in a property-oriented way and the protocols are denoted by means of extended finite state machines. It is proven using linear-time temporal logic that the fault-tolerant protocol satisfies its requirements.

Key words: Broadcast network – Communication protocols – Finite-state machines – Leader election – Message complexity – Protocol design, specification, and verification – Temporal logic

1 Introduction

In current distributed systems several functions (or services) are offered by some dedicated process(es) in the system. One might think of address assignment and registration, query coordination in a distributed database system, clock distribution, token regeneration after token loss, initiation of topology updates, load balancing, and so forth. Usually many processes in the system are potentially capable to offer such a functionality. However, for consistency reasons at any time only one process is allowed to actualy offer the function. Therefore, one process – called the "leader" – must be elected to support that function. Sometimes it suffices to elect an arbitrary process, but for other functions it is important to elect the process with the best capabilities to perform that function. In the latter case the ranking of processes is application-dependent. For instance, when using a leader for distributing the clock the ranking may be based on local clock skews, whereas for load balancing the ranking may be based on the process' local load. In this paper we abstract from this and use ranking on basis of process identities.

In this paper we consider a distributed leader election (LE) protocol and use ranking of processes on basis of process identities. Each process has a fixed unique identity and a total ordering exists on these identities, known to all processes. We assume a finite number of processes. The leader is defined as the process with the largest identity among all participating processes. Realistic distributed systems are subject to failures. The problem of leader election thus becomes of practical interest when failures are anticipated. In this paper, processes behave *dynamically* – they may participate at arbitrary moments and stop participating spontaneously without notification to any other process. Crashed processes may recover at any time. Thus, a leader has to be elected from a set of processes whose elements may change continuously. Processes communicate with each other by exchanging messages via a *broadcast* network. This network is considered to be fully reliable. A broadcast message is received by all processes except the sending process itself. Communication is asynchronous and order-preserving.

Leader election is a special case of distributed consensus problems. Several impossibility results have been obained for such problems. For instance, in [14] a number of orthogonal characteristics is identified by which the existence of a solution for the distributed consensus problem is determined. According to this classification our problem is solvable since we consider order preserving message delivery, broadcast communication and atomic send and receive.

Correspondence to: J. Brunekreef

Due to the complexity of the design of a fault-tolerant LE protocol the principle of separation of concerns is applied. That is, we develop a fault-tolerant protocol in three steps, each step resulting in a LE protocol. We start with rather strong – and unrealistic – assumptions about process and system behaviour. In each subsequent step these assumptions are weakened and a protocol is constructed starting from the protocol derived in the previous step. The steps of our design are as follows. In our initial design processes are considered to be perfect and a leader is assumed to be present initially. A process may participate spontaneously, but once it does it remains to do so and does not crash. In the second step, the assumption of an initial leader is dropped. This leads to a fully symmetric protocol which uses an (abstract) timeout mechanism to detect the absence of a leader. Finally, in the last step of our design processes may crash without giving any notification to other processes.

Existing designs of LE protocols are mainly focussed on reducing message and time complexity, scarcely paying attention to protocol verification, let alone providing a formal approach to verification. However, for the design of complex communication protocols formal methods are indispensable. The starting-point of our designs is a requirements specification in linear-time temporal logic. Temporal logic is an appropriate and expressive language for specifying properties and behaviours of reactive systems, like communication protocols, in an abstract way. As a protocol specification language we adopt extended finite state machines. The combination of temporal logic and state-transition diagrams enables a formal verification of the designed protocols. In [8] such a verification is carried out for all presented protocols. In this paper we present the proof of the most sophisticated protocol, the fault-tolerant protocol.

As efficiency plays an important role in the design of leader election protocols some complexity results are given for each protocol presented in this paper. We focus our analysis on the worst case message complexity which indicates the maximum number of broadcast messages needed to elect a leader. For N participating processes the message complexity of our initial protocol is $\mathcal{O}(N^2)$ broadcast messages, which can be improved to $\mathcal{O}(N)$ by adopting a tricky way of message buffering. Using this buffer mechanism the last two protocols have a message complexity of $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$ broadcast messages, respectively, when no crashing processes are considered.

The paper is further organized as follows. In section 2 the relation to existing work is presented. The requirements specification and design of all three protocols is presented in section 3. Furthermore, an introduction to the protocol description language and to linear-time temporal logic is given in this section. In section 4 it is verified using temporal logic that the third protocol from section 3 satisfies its requirements. The worst case message complexity of all three protocols is presented in section 5. Finally, in section 6 some concluding remarks are given and future work is addressed.

In the rest of this paper, we use the term protocol as a synonym for similar terms as distributed program, distributed algorithm, and so forth.

2 Relation to other work

Leader election protocols

The problem of leader election was originally coined by [31] in the late seventies and various LE protocols have been developed since then. A broad range of solutions exists varying in network topology (ring [11, 31, 37], mesh, complete network [2, 26, 42], and so on), communication mechanism (asynchronous, synchronous), available topology information at processes [4, 32], and so forth.

Only a few LE protocols using broadcast communication have been treated (see e.g. [20, 24]). A possible straightforward solution to a broadcast network is to superimpose a (virtual) topology – like a ring – on it and to adopt a well-known solution for this topology. However, most existing solutions are aimed at distributed systems that are assumed to behave perfectly – no failures are anticipated and a fixed number of participating processes is assumed. Moreover, the specific characteristics of broadcasting are not exploited.

Realistic distributed systems are subject to failures. Some LE protocols are known that tolerate either communication link failures (see e.g. [1, 41]) or process failures [12, 13, 20, 23, 24, 34]. In [20] the LE problem with a similar failure model and using broadcast communication is considered, however, no ordering between processes is considered. A number of LE protocols for various different kinds of broadcasting (reliable, unordered, ordered broadcasting) are presented in [24]. The authors of this paper consider, however, synchronous broadcast mechanisms such that processes reply without any delay, and consider processes to have synchronized clocks. [23] and [34] only consider process crashes prior to the start of the protocol, but no crashes during protocol execution are taken into account. We consider processes to be able to crash at any moment of time. In [12, 13] LE protocols are presented that tolerate transient process failures. These protocols belong to the category of self-stabilizing protocols [15, 38].

Specification and verification in temporal logic

Existing LE protocols are mainly focussed on reducing message and time complexity, scarcely paying attention to problem specification and protocol verification. To our knowledge no formal specification of the (dynamic) LE problem is published elsewhere. In order to correctly design (and verify!) communication protocols such a formal specification is indispensable. The specification and verification techniques we use are well-known for almost a decade: protocol specification and verification using a combination of temporal logic [33] and state-transition diagrams [6] has been applied for a number of other protocols (see e.g. [28, 21, 40]). However, the dynamic character of processes combined with a timeout mechanism so as to detect the absence of a leader makes the specification and verification more complex than traditionally considered communication protocols.

Complexity results

Numerous complexity results for LE protocols are known. We mention only a few and concentrate on worst case message and time complexity. First, consider point-to-point networks with N processes participating in the election. Early protocols for an unidirectional ring network [11, 31] have a message and time complexity of $\mathcal{O}(N^2)$ and $\mathcal{O}(N)$, respectively. These results have been improved (see e.g. [37, 30]) to protocols with a message and time complexity of $\mathcal{O}(N \log N)$ and $\mathcal{O}(\log N)$. For complete networks a message and time complexity of $\mathcal{O}(N \log N)$ and $\mathcal{O}(N)$ can be reached, see [2, 3, 25, 32]. In [42] a number of LE protocols for asynchronous complete networks is given with a message complexity of $\mathcal{O}(Nk)$ and a time complexity of $\mathcal{O}(N/k)$, with k a constant, $\log N \leqq k \leqq N$.

To elect a leader, the broadcast LE protocols defined in [24] have a worst case message complexity of $\mathcal{O}(N)$ reliable ordered broadcast messages. On failure of a leader the election must be repeated involving all remaining processes, resulting in an overall message complexity of $\mathcal{O}(N^2)$. They consider synchronised processes and synchronised communication.

3 Design of LE protocols

3.1 Introduction

In this section we describe the communication mechanism used for our protocols, introduce the protocol description language, and introduce linear-time temporal logic.

3.1.1 Communication

Processes communicate with each other by exchanging messages via a broadcast network. A broadcast message sent by some process p is received instantaneously by all processes except p itself. There is no need for separate point-to-point messages to accomplish this. In contrast with a multi-process rendez-vous in which several processes synchronize on a common communication, broadcasting is considered to be *asynchronous*. Broadcast messages are buffered by processes, resulting in a so-called buffered broadcast [17] mechanism. This buffering is order preserving. In this paper the only form of communication we consider between processes is broadcasting. Therefore, we often omit the prefix broadcast in terms like message, communication, and so on.

It is assumed that the communication network is perfect, that is, no duplication, loss or garbling of messages takes place. In this way we abstract from the design of a reliable broadcast facility on a faulty broadcast network and assume the existence of such a protocol. Protocols that establish a reliable ordered broadcast mechanism have been treated extensively in literature (see e.g. [36, 39]). In order to avoid interference of transmissions of different processes it is assumed that at most one message may be transmitted via the network at any moment of time.

The ability of broadcasting communication is often treated as a special feature of the communication network. As a result, existing notations for concurrent (and distributed) processes – like CSP [22], Estelle [10], and so on – do not provide a primitive by which a process can explicitly broadcast a message. Here we consider broadcasting as part of our description language (see also [17]).

3.1.2 Protocol description language

We denote our protocol by a Finite State Machine (FSM) diagram [6], also called *state transition diagram*. Transitions consist of an (optional) guard and zero or more actions. Depending on the guard a transition is either *enabled* or *disabled*. In a state the process selects non-deterministically between all enabled transitions, it performs the actions associated with the selected transition (in arbitrary order) and goes to the next state. When there are no enabled transitions the process remains in the same state. Evaluation of a guard, taking a state transition and executing its associated actions constitutes a single *atomic event*.

A message consists of a message type and one or more parameters. $m(p_1, \ldots, p_n)$ denotes a message of type m with parameters p_1 through p_n. The sending of this message is denoted by $!!m(p_1, \ldots, p_n)$. At execution of the send statement by process p, say, the message is buffered instantaneously at each process except p. Since broadcasting is asynchronous, execution of $!!m(\cdots)$ is never delayed due to unreadiness of a receiving process. (Notice that this means that a process must always be able to buffer a message received via the network.) Execution of $??m(\cdots)$ by a process delays that process until a message of type m is delivered. Messages sent by $!!m(\cdots)$ can be received only by $??m(\cdots)$, so corresponding input and output actions must affect the same message type and the same number of parameters (and the same parameter types). Communications can be viewed as (possibly delayed) distributed assignments, that is, for processes p and q, variables x_i and expressions E_i $(0 < i \leqq n)$ execution of $!!m(E_1, \ldots, E_n)$ in p and $??m(x_1, \ldots, x_n)$ in q establishes the multiple assignment $x_1, \ldots, x_n := E_1, \ldots, E_n$ (in q).

Guards are boolean expressions. We allow receive actions to appear in guards. This part of a guard is true only when execution of the receive action causes no delay, that is, when the corresponding message is at the head of the process' buffer. An absent guard denotes a guard that is always true.

When in a certain state a message type is at the head of the buffer for which no corresponding transition is present this is considered to be an error. This situation is called *unspecified reception* and leads to a deadlock of the system.

A process consists of a buffer process taking care of buffering messages received via the communication network, and a 'main' process. The buffer processes are left implicit – they operate according to the first-in first-out principle, and are at any moment of time ready to accept an input of the network and to offer an earlier received message to the main process. A main process is denoted by a FSM and the co-operation of these processes is considered to be the parallel composition of these FSMs. The reader should bear in mind that all processes in our system are equivalent (apart from their identity). Thus the system is the parallel composition of a number of equivalent

FSMs. The individual FSMs co-operate by exchanging messages in the way described above. The parallel composition is based on a *fair interleaving semantics* where each process gets its turn infinitely often. Furthermore, a transition has to be taken eventually when it is continuously enabled ('weak fairness' [33]).

3.1.3 Introduction to temporal logic

For our formulation of the requirements of our protocol and the subsequent verification that our protocol meets these requirements we use a first-order temporal logic based on the temporal operators $\mathcal{U}$ and $\mathcal{S}$ (see also [33]). An extensive introduction to the use of temporal logic for communication protocols can be found in [18].

A temporal formula is constructed from predicates, boolean operators (such as $\neg$ and $\wedge$) and temporal operators like $\Box$ (pronounce 'always'), $\Diamond$ ('eventually'), $\mathcal{U}$ ('until'), $\mathcal{W}$ ('unless'), $\bigcirc$ ('next'), $\blacksquare$ ('always in the past'), $\blacklozenge$ ('some time in the past'), $\mathcal{S}$ ('since') and $\mathcal{J}$ ('just'). Let φ and ψ be arbitrary temporal formulas. We consider the future (and the past) in a *strict sense*, that is, the current moment is excluded. Informally speaking, $\Box\varphi$ means that φ will be true at every moment in the future. $\Diamond\varphi$ means that φ will be true at some moment in the future, and $\varphi\mathcal{U}\psi$ means that ψ will become true eventually and that φ will be true continuously until that moment. $\varphi\mathcal{W}\psi$ means that either φ holds indefinitely or $\varphi\mathcal{U}\psi$ holds (weak until). $\bigcirc\varphi$ means that φ holds at the next moment in time (our time domain is discrete since we use sequences, see below). The temporal operators which refer to the past are informally defined as follows. $\blacksquare\varphi$ means that φ has been true at every moment in the past, $\blacklozenge\varphi$ means that φ has been true at some moment in the past, and finally, $\varphi\mathcal{S}\psi$ means that ψ has been true at some moment in the past and that φ has been true continuously since that moment. $\mathcal{J}\varphi$ means that φ has just become true. At each moment of time the predicate *true* holds. Predicate *false* equals $\neg$ *true*.

The formal semantics of our form of temporal logic is defined by interpreting temporal formulas in a model. We consider a (possibly infinite) sequence s of states $(s_0, s_1, \ldots, s_n, \ldots)$ starting from the initial state s_0. A model is a sequence s together with a valuation function V assigning a subset of states to each predicate (giving the states in which the predicate is true). Given a model (s, V) the meaning of temporal formulas is defined by a satisfaction relation (denoted by $\models$) between the model and the current state (represented by its number in s), and a temporal formula. This satisfaction relation holds if and only if the formula is true in that state in the model. For $s = (s_0, s_1, \ldots, s_n, \ldots)$ and φ, ψ arbitrary temporal formulas, $\models$ is defined as follows:

$s, V, n \models P$ iff $s_n \in V(P)$ for each predicate P

$s, V, n \models \neg\varphi$ iff $s, V, n \not\models \varphi$

$s, V, n \models \varphi \wedge \psi$ iff $s, V, n \models \varphi$ and $s, V, n \models \psi$

$s, V, n \models \varphi\mathcal{U}\psi$ iff there exists $m > n$ such that $s, V, m \models \psi$ and $s, V, i \models \varphi$ for all i with $n < i < m$

$s, V, n \models \varphi\mathcal{S}\psi$ iff there exists m with $0 \leqq m < n$ such that $s, V, m \models \psi$ and $s, V, i \models \varphi$ for all i with $m < i < n$.

In our requirements (section 3.2.1 below) and our verification (section 4), all formulas should be interpreted to hold for all states (i.e. $\forall n: n \geqq 0$). The semantics of the remaining temporal operators can now be defined for arbitrary φ and ψ as follows:

$\Diamond\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$

$\Box\varphi \equiv \neg\Diamond\neg\varphi$

$\bigcirc\varphi \equiv \mathit{false}\,\mathcal{U}\,\varphi$

$\varphi\mathcal{W}\psi \equiv \Box\varphi \vee \varphi\mathcal{U}\psi$

$\blacklozenge\varphi \equiv \mathit{true}\,\mathcal{S}\,\varphi$

$\blacksquare\varphi \equiv \neg\blacklozenge\neg\varphi$

$\mathcal{J}\varphi \equiv \varphi \wedge \neg\varphi\,\mathcal{S}\,\neg\varphi.$

Predicate $\mathcal{J}$ characterizes the initial state (i.e., $n = 0$) and is equivalent to $\blacksquare$*false*. As usual the unary operators bind stronger than the binary ones. The temporal operators $\mathcal{S}$, $\mathcal{U}$, and $\mathcal{W}$ bind equally strong and take precedence over $\wedge$, $\vee$, and $\Rightarrow$. $\Rightarrow$ binds weaker than $\wedge$ and $\vee$, and $\wedge$ and $\vee$ bind equally strong.

3.2 A first stepping stone

In this section we design a leader election protocol assuming that a leader process is present initially and processes do not crash. We start by defining the precise requirements of the problem.

3.2.1 Requirements in temporal logic

The formulation of the requirements is as abstract as possible, that is, without reference to a possible protocol. In particular we refrain from mentioning certain states of the protocol. We only use a predicate *leader*(i) which represents the fact that the process with identity i is the current leader. This identity i is part of a finite set Id totally ordered by $<$. We use i, j, k to denote elements of Id.

In our requirements we use quantification over Id. By default, this quantification should be interpreted in a restricted way in the sense that not all identifications are involved in this quantification (the whole set Id), but only those identifications corresponding to the processes actually participating at that moment (so, always a finite subset of Id). We could have made this explicit by introducing the set, $\mathcal{P}$ say, of participating processes and replacing $\forall i: \ldots$ by $\forall i \in \mathcal{P}: \ldots$ and replacing $\exists i: \ldots$ by $\exists i \in \mathcal{P}: \ldots$. For ease of notation we have left this intended form of quantification implicit. In cases where quantification over the whole set Id is needed, this is explicitly indicated.

The requirements for leader election are as follows. The most basic requirement states that there must always be at most one leader. Since a change of leadership may take some time there can be temporarily no leader at all.

P1 $\forall i, j \neq i: \mathit{leader}(i) \Rightarrow \neg\mathit{leader}(j)$.

If we just take the above requirement we can easily devise a protocol by just not electing a leader at all. We should also state that there will be 'enough' leaders in due time. Because we are working in a framework using a qualitative notion of time this should be formulated by the liveness requirement below that there will be infinitely often a leader (this does not imply that there will be infinitely many leaders).

P2 $\Diamond(\exists i: leader(i))$.

The last two requirements make sense of the order $<$ on *Id*. The idea is that processes with a higher identity have priority in being elected as leader over processes with a lower identity. P3 states that a leader in the presence of a participating process with a higher identity will capitulate eventually. We do not state anything about the possible future leadership of this 'better' process.[1]

P3 $\forall i, j > i: leader(i) \wedge \neg leader(j) \Rightarrow \Diamond \neg leader(i)$.

Observe that j is a participating process. For reasons of efficiency it is not desirable that a leader eventually capitulates in presence of a 'sleeping' process that may participate at some unknown time in the future.

The last requirement states that the next leader will be an improvement over the previous one (i.e., will have a higher identity).

P4 $\forall i \in \mathscr{P}, j \in Id: leader(i) \wedge \bigcirc \neg leader(i)$
$\wedge \Diamond leader(j) \Rightarrow i < j$,

where we refer to the last moment of leadership of process i (first two conjuncts in premise) and the succession of process j (third conjunct). As process j does not need to participate at the moment i is a leader, it ranges over *Id*.

The last two requirements impose constraints on the capitulation of a leader process and the ordering of its successor. Note that P4 implies that a process that capitulates once, will not become a leader any more.

3.2.2 A first protocol

In this section we construct a LE protocol starting from requirements P1 through P4. To keep the design manageable it is assumed that a leader is present initially and all other processes are 'asleep'.

Each process has a fixed unique identity. Initially processes only have their own identity at their disposal (*my_id*) and have no knowledge of other processes' identities. The processes that do not yet take part in the election decide – non-deterministically – whether to join the election or not. Thus, a subset of all processes actually takes part in the election.

Initially a process does not know the identity of the leader, and, consequently it can not decide whether it becomes a leader or not. Once the identity of the leader is known there are two possible outcomes: the process should become (the new) leader or not. From the above we conclude that a process may be in one of the following possible states: *candidate*, when it does not yet know whether it will become a leader or not, *leader* when it actually is a leader, and *defeated* when it is defeated. A process starts in the *start* state.

Once a process joins the election, that is, when it becomes a candidate, it transmits its identity *my_id* by means of an *I*(*my_id*) (Identify) message. On receipt of an identity a leader compares this identity with its own identity. In case the received id is larger than its own id the leader moves to the defeated state (there is a 'better' process), and gives the candidate the right of succession by transmitting the candidate's id with an *R*-message (Response). In the other case, the leader remains leader and transmits its own id using *R*(*my_id*). The actions of a candidate on receipt of an identity follow quite straightforward – when it receives an *R*-message with its own id it becomes a leader, when it receives an *R*-message with a larger id it becomes defeated, and otherwise it remains a candidate.

There is however a little flaw in the above informally described protocol: when two (or more) processes are in the candidate state and one of them causes the leader to capitulate (i.e., to become defeated) the rest of the candidates may not receive a response of the leader on their original *I*-message, remaining candidate forever. The problem is that the 'old' leader capitulates while the 'new' leader has already processed (and ignored) the *I*-messages of the other candidates, while being a candidate. This problem is resolved by letting a candidate (re-)transmit its own id on receipt of an *R*(*id*) message with *id* < *my_id*.

We thus obtain the following protocol (see Fig 1). Some notational remarks are in order. States are represented by rounded boxes and transitions are denoted by arrows. The operator & should be read as "such that". Transition labels consist of an optional guard and an optional set of actions separated by a horizontal straight line. The initial state is indicated by having a grey color.

We deliberately have chosen to only permit the leader to deal with succession inquiries – i.e. only on receipt of

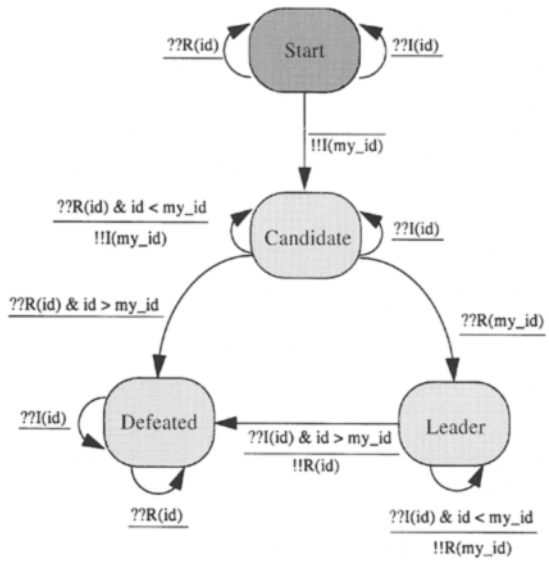

Fig. 1. Finite state machine diagram of Protocol 1

[1] Note that the assumption that j is not a leader is already guaranteed by P1, and could thus in principle be omitted

a notification of the leader a candidate becomes either leader or defeated. To accomplish this, messages originated by the leader are distinguished from those originated in other states. In the setting of this protocol it is essential that candidates do not become defeated (or, even worse, leader) on receipt of messages from other candidates; otherwise P2 may be violated. This is exemplified by the following scenario in which there is only a single message type. Consider three processes i, j, and k, one of which is a leader, k, say. Assume i and j do not take part in the election yet. Let $i > j > k$. Suppose j joins the election by transmitting its identity. Since i is still in the start state it ignores j's id. Before k reacts on the receipt of j's id, i joins the election and transmits its id. On receipt of this message j becomes defeated. As k capitulates (due to j's id received earlier) and as j will not become its successor (due to i's id) no process is able to grant i the right of succession, and, consequently, no leader process willl ever be elected. The problem is that a candidate may not only be forced to become defeated by the leader process, but also by other candidates. Therefore, we distinguish between id's originating from candidates and those submitted by leader processes. Candidates become either defeated or leader only on receipt of messages from leaders and they ignore others. In the above example j will thus not become defeated on receipt of i's id.

3.3 A symmetric LE protocol

We now drop the unnatural assumption of a leader being present initially. In this section we design a LE protocol starting from the previous protocol in case no leader may be present initially. Like in the previous section processes are considered to be perfect and the protocol has to be consistent with respect to requirements P1 through P4.

Let us first remark that in the current setting Protocol 1 does not suffice as it does not satisfy P2 – no leader will ever be present in case a leader is absent initially. The problem now is that a candidate must be able to detect the absence of a leader.

To solve this problem each process is equipped with a *timer* and the absence of a leader is notified by means of a timeout mechanism. A timer is started by the *start-timer* action. Typically, timeout operations induce the specification of execution times of protocol operations and the propagation delay of the communication medium. This tends to a rather complex semantics of timeout operations and complicates the verification significantly. Therefore, we adopt an abstract notion of timeout, which defines *what* a timeout condition achieves, but not *how* this is achieved. That is, it is abstracted from how to implement timeouts using a kind of (synchronized) clock mechanism. An identical treatment of timeouts has recently been given in [19].

A *timeout* is modeled as an ordinary action and may appear as (part of) a guard. Timeouts can be used to detect the establishment of a *global* condition in a protocol. They are, therefore, a powerful concept and may be expensive to implement. However, they drastically simplify the description and verification of protocols with timeout operations, and suggest a unified approach for implementing timeouts using real-time clocks.

The idea now is that a process starts its timer when it becomes a candidate. When receiving a response of the leader on its initial $I(my\_id)$ message the timer plays no role and the process progresses as in the first protocol. In absence of a response of a leader, the candidate goes to the leader state at the occurrence of a timeout. Thus, a timeout guard must be disabled in case a leader is present. This leader process might be the leader at the start of the timer, but might also be a 'fresh' one. Therefore, a timeout guard is defined to be true (the timer expires) only when a process has received and processed all responses to its message sent at starting the timer. This timeout mechanism is usually called *non-premature*. A precise characterization of the timeout mechanism is given in section 4. We thus obtain the protocol as depicted in Figure 2(a).

Recall that the reason for introducing two different message types to exchange identities in Protocol 1 was to avoid the violation of P2. We observe that – due to the timeout mechanism – this problem does no longer occur. Therefore, there is no objection against replacing the response messages by I-messages. This results in the protocol as depicted in Figure 2(b). As a consequence, candidates can now be forced to become defeated by receiving messages from other candidates. In Protocol 1 a candidate only reacts to messages sent by the leader.

Some significant simplifications to the latter protocol can be made. Observe that there are two possible transitions from the candidate state to the leader state, one of which may take place when no leader is present (labelled with a timeout guard). The other transition is enabled on receipt of an $I(my\_id)$ message which is only sent when a leader capitulates. It is not hard to see that the protocol's correctness is not affected by the removal of this message transmission. So, in that case a leader moves without any notification to the defeated state on receipt of a larger id than its id. This implies that one of the transitions to the leader state will never be enabled and, hence, may safely be eliminated. Thus we obtain the protocol depicted in Figure 3, referred to as "Protocol 2".

3.4 A fault-tolerant LE protocol

In this section we drop the assumption of perfect processes and revise our earlier designs by considering processes that cease participation without notifying other processes. After halting a process does not behave maliciously. This kind of failures is known as *crash faults* (see e.g. [16]). Crashed processes may recover and (re-)join at any time. It is assumed that recovered processes restart in the start state. This should not be confused with "self-stabilizing" systems [15, 38] where processes may recover in any state. The number of times a process can crash or recover during an election is unlimited. A process cannot crash during the execution of an atomic event.

Recall the requirements as specified in section 3.2.1. Since the assumptions about process behaviour are now strongly modified it needs to be checked whether the initial requirements are still realistic. For instance, it is rather unrealistic to require P2 bearing in mind that all processes may crash eventually. We, therefore, first adapt the requirements to the new context.

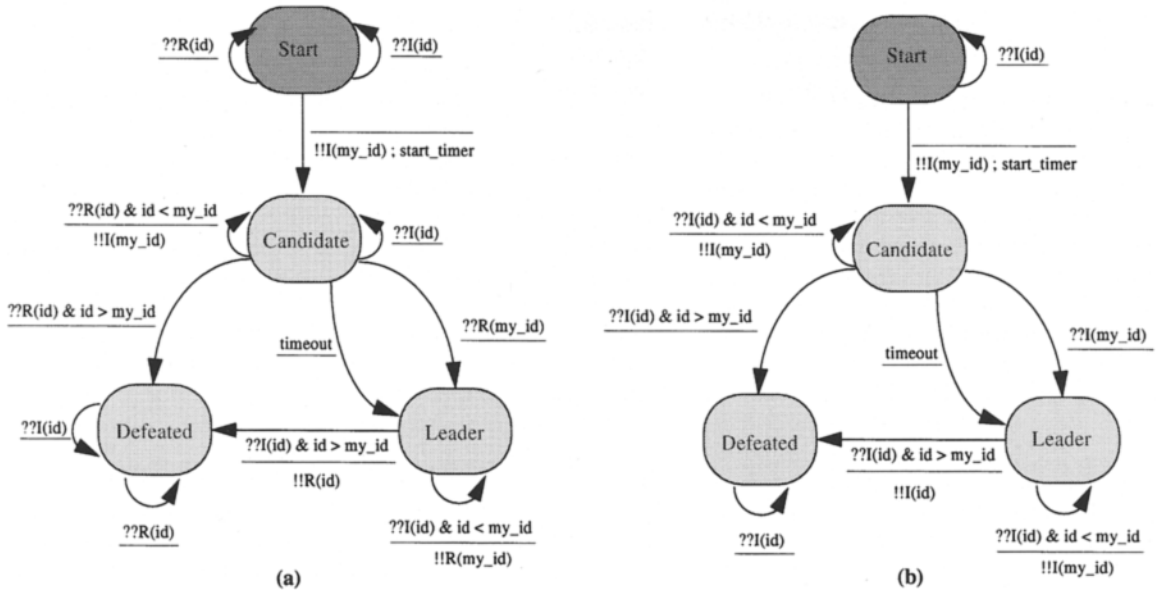

Fig. 2. Finite state machine diagrams of two derivatives of Protocol 1

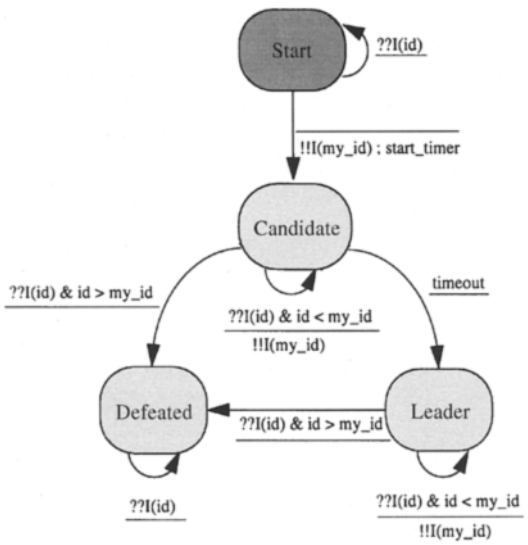

Fig. 3. Finite state machine diagram of Protocol 2

3.4.1 Requirements revisited

It is still essential that at any moment of time there is at most one leader:

Q1 $\forall i, j \neq i : leader(i) \Rightarrow \neg leader(j)$.

In order to distinguish between our initial requirements P1 through P4 and the new ones we label new requirements with Q. Again, quantifications, by default, implicitly range over the processes actually participating at that moment. This includes crashed processes.

P2 claims that there (always) will be infinitely often a leader. As stated above, it is unrealistic to demand P2 since potentially all processes may fail. We therefore claim that always there will be infinitely often a leader if there exists a process at some time which will definitely not crash from then on and for which all better processes have (and remain) crashed. Predicate $dead(i)$ indicates the fact that process i has crashed. Formally,

Q2 $\Diamond(\exists i \in Id : \Box(\neg dead(i) \wedge (\forall j > i : dead(j))))$
$\Rightarrow \Box\Diamond(\exists k : leader(k))$.

$\Box\Diamond P$ for predicate P states that at any point in the execution it is true that eventually P will hold.

Quite evidently, a crashed process cannot act as a leader process (and vice versa).

Q3 $\forall i : \neg(leader(i) \wedge dead(i))$.

The next requirement addresses the question in what circumstances a leader capitulates. Well, a leader should be the process with the highest identity among all living participating processes. This implies that a leader should capitulate as soon as there is some other (living) process which is an improvement. However, when this better process crashes the above claim is too strong. We, therefore, require

Q4 $\forall i, j > i : leader(i) \wedge \neg dead(j)$
$\Rightarrow \neg leader(i) \vee \Diamond dead(j)$.

When a leader capitulates this may be caused by either the crash of this process or the fact that there was a better (living) process. Formally,

Q5 $\forall i : \mathscr{J}\neg leader(i) \Rightarrow dead(i) \vee \blacklozenge(\exists i > i : \neg dead(j))$.

Both Q4 and Q5 refer to the catipulation of a leader. It remains to require something about the succession of leaders. Previously we required that leaders must be succeeded by better ones. This claim is still valid. However, it needs a more careful formulation, since, it is invalid in case, for instance, a leader capitulates by crashing. It, therefore, seems reasonable to require

Q6 $\forall i \in \mathscr{P}, j \in Id : leader(i) \wedge \Diamond leader(j)$
$\wedge \neg dead(i) \mathscr{U} leader(j) \Rightarrow i \leqq j$.

Informally formulated: given some leader process, i say, its successor, process j, is not less qualified than i provided that i does not crash in between the leaderships of i and j. Q6 thus claims nothing about the relation between a leader and its successor when the leader crashes in the meanwhile. Furthermore, crashes of other processes do not have any influence. Notice that a leader may be succeeded by itself (i.e. i and j are not necessarily distinct) as it may capitulate due to the presence of a better candidate that crashes before becoming a leader.

The consistency of Q1–Q6 with respect to P1–P4 is discussed below. In case processes do not crash Q4 boils down to P3, that is, under the assumption $\Box(\forall i \in Id: \neg dead(i))$ Q4 simply reduces to P3. In a similar way Q2 reduces to P2 as shown below

$$\Diamond(\exists i \in Id: \Box(\neg dead(i) \wedge (\forall j > i: dead(j))))$$
$$\Rightarrow \Box\Diamond(\exists k: leader(k))$$
$$= \{\Box(\forall i \in Id: \neg dead(i))\}$$
$$\Diamond(\exists i \in Id: \Box(\forall j > i: false))$$
$$\Rightarrow \Box\Diamond(\exists k: leader(k))$$
$$= \{\ \}$$
$$\Diamond(\exists i \in Id: i = i_{max} \wedge (\forall j: j \leqq i_{max}) \wedge \Box(\forall j > i: false))$$
$$\Rightarrow \Box\Diamond(\exists k: leader(k))$$
$$= \{(\forall j > i_{max}: false) = true\}$$
$$\Diamond(\exists i \in Id: i = i_{max} \wedge (\forall j: j \leqq i_{max}))$$
$$\Rightarrow \Box\Diamond(\exists k: leader(k))$$
$$= \{Id \text{ is finite, so } i_{max} \text{ exists}\}$$
$$\Diamond true \Rightarrow \Box\Diamond(\exists k: leader(k))$$
$$= \{\text{calculus}\}$$
$$\Diamond(\exists k: leader(k))$$

where in the last step we use the fact that all formulas should be interpreted to hold over all states (see section 3.1.3.).

The relationship between Q6 and P4 is more subtle. When processes do not crash we derive for Q6:

$$\forall i \in \mathscr{P}, j \in Id: leader(i) \wedge \Diamond leader(j)$$
$$\wedge \neg dead(i) \mathscr{U} leader(j) \Rightarrow i \leqq j$$
$$= \{\Box(\forall i \in Id: \neg dead(i))\}$$
$$\forall i \in \mathscr{P}, j \in Id: leader(i) \wedge \Diamond leader(j)$$
$$\wedge true \mathscr{U} leader(j) \Rightarrow i \leqq j$$
$$= \{true \mathscr{U} leader(j) = \Diamond leader(j)\}$$
$$\forall i \in \mathscr{P}, j \in Id: leader(i) \wedge \Diamond leader(j) \Rightarrow i \leqq j\ .$$

In the context of the previous protocols this resulting requirement, however, allows a leader to capitulate (in presence of a better candidate, cf. P3), become a leader again, capitulate (there is still a better candidate), and so on, in a repetitive way. In case processes do not crash this is – in our opinion – not desirable as no real progress is made: when a leader catipulates due to the presence of a better candidate one expects that at some time a new (and better) leader emerges. Therefore, requirement P4 was introduced. For Protocol 3 this situation is different as each process, including candidates, may crash spontaneously. Thus a leader may capitulate because a better candidate is noticed, but before this candidate becomes a leader it crashes. Then it must be allowed that the capitulated leader becomes a leader again. This justifies Q6.

3.4.2 Design of a fault-tolerant protocol

We take the previous protocol as a starting point for our design of a fault-tolerant LE protocol. The crucial point now is that in absence of a leader after it crashes, a defeated process might be a valid successor.

So as to involve defeated processes in the election we consider two cases. First, to avoid a candidate to become a leader in case a leader crashed and a better defeated process is present, defeated processes become a candidate on receipt of an I-message with a smaller id than their own id – thus joining the competition about the leadership and thus avoiding violation of Q4. Other I-messages are still ignored when being defeated. It should be observed that this does not suffice in case a leader crashes, at least one defeated process is present (that will never crash), and no candidate will ever appear. In this scenario no leader will ever be elected, although there is some process that will never crash. This violates Q2. Therefore, we should have a mechanism via which defeated processes will rejoin the election in absence of a leader. Using the fact that the underlying broadcast facility is reliable, several techniques can be applied to accomplish this.[2] Here we abstract from a specific technique and model this by adding a transition labelled with an absent guard from defeated to the candidate state, such that a defeated process may (re-)join the election spontaneously by identifying itself and starting its timer.[3] We model the fact that processes may crash at arbitrary times by a possible transition from each possible state to a new state, named *dead* state. We denote these transitions by dotted arrows. The difference between transitions represented by dotted, respectively solid, arrows should be interpreted as follows. In case of a dotted arrow the transition is always possible (and hence can be non deterministically chosen), but not necessary (that is, it can be ignored indefinitely). On the other hand, a solid arrow represents a necessary transition, that is, a transition that eventually has to be taken whenever it is continuously

[2] For instance, a leader may transmit on a regular basis "I am here" messages and in absence of such messages a timeout could expire in a defeated process, thus forcing it to become starting (or candidate). Another possibility would be to let a defeated process regularly check whether a leader is present (see e.g. [20]).

[3] It should be noted that we now have two transitions with equivalent actions, one of which has a true guard from the defeated state to the candidate state. These transitions cannot be combined into a single transition with a true guard as it would then be no longer guaranteed that this transition is made on receipt of an I-message with an identity larger than that of the recipient: a process may then perform the transition whenever it likes

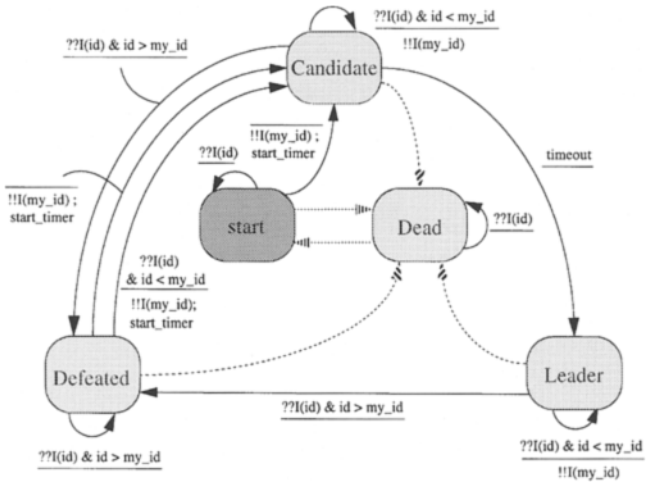

Fig. 4. Finite state machine diagram of Protocol 3

enabled. Representing crash transitions by solid arrows would imply that all processes crash eventually which is rather unnatural. The dotted arrows and solid arrows are similar to the modal relations $\rightarrow\hat{\Diamond}$, respectively $\rightarrow\hat{\Box}$ of modal transition systems (see e.g. [29]).

Similarly, the fact that processes may recover spontaneously after crashing is modeled by a (dotted) transition from the dead to the start state. This yields the protocol depicted in Figure 4, called "Protocol 3". For the sake of brevity, a transition label is omitted when both its associated guard and its set of actions are absent. In order to prevent unspecified receptions (and thus a system deadlock), a process in the Dead state is able to consume messages from its buffer (see Theorem 4.5).

4 Verification by temporal logic

In the previous section we informally motivated our design decisions. In this section we formally prove that the last protocol satisfies its requirements Q1 through Q6. We furthermore, prove that for this protocol unspecified receptions cannot occur. We do not intend to give a completely formalized proof. Such a proof is well possible, but however, requires a formalization of the assumptions, a transformation of the protocol to our proof formalism (temporal logic), and so on, which would make the proofs too much involved. We, therefore, confine ourselves to presenting only the main ideas of the proof. The correctness of the first two protocols can be proven in a similar way as the third protocol. These verifications are reported in [8].

4.1 Notations and conventions

We use the following notations and conventions. The fact of being a leader, that is, *leader*(i), is identified with the fact that process i is in the leader state. To distinguish between the conceptual notion of being a leader and the internal protocol states, L_i is used to denote that i is in the leader state of the protocol. Similarly, predicates S_i, C_i, D_i, and F_i denote that process i is in the start, candidate, dead or defeated state, respectively. The local buffer of process i is symbolized by Q_i. Assertion $\text{SEND}_i(m(p_1, \ldots, p_n))$ is true (in some state of the state sequence) only when process i executes $!!m(p_1, \ldots, p_n)$ at leaving that state. Similarly, assertion $\text{RCV}_i(m(p_1, \ldots, p_n))$ is true if and only if guard $??m(p_1, \ldots, p_n)$ evaluates to true and the corresponding transition is taken. For some protocol state guard TO_i for process i evaluates to true whenever i's timeout occurs and the corresponding transition is taken.

4.2 Reliable broadcast communication

We first formally define some relevant assumptions about the broadcast mechanism which we use as underlying communication facility. Let m, m^p, and m^q be unique messages, that is, both their originator and moment of origination are unique. (It has been shown in [27] that messages need to be uniquely identifiable so as to specify communication mechanisms in temporal logic by axioms like those below. In this paper we accomplish this by message numbering by the sender. As from the context the dependence on the identity of the sender is explicit, this dependence is often omitted. In the sequel we use p, q as numbers of messages.)

The broadcasting mechanism we use is formally specified as follows.

Assumption 4.1. $\forall i: \text{SEND}_i(m) \Rightarrow \bigcirc(\forall j \neq i: m \in Q_j)$.

Assumption 4.2. $\forall i: m \in Q_i \Rightarrow \blacklozenge(\exists j \neq i: \text{SEND}_j(m))$.

Assumption 4.3. $\forall i, j: \text{SEND}_i(m^p) \wedge \Diamond \text{SEND}_j(m^q) \Rightarrow (\forall k \neq i, j: \Diamond(m^p \in Q_k \wedge \Diamond m^q \in Q_k))$.

Assumption 4.4. $\forall i, j \neq i: \text{SEND}_i(m^p) \Rightarrow \neg \text{SEND}_j(m^q)$.

Assumption 4.1 states that messages that are sent are received instantaneously by all processes, except the sending process. Note that it also implies that messages are not lost by the communication network. Assumption 4.2 phrases that messages are not spontaneously generated by the network, and assumption 4.3 expresses that the network is order-preserving. Assumption 4.4 says that at most one process may send over the network at a time.

Observe that it immediately follows from axiom 4.2 that a process does not receive its own transmitted messages. That is, for all unique messages m

(1) $\forall i: m \in Q_i \Rightarrow \blacksquare \neg \text{SEND}_i(m)$.

The relation between buffering of messages and the actual processing of messages is given by the following theorem which states that queued messages will eventually be processed. This is, of course, not a property of the broadcast mechanism, but a desirable property of the protocol. We provide the proof for Protocol 3. Note that the theorem implies that no unspecified receptions can occur.

Theorem 4.5. $\forall i: m \in Q_i \Rightarrow \Diamond \text{RCV}_i(m)$.

Proof. As there is only one message type involved, as corresponding transitions exist for this message type (for all possible parameter values) in all states, and as processes

do not receive their own transmitted messages, it is evident that all messages can be processed in each state. The fact that a message will eventually be processed follows from the (weak) fairness assumption saying that a transition that is continuously enabled will eventually be taken. □

4.3 The timeout mechanism

The semantics of the timeout mechanism were informally defined in section 3.3. In order to facilitate a formal proof we formalize this semantics. This formalization is essential so as to prove the invariance of Q1 through Q6.

We characterize in general terms, that is without reference to the protocol, a 'non-premature' timeout in a broadcast network. A timer is started at the transition of message m, say. This message has to be received (and processed) by all its recipients before the timer may expire. Formally, for all unique messages m^p

Assumption 4.6. $\forall i{:}\text{SEND}_i(m^p) \Rightarrow \neg \text{TO}_i^p \mathcal{W} (\forall j \neq i{:} \blacklozenge \text{RCV}_j(m^p))$.

Strictly speaking, the timeout assertion is associated to m^p, and as m^p is unique, the occurrence of the timeout is considered to be unique. When necessary this dependence on m^p is explicitly indicated by referring to the number of p of m. As, in general, it is not guaranteed that each process is capable of processing a message of type m in some state, we use the $\mathcal{W}$ operator instead of the $\mathcal{U}$ operator. In absence of unspecified receptions – as in the presented protocols – we could equally well use the $\mathcal{U}$ operator. Now, however, a timeout may be enabled without forcing the originator of m^p to receive and process all replies to m^p. Let $r_{m^p,j}$ be a reply to m^p transmitted by process j. We additionally require

Assumption 4.7. $\forall i{:}\text{TO}_i^p \Rightarrow (\forall j \neq i{:} r_{m^p,j} \notin Q_i)$.

Here it should be mentioned that processing a message and sending a reply to this message is considered to constitute an atomic event.[4] For the protocol at hand we should substitute $I^p(i)$ and $I^q(j)$ $(i < j)$ for m^p and $r_{m^p,j}$, respectively in assumptions 4.6 and 4.7.

The formal semantics of a non-premature timeout in broadcasting networks is now defined by assumptions 4.6 and 4.7. Summarizing, according to assumption 4.6 all processes (except the sender) receive m, process this message and, if appropriate, send a reply. These replies are forced to be received and processed by the originator of m as phrased by assumption 4.7.

4.4 Timeout properties

In the previous section we characterized the non-premature timeout in a rather general context. For the protocol at hand we have some properties which hold for the timeout mechanism. These properties are directly derived from the protocol specifications. As they are frequently used in the verification we treat them separately.

Due to the intrinsic recursive behaviour of Protocol 3, predicates must be defined carefully. When stating, for instance, $C_i \wedge \Diamond \text{TO}_i^p$ there is no formal relation between the two conjuncts: process i may be a candidate for a while, leave this state and become a candidate again and then leaving this state on TO_i^p. Stating C_i referring to the first period in the candidate state has no relation at all with TO_i^p. In order to establish such a relation the idea is to refer to the $I(i)$ message on which process i has become a candidate – and which must have number p such that it corresponds with the next timeout of i to occur.[5] Note that it is possible to refer to the $I(i)$ message on which i has become a candidate in the temporal logic formalism we use. However, we also want to refer to the receipt of this message by some other process. This is not possible in temporal logic, but is rather straightforward when introducing explicit labelling of I messages.

We have the following timeout properties. The first one states that a process can only perform a timeout when being a candidate.

Property 4.8. $\forall i{:}\text{TO}_i \Rightarrow C_i$.

Once a process enters the candidate state by transmission of $I^p(i)$ and the corresponding timeout occurs eventually, that is $\Diamond \text{TO}_i^p$, it does not leave the candidate state until this timeout occurs. Note that this also implies that the process does not crash in between the transmission and the corresponding timeout.

Property 4.9. $\forall i{:}\text{SEND}_i(I^p(i)) \wedge \bigcirc C_i \wedge \Diamond \text{TO}_i^p \Rightarrow C_i \mathcal{U} \text{TO}_i^p$.

Candidate i becomes defeated on receipt of $I(j)$ with $i < j$:

Property 4.10. $\forall i{:} C_i \wedge (\exists j > i{:}\text{RCV}_i(I(j))) \Rightarrow \bigcirc F_i$.

We now state the following lemma which phrases that no $I(j)$ message is received by process i $(i < j)$ after entering the candidate state until its timeout occurs (provided its timeout occurs at some time).

Lemma 4.11. $\forall i{:}\text{SEND}_i(I^p(i)) \wedge \bigcirc C_i \wedge \Diamond \text{TO}_i^p \Rightarrow \neg (\exists j > i{:} \text{RCV}_i(I(j))) \mathcal{U} \text{TO}_i^p$.

Proof. By contradiction. Assume C_i and $\Diamond \text{TO}_i^p$. It follows directly from properties 4.9 and 4.10 that candidate i becomes defeated on receipt of $I(j)$, $j > i$. Consequently, no timeout will appear. This contradicts with the assumption. □

One can now infer from the timeout semantics and the above lemma that process j can prevent the occurrence of the timeout of another process, i say, by transmitting $I(j)$ with $i < j$, as reply to the receipt of $I^p(i)$.

4.5 Protocol verification

We prove the requirements Q1 through Q6 one by one. The first proof obligation is Q1 and is the hardest to prove. Therefore, we divide the verification of Q1 into some parts

[4] This implies that a process must reply immediately on processing of a message and is not allowed to wait arbitrarily long with replying. It can be verified that the presented protocols conform to this principle

[5] We remark that another possibility would be to equip the C_i predicates with a number like the TO_i^p predicates and let the relationship with the $I^p(i)$ message on which i has become a candidate implicit. For the sake of clarity we prefer to give the explicit relation

and give some lemmas that are used later on in proving this requirement. Let CLF_j denote that process j is candidate, leader or defeated, and let DS_j denote that it is either dead or start.

The first lemma states that if process j is candidate, leader, or defeated on processing of message $I^p(i)$ it can neither be a leader nor perform its timeout at the occurrence of TO_i^p. That is,

Lemma 4.12. $\forall i,j: CLF_j \wedge \text{RCV}_j(I^p(i)) \wedge \Diamond \text{TO}_i^p \Rightarrow \Box(\text{TO}_i^p \Rightarrow \neg L_j \wedge \neg \text{TO}_j)$.

Proof. Let j be a process for which CLF_j holds. From the protocol we infer that on receipt of $I(i)$, either j becomes defeated in case $i > j$, or replies with $I(j)$ otherwise.

(2) $\forall i,j: CLF_j \wedge \text{RCV}_j(I(i)) \Rightarrow (j < i \Rightarrow \bigcirc F_j)$

$\wedge\ (j > i \Rightarrow \text{SEND}_j(I(j)))$.

Process j can prevent the occurrence of TO_i^p $(i < j)$ by transmitting $I(j)$ on processing $I^p(i)$. This follows from the timeout properties. Together with (2) this results in

(3) $\forall i,j: CLF_j \wedge \text{RCV}_j(I^p(i)) \wedge \Diamond \text{TO}_i^p \Rightarrow j < i$.

We thus concentrate, given that i performs TO_i^p once, on the case $j < i$. According to (2) process j becomes defeated on processing $I^p(i)$. It can only become a leader by transmitting $I^q(j)$ on becoming a candidate.

(4) $\forall j: C_j \Rightarrow \blacklozenge \text{SEND}_j(I(j))$.

As process i is still being a candidate, according to property 4.9, j is not able to become a leader before i is becoming a leader – j has to wait for i's reply on $I^q(j)$ (see timeout semantics) and as $j < i$ process i will reply on $I^q(j)$ (see (2)) thus preventing j to become a leader. In the above reasoning we only have considered perfect processes, i.e. processes that do not crash. However, when considering the crash of process j $(i > j)$ it can be deduced in a similar way that after reviving j cannot become a leader before i is becoming a leader. Note that due to property 4.9 process i does not crash before becoming a leader. So, crashes of i do not have to be taken into account. □

Lemma 4.13. $\forall i,j: DS_j \wedge \text{RCV}_j(I^p(i)) \wedge DS_j \mathcal{U} \text{TO}_i^p \Rightarrow \Box(\text{TO}_i^p \Rightarrow DS_j)$.

Proof. Straightforward. □

Lemma 4.14. $\forall i,j: DS_j \wedge \text{RCV}_j(I^p(i)) \wedge \neg(DS_j \mathcal{U} \text{TO}_i^p) \Rightarrow \Box(\text{TO}_i^p \Rightarrow \neg L_j \wedge \neg \text{TO}_j)$.

Proof. Let j be a process for which DS_j holds at processing $I^p(i)$ and i be a process for which $\Diamond \text{TO}_i^p$ holds. It follows from the protocol that j ignores $I^p(i)$ as start and dead processes ignore all messages.

(5) $\forall i: S_i \wedge \text{RCV}_i(m^p) \Rightarrow \bigcirc S_i \wedge \neg \text{SEND}_i(m^q)$.

(6) $\forall i: D_i \wedge \text{RCV}_i(m^p) \Rightarrow \bigcirc D_i \wedge \neg \text{SEND}_i(m^q)$.

Consider the case that j has left the start/dead state after processing $I^p(i)$ and before i performs its timeout, that is, $\neg(DS_j \mathcal{U} \text{TO}_i^p)$. According to

(7) $\forall j: DS_j \wedge \bigcirc \neg DS_j \Rightarrow \bigcirc C_j$.

j has become a candidate and due to (4) must have sent $I(j)$ in order to do so. According to assumption 4.1 i will receive this message. As $I(j)$ is not a reply on $I^p(i)$, process i is not forced to process this message before performing its timeout. This suggests the following case analysis.

First, consider the case that i processes $I(j)$ prior to its timeout. According to lemma 4.11 this implies $i > j$, given that i will perform its timeout once. Due to (2) i replies with $I(i)$, and as j is forced to wait for this reply before becoming a leader it will not be able to perform its timeout (due to lemma 4.11). In the other case, i processes $I(j)$ after performing its timeout. But then, j cannot be a leader at the moment i performs its timeout, as it is forced, according to assumption 4.7, to wait for the reply of i.

When process j crashes in the meanwhile it can be verified using identical arguments that j cannot become a leader before i does. □

Lemma 4.15. $\forall i,j: \text{RCV}_j(I^p(i)) \wedge \Diamond \text{TO}_i^p \Rightarrow \Box(\text{TO}_i^p \Rightarrow \neg L_j \wedge \neg \text{TO}_j)$.

Proof. Follows directly from lemmas 4.12, 4.13, and 4.14. □

This lemma is the crux to the proof of the following lemma which states that when a process performs its timeout, it is the only process that does so (so, two – or more – processes cannot become a leader simultaneously), and there are no leader processes.

Lemma 4.16. $\forall i,j \neq i: \text{TO}_i \Rightarrow \neg L_j \wedge \neg \text{TO}_j$.

Proof. Let i be a process for which $\Diamond \text{TO}_i^p$ holds. According to property 4.8 a timeout can only occur being a candidate. Process i only becomes candidate after sending $I^p(i)$ (4). TO_i^p is associated with $I^p(i)$ and can be performed if and only if all processes have processed this message (assumption 4.7). The lemma follows now directly from lemma 4.15. □

Theorem 4.17. $\forall i,j \neq i: L_i \Rightarrow \neg L_j$.

Proof. From the protocol we deduce that on occurrence of a timeout a process becomes a leader immediately

(8) $\forall i: \text{TO}_i \Rightarrow \bigcirc L_i$.

In addition, after just becoming a leader the process must have performed a timeout:

(9) $\forall i: \bigcirc \blacklozenge L_i \Rightarrow \text{TO}_i$.

As the occurrence of a timeout is the only way to become a leader (immediately after that) theorem 4.17 reduces to lemma 4.16. □

Theorem 4.18. $\Diamond(\exists i \in Id: \Box(\neg D_i \wedge (\forall j > i: D_j))) \Rightarrow \Box\Diamond(\exists k: L_k)$.

Proof. Consider the process with the maximum id, i' say, for which $\Diamond\Box(\neg D_{i'} \wedge (\forall j > i': D_j))$ holds. According to the premise of Q2 this process exists. The idea of the proof is to establish that process i' will always become a leader sooner or later. That is, we prove

(10) $\Diamond L_{i'}$

from which we directly deduce Q2. The proof is as follows. Consider process i' at the moment that all better processes than i' are crashed for ever, that is, $(\forall k > i' : \Box D_k)$. Remark that – although all better processes are crashed – process i' may still have messages originating from these processes in its buffer, as processes may process buffered messages at their own pace. Now refer to the moment at which i' has processed all messages from these processes. That is, assume

(11) $\mathscr{I} \Rightarrow \Diamond(\forall k > i' : m_k \notin Q_{i'} \wedge \Box D_k)$,

where m_k denotes a message originating from process k. Distinguish between two cases: i' is already a leader, or it is not. Consider the first case, so $L_{i'}$ holds. From the protocol description we immediately infer that leaders can only capitulate by either crashing or receiving an $I(k)$ message with k larger than their own identity. Formally,

(12) $\forall i : \mathscr{I} \neg L_i \Rightarrow D_i \vee \blacklozenge(\exists j > i : \text{RCV}_i(I(j)))$.

Given that i' does not crash there is only one possibility to capitulate, namely by receiving $I(k)$, $k > i'$. It is straightforward to observe that $I(k)$ messages are only transmitted by process k.

(13) $\forall i, k : \text{SEND}_i(I(k)) \Rightarrow i = k$.

Furthermore, crashed processes do not transmit messages.

(14) $\forall k : \text{SEND}_k(m) \Rightarrow \neg D_k$.

Using (11) and the above reasoning it can easily be deduced that it is impossible for i' to receive a message $I(k)$, $k > i'$, and consequently, it is impossible for i' to capitulate. Thus, we conclude:

(15) $L_{i'} \wedge (\forall k > i' : \Box D_k \wedge m_k \notin Q_{i'}) \Rightarrow \Box L_{i'}$.

Secondly, we consider the case that i' is not a leader. Recall (11). From the protocol we directly infer that processes that will never crash and are not leader (yet) will become a candidate once.

(16) $\forall i : \Box \neg D_i \wedge \neg L_i \Rightarrow \Diamond C_i$.

Once, process i' transmits its I-message and becomes a candidate. As there is no 'better' process that can reply – they are all crashed for ever – it follows from assumptions 4.6 and 4.7 that i' can perform its timeout and becomes a leader. Using an analogous reasoning as for the first case we conclude that i' will be a leader indefinitely. □

Theorem 4.19. $\forall i : \neg(L_i \wedge D_i)$.

Proof. Trivial. □

Theorem 4.20. $\forall i, j > i : L_i \wedge \neg D_j \Rightarrow \Diamond \neg L_i \vee \Diamond D_j$.

Proof. Assume $L_i \wedge \neg D_j \wedge j > i$. Distinguish between two cases: $\Box \neg D_j$ and $\Diamond D_j$. The interesting case is $\Box \neg D_j$. For this case the theorem reduces to

(17) $\forall i : L_i \wedge (\exists j > i : \Box \neg D_j) \Rightarrow \Diamond \neg L_i$.

Using (16) and theorem 4.17 this property holds when the following property does

(18) $\forall i : L_i \wedge (\exists j > i : C_j \wedge \Box \neg D_j) \Rightarrow \Diamond \neg L_i$.

This property is proven as follows. Assume $L_i \wedge C_i \wedge \Box \neg D_j \wedge j > i$. According to (4) j has send $I(j)$ to become a candidate. This message is processed by i after it became a leader – otherwise $I(j)$ would have prevented i of becoming a leader. If i has already capitulated $\Diamond \neg L_i$ follows directly. In case L_i holds, i capitulates on processing $I(j)$, $j > i$, according to (2). □

Theorem 4.21. $\forall i : \mathscr{I} \neg L_i \Rightarrow D_i \vee \blacklozenge(\exists j > i : \neg D_j)$.

Proof. Let i be a process for which L_i holds. According to (12) there are only two ways in which i can capitulate. In case it spontaneously crashes, we have $\mathscr{I} \neg L_i \Rightarrow D_i$. Alternatively, it capitulates on receipt of $I(j)$ with $j > i$. Due to (13) $I(j)$ can only be transmitted by process j. Furthermore, crashed processes cannot transmit messages (due to (14)). Thus, we conclude

(19) $\forall i, j > i : \text{RCV}_i(I(j)) \Rightarrow \blacklozenge(\neg D_j \wedge \text{SEND}_j(I(j)))$.

This directly implies

(20) $\forall i : \mathscr{I} \neg L_i \wedge \neg D_i \Rightarrow \blacklozenge(\exists j > i : \neg D_j)$. □

Theorem 4.22. $\forall i \in \mathscr{P}, j \in Id : L_i \wedge \Diamond L_j \wedge \neg D_i \mathscr{U} L_j \Rightarrow i \leqq j$.

Proof. Let process j be the immediate successor of leader i and assume i does not crash in between the leaderships of i and j. The proof is by contradiction. Assume $i > j$. From the protocol description we immediately infer that

(21) $\forall i : CLF_i \wedge \Box \neg D_i \Rightarrow \Box CLF_i$.

So, in case a leader capitulates and does not crash it is either candidate, leader or defeated. From (3) it follows that a process cannot become a leader in presence of a better candidate, leader or defeated process. This implies that j ($j < i$) cannot become a leader when i is still in one of these states. However, according to the premise $L_i \wedge \neg D_i \mathscr{U} L_j$ and the above property, this is the case. This contradicts with j being the successor of i. □

5 Complexity analysis of the protocols

Much work has been devoted in literature on designing efficient LE protocols. In general, the following complexity measures are considered: *message complexity* (the number of messages needed to elect a leader), *time complexity* (the number of time units needed to elect a leader) and *bit complexity* (the total number of bits to elect a leader). In this section we discuss the worst case message complexity of our protocols in terms of the number of broadcast messages. For the sake of brevity we only present a detailed analysis of the complexity results of Protocol 1, for an elaborated analysis of the other protocols we refer to [8].

In our protocols all messages are broadcasted, so each message is received by all processes (except the sender). In a distributed system where processes spontaneously wake up, each process at least has to send one (initial) message to the other processes so as to identify itself, resulting in a minimum message complexity of $\Omega(N)$.

For reasons of simplicity a process identity is represented by a positive natural number.

Theorem 5.1. *The worst case message complexity of Protocol 1, for N participating processes and process i being the initial leader, is* $\frac{1}{2}N(N+1) - \frac{1}{2}i(i-3) - 2$.

Proof. Let i be the leader and all other processes be in the start state. In the worst case scenario all processes become a candidate simultaneously and send their initial I-message in increasing order (strictly speaking, the order of the last $(N-i-1)$ I-messages is irrelevant). Due to the fact that from all better processes $I(i+1)$ is processed first, process $i+1$ becomes the next leader on receiving $R(i+1)$. This message evokes an I-message from all candidates better than $i+1$. If these replies are sent with $I(i+2)$ first, process $i+2$ becomes the next leader. This scenario is repeated until ultimately process N becomes leader. In each "round" the number of candidates is reduced by one and the number of reactions on an R-message is maximal. This indicates that the above is indeed the worst case scenario.

We now focus our attention on all transitions in which messages are transmitted and determine the number of transmitted messages for the worst case scenario. In order to let all processes become a candidate $(N-1)$ messages are needed. Candidate $j, j > i$, receives $(j-2)$ $R(k)$ messages with $k < j$. Consequently, it replies with $(j-2)$ $I(j)$ messages. Leader i must respond to the first $(i-1)$ I-messages with $R(i)$ before it capitulates. As subsequent leaders j capitulate immediately they do not transmit any $R(j)$ messages. Finally, in order to let readers capitulate $(N-i)$ messages are needed. Combination of these results leads to the above result. □

The message complexity can be improved significantly by the idea of *'smart' buffering*. According to this principle, messages are buffered depending on their parameter: at each moment of time a process buffer only contains the I-message (or R-message) with the largest id received so far, that is, not processed yet. Here, R-messages have priority over I-messages. In this way a buffer contains at most one message at a time. Adopting this buffering mechanism reduces the message complexity to $\mathcal{O}(N)$, independent of the initial leader:

Theorem 5.2. *With smart buffering the worst case message complexity of Protocol 1 is* $2N - 2$.

Proof. Buffering of several initial I-messages now leads to a single R-message to the process with the highest *id*, which makes this process the new leader and forces the other processes to the defeated state. The worst case scenario appears when each initial message is separately answered by an R-message. It does not matter which process is the initial leader or in which order the processes send their initial I-message. In this case $2(N-1)$ messages are needed. □

Theorem 5.3. *Protocol 2 has a worst case message complexity of* $2^N - 1$ *using 'simple' buffering and of* $2N - 1$ *using smart buffering.*

Finally, we consider Protocol 3. First we consider an election without crashing processes. We obtain the following results:

Theorem 5.4. *With perfect processes Protocol 3 has a worst case message complexity of* $2^N - 1$ *using 'simple' buffering and of* $\frac{1}{2}N(N+1)$ *using smart buffering.*

Next, we analyze the complexity in case K processes crash $(0 \leqq K < N)$. Many complex scenarios are possible, dependent on what moment during an election a process crashes. For simplicity, we assume that crashed processes do not recover and defeated processes only return spontaneously to the candidate state when a leader is actually absent. The worst case scenario occurs when K processes crash after the initial election has been completed (i.e., process N is leader and all other processes are defeated).

Theorem 5.5. *When at most* K $(0 \leqq K < N)$ *processes crash during the election the worst case message complexity of Protocol 3 is* $\frac{1}{6}K^3 - \frac{1}{2}NK^2 + (\frac{1}{2}N^2 - \frac{1}{6})K$ *using smart buffering.*

6 Conclusions

In this paper we have designed and specified a series of dynamic leader election (LE) protocols in broadcast networks, and verified a fault-tolerant protocol. From this extensive case study in protocol design, specification, and verification we make the following remarks.

We started our design by formally capturing the protocol requirements. Rather surprisingly, no such precise – and abstract – problem specification for dynamic LE currently exists in literature. When considering the protocol's correctness that is even more remarkable as a formal problem specification is indispensable for a formal verification.

Linear-time temporal logic was used so as to express the requirements and to perform the verification. The formalism turned out to be very convenient for specifying the requirements in a rather abstract way. Due to the dynamic character of processes it is not straightforward to give such a specification in, for instance, a process algebraic formalism without aiming at a particular protocol (see also [9]).

The protocols are constructed in a step-wise fashion starting from the formal requirements. The step-wise approach aids not only in the clarity and conciseness of the protocols, but also – and more important – in reasoning about them ('separation of concerns'). Due to our experience we believe that this is a feasible approach for the design of complex, dynamic communication protocols. We believe that we would not have ended up with the current concise and lucid fault-tolerant protocol without this approach.

The use of temporal logic for the specification and verification of communication protocols is well-known for a decade (see e.g. [28, 21, 40]). This case study shows – once more – that this technique combined with the state transition approach is very convenient. In fact, we have shown that these techniques are also applicable when designing a new protocol whereas most case studies focus on already existing protocols with commonly agreed requirements. Furthermore, the dynamic character of processes makes the problem considerably more complex

than traditionally verified protocols. By the use of abstract timeouts the protocol could be verified in a similar way as a protocol without timeouts.

Ideally, detailed proofs of complex protocols are required in which each step of the proof is formalized and for which informal arguments are minimized. Such detailed proofs are well possible in our framework and require a formalization of the assumptions, translation of the protocols into the proof formalism, and so on. The proofs in this paper constitute a useful stepping-stone towards such a detailed proof. Obtaining a completely formalized proof is considered to be an interesting subject for further research.

For the development and analysis of the LE protocols in this paper we also applied techniques from algebraic process theory [5] and we made use of simulation tools for process algebra [35]. This is reported in [7, 9].

In the first instance the construction of protocols was aimed at correctness with respect to the requirements and minimizing the number of transitions – rather than optimizing their efficiency. As efficiency, though, plays an important role in the field of LE protocols we analyzed the protocols' worst case message complexity, that is, the maximum number of messages needed to elect a leader. During this analysis the use of protocol simulation facilities [35] was of considerable help. With the aid of these tools it turned out that the introduction of an alternative buffering mechanism reduces the message complexity significantly.

A possible (and interesting) extension to the LE problem is to consider identities that may change during operation as opposed to fixed identities. We remark that the final, fault-tolerant protocol is also applicable in this context.

This case study shows the usefulness of manual verification for a non-trivial protocol problem and is helpful in gaining experience of how such a verification is best conducted. Application to other protocols must show how useful this information turns out to be.

Acknowledgements. The authors gratefully acknowledge Jan Bergstra (Univ. of Amsterdam & Univ. of Utrecht) for initiating and stimulating our fruitful cooperation. We are also grateful to Jan Friso Groote (Univ. of Utrecht) for his assistance during the beginning of our work. Henk Eertink (Univ. of Twente), Ruurd Kuiper (Univ. of Eindhoven), Yat Man Lau (Philips Research), and Marnix Vlot (Philips Research) are kindly acknowledged for commenting on parts of a draft version of this paper. The comments of the anonymous referees have strongly improved the presentation of the paper.

References

1. Abu-Amara HH: Fault-tolerant distributed algorithm for election in complete networks. IEEE Trans Comput 37: 449–453 (1988)
2. Afek Y, Gafni E: Time and message bounds for election in synchronous and asynchronous complete networks. SIAM J Comput 20: 376–394 (1991)
3. Attiya H: Constructing efficient election algorithms from efficient traversal algorithms. In: van Leeuwen J (ed) Distributed algorithms. Lect Notes Comput Sci, vol 312. Springer, Berlin Heidelberg New York 1987, pp 337–344
4. Attiya H, van Leeuwen J, Santoro N, Zaks S: Efficient elections in chordal ring networks. Algorithmica 4: 437–446 (1989)
5. Baeten JCM, Weijland WP: Process algebra. Cambridge Tracts in Theoretical Computer Science, vol 18, Cambridge University Press 1990
6. von Bochmann G: Finite state description of communication protocols. Comput Networks 2: 361–372 (1978)
7. Brunekreef JJ: On modular algebraic protocol specification. PhD thesis, University of Amsterdam, The Netherlands 1995
8. Brunekreef JJ, Katoen J-P, Koymans RLC, Mauw S: Design and analysis of dynamic leader election protocols in broadcast networks. Memoranda Informatica 93-43, Department of Computer Science, University of Twente, The Netherlands (1993)
9. Brunekreef JJ, Katoen J-P, Koymans RLC, Mauw S: Algebraic specification of dynamic leader election protocols in broadcast networks. In: Ponse A, Verhoef C, van Vlijmen SFM (eds) Algebra of communicating processes. Workshops in Computing, Springer, Berlin Heidelberg New York 1994, pp 338–357
10. Budkowski S, Dembinski P: An introduction to Estelle: a specification language for distributed systems. Comput Networks ISDN Syst 14: 3–23 (1987)
11. Chang E, Roberts R: An improved algorithm for decentralized extrema-finding in circular configurations of processors. Commun ACM 22: 281–283 (1979)
12. Dolev S, Optimal time self stabilization in dynamic systems. In: Schiper A (ed) Distributed algorithms. Lect Notes Comput Sci, vol 725. Springer, Berlin Heidelberg New York 1993, pp 160–173
13. Dolev S, Israeli A, Moran S: Uniform dynamic self-stabilizing leader election. In: Toueg S et al (eds) Distributed algorithms. Lect Notes Comput Sci, vol 579. Springer, Berlin Heidelberg New York 1992, pp 167–180
14. Dolev D, Dwork C, Stockmeyer L: On the minimal synchronism needed for distributed consensus. J ACM 34: 77–97 (1987)
15. Dijkstra EW: Self-stabilizing systems in spite of distributed control. Commun ACM 17: 634–644 (1974)
16. Fisher MJ: A theoretician's view of fault-tolerant distributed computing. In: Simons B, Spector A (eds) Fault-tolerant distributed computing. Lect Notes Comput Sci, vol 448. Springer, Berlin Heidelberg New York 1991, pp 1–9
17. Gehani NH: Broadcasting sequential processes. IEEE Trans Softw Eng 10: 343–351 (1984)
18. Gotzhein R: Temporal logic and its applications – a tutorial. Comput Networks ISDN Syst 24: 203–218 (1992)
19. Gouda MG: Protocol verification made simple: a tutorial. Comput Networks ISDN Syst 25: 969–980 (1993)
20. Gusella R, Zatti S: An election algorithm for a distributed clock synchronization program. In: Proc 6th IEEE Int Conf on Distributed Computing Systems (1986), pp 364–371
21. Hailpern BT, Owicki SS: Modular verification of computer communication protocols. IEEE Trans Commun 31: 56–68 (1983)
22. Hoare CAR: Communicating sequential processes. Prentice-Hall, Englewood Cliffs 1985
23. Itai A, Kutten S, Wolfstahl Y, Zaks S: Optimal distributed t-resilient election in complete networks. IEEE Trans Softw Eng 16: 415–420 (1990)
24. King C-T, Gendreau TB, Ni LM: Reliable election in broadcast networks. J Parallel Distrib Comput 7: 521–540 (1989)
25. Korach E, Kutten S, Moran S: A modular technique for the design of efficient distributed leader finding algorithms. ACM Trans Prog Lang Syst 12: 84–101 (1990)
26. Korach E, Moran S, Zaks S: Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In: Proc ACM Symp Principles Distributed Comput (1984), pp 199–207
27. Koymans RLC: Specifying message passing systems requires extending temporal logic. In: Banieqbal B et al (eds) Proc Colloquium on Temporal Logic and Specification. Lect Notes Comput Sci, vol 398. Springer, Berlin Heidelberg New York 1989, pp 213–223
28. Lamport L: Specifying concurrent program modules. ACM Trans Prog Lang Syst 5: 190–222 (1983)
29. Larsen KG, Thomsen B: A modal process logic. In: Proc IEEE Symposium on Logic in Computer Science (1988), pp 203–210
30. van Leeuwen J, Tan RB: An improved upperbound for distributed election in bidirectional rings of processors. Distrib Comput 2: 149–160 (1987)

31. LeLann, G: Distributed systems – towards a formal approach. In: Gilchrist B (ed) Information Processing (vol. 77) (IFIP). North-Holland, Amsterdam 1977, pp 155–160
32. Loui MC, Matsushita TA, West DB: Election in a complete network with a sense of direction. Inf Process Lett 22: 185–187 (1986) (correction in Inf Process Letters 28: 327 (1988))
33. Manna Z, Pnueli A: The temporal logic of reactive and concurrent systems – Specification. Springer, Berlin Heidelberg New York 1992
34. Masuzawa T, Nishikawa N, Hagihara K. Tokura N: Optimal fault-tolerant distributed algorithms for election in complete networks with a global sense of direction. In: Bermond J-C, Raynal M (eds) Distributed algorithms. Lect Notes Comput Sci. vol 392, Springer, Berlin Heidelberg New York 1989, pp 171–182
35. Mauw S, Veltink G: A process specification formalism. Fund Inf VIII: 85–139 (1990)
36. Melliar-Smith PM, Moser LE, Agrawala V: Broadcast protocols for distributed systems. IEEE Trans Parallel Distrib Syst 1: 17–25 (1990)
37. Peterson GL: An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. ACM Trans Program Lang Syst 4: 758–762 (1982)
38. Schneider M: Self-stabilization. ACM Comput Surv 25: 45–67 (1993)
39. Schneider FB, Gries D, Schlichting RD: Fault-tolerant broadcasts. Sci Comput Program 4: 1–16 (1984)
40. Shasha DE, Pnueli A, Ewald W: Temporal verification of carrier-sense local area network protocols. In: Proc ACM Symposium on Principles of Programming Languages (1984), pp 54–65
41. Shrira L, Goldreich O: Electing a leader in a ring with link failures. Acta Inf 24: 79–91 (1989)
42. Singh G: Efficient distributed algorithms for leader election in complete networks. In: Proc 11th IEEE Int Conf on Distributed Computing Systems (1991), pp 472–479

Jacob Brunekreef is a lecturer in computer science at the University of Amsterdam. He received his M.S. in electronics in 1975 from the University of Twente and his Ph.D. in computer science in 1995 from the University of Amsterdam. His research focuses on formal specification and validation of concurrent systems like communication protocols, industrial architectures, etc.

Joost-Pieter Katoen received his M.S. in computer science from the University of Twente in 1987, and is currently researcher at the computer science department of the same university. At the Eindhoven University of Technology he finished a two-year postgraduate course in 1989 on information and communication technology. From 1990 till 1992 he was a research scientist at Philips Research Laboratories Eindhoven. His main research interests are communication protocols, especially formalisms for their specification, verification and design, and performance evaluation of distributed systems.

Ron Koymans is a research scientist at Philips Research Laboratories Eindhoven since 1989. He received his M.S. in mathematics and computer science from the University of Utrecht in 1982. From 1983 till 1989 he was a researcher, the first two years at the University of Nijmegen and the next four years at the Eindhoven University of Technology where he received his Ph.D. in computer science in 1989. His current research interests include modal and temporal logics, foundations of real-time distributed computing, protocol analysis and design, and automatic test generation.

Sjouke Mauw is a lecturer at the Eindhoven University of Technology, The Netherlands. His research focuses on formal specification and verification of concurrent systems. He designed the specification language PSF, which is based on process algebra and algebraic specifications. Being associate rapporteur for the International Telecommunication Union (ITU), he is responsible for the formalization of Message Sequence Charts. This is a graphical specification language recommended by the ITU. He received his M.S. in mathematics in 1985 and Ph.D. in computer science in 1991, both from the University of Amsterdam.