Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks

Jacob Brunekreef
Programming Research Group
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
jacob@fwi.uva.nl

Joost-Pieter Katoen
Dept. of Computing Science
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
katoen@cs.utwente.nl

Ron Koymans
Philips Research Laboratories
P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands
koymans@prl.philips.nl

Sjouke Mauw
Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
sjouke@win.tue.nl

September 8, 1993

Abstract

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 system. As the problem at hand is considerably complex we adopt a step-wise refinement design method starting from a simple leader election protocol. In a first refinement a symmetric solution is obtained and eventually a fault-tolerant protocol is constructed. This gives rise to three protocols. The worst case message complexity of all protocols is analyzed.

A formal approach to the 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 protocols satisfy their requirements. Furthermore, the protocols are specified in more detail in the process algebra formalism ACP.

Keywords & Phrases: communication protocols, finite-state machines, leader election, protocol specification and verification, temporal logic, process algebra.

1980 Mathematics Subject Classification (1985 revision): 68Q20, 68Q25, 68Q60.

CR Categories (1991 version): D.1.3, D.2.4, F.2.2.

Contents

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 co-ordination in a distributed database system, clock distribution, token regeneration after token loss, and so forth. Usually many processes in the system are capable to offer such a functionality. However, at any time only one process is allowed to actually 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 which is best according to some suitable criteria to perform that function.

In this paper we consider a distributed leader election (LE) protocol which elects the most favourable process (relative to some criteria explained later) as leader. 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 obtained for such problems. For instance, in [DDS87] a number of orthogonal characteristics are 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.

Due to the complexity of the design of a fault-tolerant LE protocol a *step-wise refinement* approach is adopted. 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.

As efficiency plays an important role in the design of leader election protocols a complexity analysis is given for each protocol presented in this paper. We focus our analysis on the worst case message complexity which indicates the maximum number of messages needed to elect a leader. For N participating processes the message complexity of our initial protocol is $\mathcal{O}(N^2)$, 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 $\mathcal{O}(N^2)$, respectively, when no crashing processes are considered.

Existing designs 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. Such a verification is carried out for all presented protocols.

The protocols are also specified in the process algebra formalism ACP (Algebra of Communicating Processes). Both the separate components (protocol processes, buffers, the communication medium) and the parallel composition of these components are specified, giving a complete formal specification of the whole distributed behaviour of the protocols. Some aspects of a formal verification of the protocols within the process algebra framework are discussed, but a complete verification of the protocols in ACP lies beyond the scope of this paper. A validation of the protocols is achieved by simulation runs of the specifications in the executable formalism PSF (Process Specification Formalism), which is close to ACP.

The paper is further organized as follows. In section 2 the relation to existing work is presented. The requirements specification, design and complexity analysis 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 all protocols from section 3 satisfy the requirements. An introduction to ACP and a specification of the protocols in ACP is given in section 5. Verification and validation in ACP of the protocols is discussed in section 6. Finally, in section 7 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 algorithms
The problem of leader election was originally coined by [LeL77] in the late seventies and various LE protocols have been developed since then. A broad range of solutions exists varying in network topology (ring [LeL77, CR79, Pet82], mesh, complete network [KMZ84, AG91, Sin91], and so on), communication mechanism (asynchronous, synchronous), available topology information at processes [LMW86, AvLSZ89], and so forth. A possible straightforward solution to a broadcast network is to superimpose a topology —like a ring— on it and to adopt a well-known solution for this topology. However, 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. A few LE protocols are known that tolerate either communication link failures (see e.g. [AA88, SG87]) or process failures [GZ86, IKWZ90, MNHT89, DIM93]. In [GZ86] the LE problem with a similar failure model and using broadcast communication is considered, however, no ordering between processes is considered. [IKWZ90] and [MNHT89] 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 [DIM93] a LE protocol is constructed which tolerates transient process failures. This protocol belongs to the category of self-stabilizing protocols [Dij74]. This protocol, however, assumes a complete network topology and does not require identities to be distinct.

Complexity Results
LE protocols vary in complexity. Early protocols for a ring network (as given in [CR79, LeL77]) have a worst case message complexity of $\mathcal{O}(N^2)$ and a worst case time complexity of $\mathcal{O}(N)$, N being the number of participants in the election. Later on these results have been improved (see e.g. [Pet82, vLT87]) to protocols with a message complexity of $\mathcal{O}(N \log N)$ and a time complexity of $\mathcal{O}(\log N)$. For a complete network LE protocols have been designed with a worst case message complexity of $\mathcal{O}(N \log N)$ and a worst case time complexity of $\mathcal{O}(N)$, see [AG91, Att87, KKM85, LMW86]. In [Sin91] 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, $\log N \leq k \leq N$.

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 [MP92] and state-transition diagrams has been applied for a number of other protocols (see e.g.[Lam83, HO83, SPE84]). 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.

Specification and Validation in Process Algebra
Many simple existing communication protocols have been specified and verified in ACP, see [Bae90] for examples. Such verifications imply many algebraic computations on process expressions, showing that the specified protocol has the required (external) behaviour. However, more complex protocols (like the leader election protocols in this paper) are too large for manual algebraic verification. These protocols can be validated by simulation runs of their behaviour. To this extent a protocol is translated to the executable formalism PSF[MV90], which is strongly related to ACP. See [MV93] for examples of protocol specification using PSF.

3 Design and Complexity Analysis of LE Protocols

3.1 Introduction

3.1.1 Communication

Processes communicate with each other by exchanging messages via a broadcast network like Ethernet [MB76]. A broadcast message sent by some process p is received instantaneously by all processes except p itself. 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 (so-called buffered broadcast [Geh84]). 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 network and simply assume the existence of such a protocol (see e.g. [SGS84]). 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 [Hoa85], Estelle [BD87], 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 [Geh84]).

3.1.2 Protocol Description Language

We denote our protocol by a Finite State Machine (FSM) diagram [vB78], 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 constitute 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 $!!m(\ldots)$ 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 $??m(\ldots)$ by a process delays that process until a message of type m is delivered. Messages sent by $!!m(\ldots)$ can be received only by $??m(\ldots)$, 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 $(0<i\leq n)$ 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 received 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' [MP92]).

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 [MP92]). An extensive introduction to the use of temporal logic for communication protocols can be found in [Got92].

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'), $\odot$ ('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). $\odot\varphi$ 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* equivales $\neg$ *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 that model. For s=$(s_0, s_1, \ldots, s_n, \ldots)$ and φ, ψ arbitrary temporal formulas, $\models$ is defined as follows:

$$
\begin{array}{lll}
s, V, n \models P & \text{iff} & s_n \in V(P) \;\; \text{for each predicate } P \\
s, V, n \models \neg\varphi & \text{iff} & s, V, n \not\models \varphi \\
s, V, n \models \varphi \wedge \psi & \text{iff} & s, V, n \models \varphi \text{ and } s, V, n \models \psi \\
s, V, n \models \varphi\, \mathcal{U}\, \psi & \text{iff} & \text{there exists } m > n \text{ such that } \; s, V, m \models \psi \;\text{ and} \\
 & & s, V, i \models \varphi \;\text{ for all } i \text{ with } n < i < m \\
s, V, n \models \varphi\, \mathcal{S}\, \psi & \text{iff} & \text{there exists } m \text{ with } 0 \leq m < n \text{ such that } \; s, V, m \models \psi \;\text{ and} \\
 & & s, V, i \models \varphi \;\text{ for all } i \text{ with } m < i < n \quad .
\end{array}
$$

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. $\forall\, n : n \geq 0$). The semantics of the remaining temporal operators can now be defined for arbitrary φ and ψ as follows:

$$
\begin{array}{lll}
\Diamond\, \varphi & \equiv & \mathit{true}\, \mathcal{U}\, \varphi \\
\Box\, \varphi & \equiv & \neg \Diamond \neg \varphi \\
\odot\, \varphi & \equiv & \mathit{false}\, \mathcal{U}\, \varphi \\
\varphi\, \mathcal{W}\, \psi & \equiv & \Box\, \varphi \;\vee\; \varphi\, \mathcal{U}\, \psi \\
\blacklozenge\, \varphi & \equiv & \mathit{true}\, \mathcal{S}\, \varphi \\
\blacksquare\, \varphi & \equiv & \neg \blacklozenge \neg \varphi \\
\mathcal{J}\, \varphi & \equiv & \varphi \;\wedge\; \neg \varphi\, \mathcal{S}\, \neg \varphi \quad .
\end{array}
$$

Predicate $\mathcal{I}$ characterizes the initial state (i.e., n=0) and is equivalent to $\neg(\mathit{true}\, \mathcal{S}\, \mathit{true})$. 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 $\mathit{leader}(i)$ which represents the fact that the process with identity i is the current leader. This identity i is part of a countable set Id totally ordered by $<$. We use i, j, k to denote elements of Id.

In our requirements we use quantification over Id. We stress that 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 an auxiliary predicate *participating* and replacing every universal quantification $(\forall\, i :: \ldots)$ by $(\forall\, i : \mathit{participating}(i) : \ldots)$ and replacing every existential quantification $(\exists\, i :: \ldots)$ by $(\exists\, i : \mathit{participating}(i) : \ldots)$. For ease of notation we have left this intended form of quantification implicit.

The requirements for the protocol 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).

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

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).

$$\mathrm{P2} \quad : \quad \Diamond\, (\exists\, i :: \mathit{leader}(i)) \quad .$$

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 process with a higher identity will capitulate eventually (we do not state anything about the possible future leadership of this 'better' process)[1].

$$\mathrm{P3} \quad : \quad (\forall\, i :: \mathit{leader}(i) \;\wedge\; (\exists\, j : i < j : \neg\, \mathit{leader}(j)) \;\Rightarrow\; \Diamond\, \neg\, \mathit{leader}(i)) \quad .$$

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

$$\begin{aligned} \mathrm{P4} \quad : \quad (\forall\, i, j :: \; & \mathit{leader}(i) \;\wedge\; \odot\; \neg\, \mathit{leader}(i) \\ & \wedge\; (\forall\, k :: \neg\, \mathit{leader}(k))\; \mathcal{U}\; \mathit{leader}(j) \;\Rightarrow\; i < j) \quad , \end{aligned}$$

where we refer to the last moment of leadership of process i (first two conjuncts in premise) and the moment of succession of process j (third conjunct).

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'.

[1] Note that the assumption that j is no leader is superfluous in light of P1. We have added this assumption because we think the formulation of P3 is more clear in this way.

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 *failed* 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 failed 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 failed, 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 failed) the rest of the candidates may not receive a response of the leader, remaining candidate forever. 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 Figure 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.

Notice that we deliberately have chosen to permit the leader process only to deal with succession inquiries. This is accomplished by distinguishing between messages originated by the leader and those originated by candidates. When both the leader and candidates transmit their identities by the same message type one should realize that candidates may force other candidates to become failed which may cause violation of P2. This can be seen as follows. Consider the following scenario of three processes, p, q, and r, one of which is a leader, r, say. Assume p and q do not take part in the election yet. Let $p{>}q{>}r$. Suppose q joins the election by transmitting its identity. Since p is still in the start state it ignores q's id. Before r reacts on the receipt of q's id, p joins the election and transmits its id. This will force q to become failed. As r capitulates (due to q's id received earlier) and as q will not become its successor (due to p's id) no process is able to grant p the right of succession, and, consequently, no leader process will ever be elected. The problem is that a candidate may not only be forced to become failed 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 failed or leader only on receipt of messages from leaders and they ignore others. In the above example q will thus not become failed on receipt of p's id.

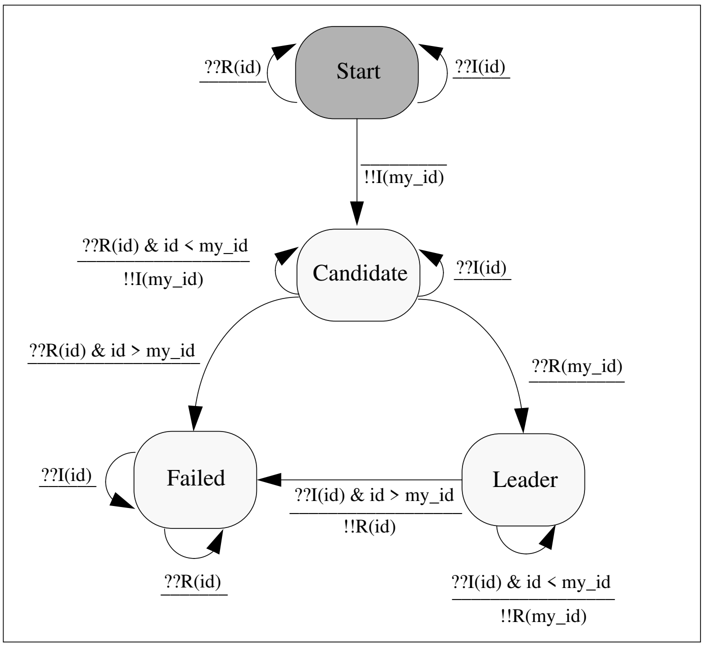

Figure 1: Finite state machine diagram of Protocol 1.

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. As 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.

A straightforward approach to detect the absence of a leader is to equip each process with a *timer* process and to detect the absence of a leader by means of a timeout mechanism. A timer is started by the *start-timer* action. A *timeout* is modeled as an ordinary action and may be used as (part of) a guard. In contrast to ordinary guards, timeout actions can be used to detect the establishment of a *global* condition in a protocol. They are abstract in the sense that they do not describe how the occurrence of this global condition can be detected using a kind of clock mechanism. A similar treatment of timeout actions is recently given in [Gou93].

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 messages 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 failed by receiving messages from other candidates. In Protocol 1 a candidate only reacts to messages sent by the leader.

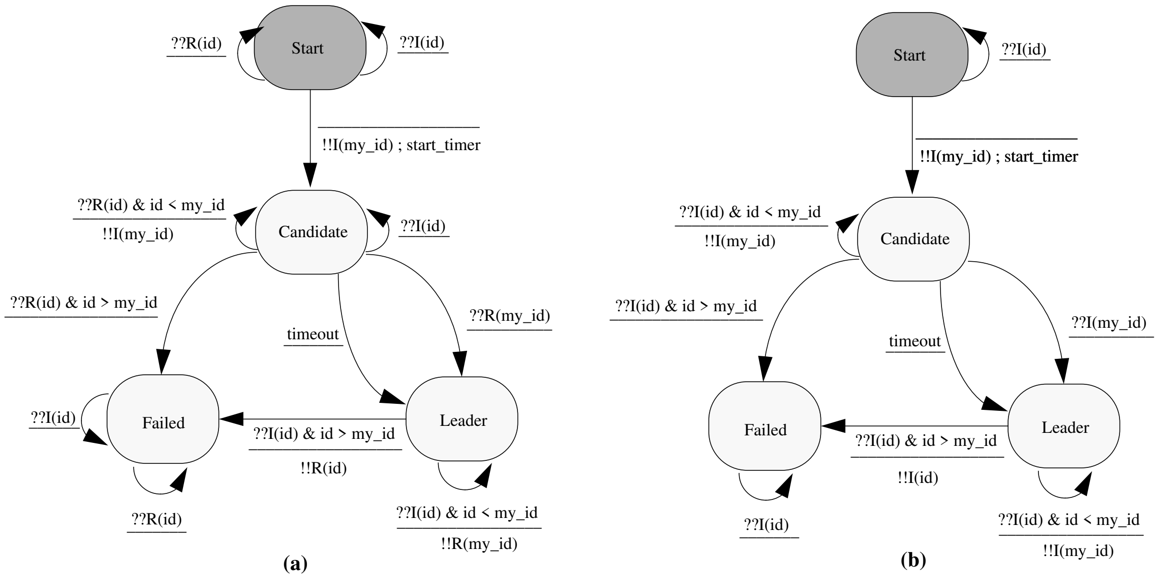

Figure 2: Finite state machine diagrams of two derivates of Protocol 1.

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 failed 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. [Fis91]). 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 [Dij74, Sch93] where processes may recover in any state. The number of

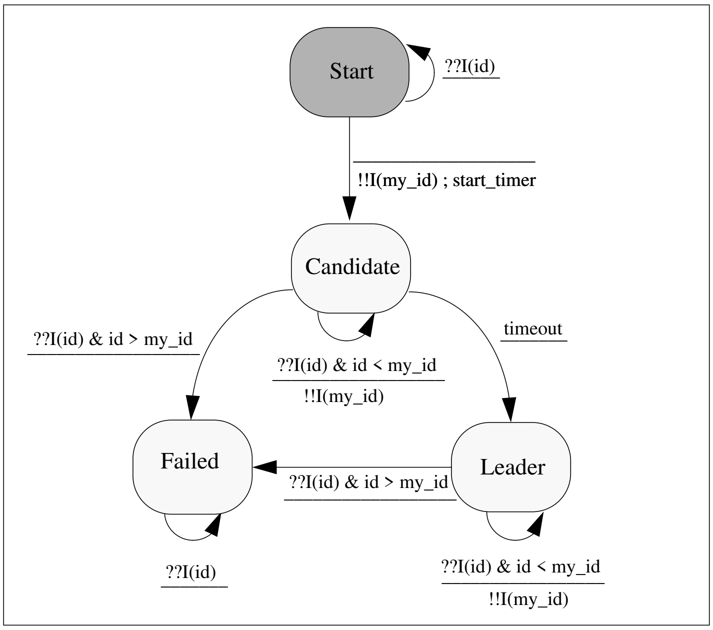

Figure 3: Finite state machine diagram of Protocol 2.

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 reformulate the requirements.

3.4.1 Requirements Revisited

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

$$\text{Q1} \quad : \quad (\exists\, i :: \mathit{leader}(i) \;\Rightarrow\; (\forall\, j : i \neq j : \neg\, \mathit{leader}(j))) \quad .$$

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

As stated above, it is unrealistic to demand P2 since potentially all processes may fail. We therefore only claim P2 in case 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 $\mathit{dead}(i)$ indicates the fact that process i has crashed. Formally,

$$\text{Q2} \quad : \quad \Diamond\,(\exists\, i :: \Box\,(\neg\, \mathit{dead}(i) \;\wedge\; (\forall\, j : i < j : \mathit{dead}(j)))\,) \;\Rightarrow\; \Box\,\Diamond\,(\exists\, i :: \mathit{leader}(i)) \quad .$$

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

$$\text{Q3} \quad : \quad (\forall\, i :: \neg\,(\mathit{leader}(i) \;\wedge\; \mathit{dead}(i))) \quad .$$

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 the following weakened variant of P3:

$$\text{Q4} \quad : \quad (\forall\, i, j :: \mathit{leader}(i) \;\wedge\; \neg\, \mathit{dead}(j) \;\wedge\; i < j \;\Rightarrow\; \Diamond\, \neg\, \mathit{leader}(i) \;\vee\; \Diamond\, \mathit{dead}(j)) \quad .$$

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,

$$\text{Q5} \quad : \quad (\forall\, i :: \mathcal{J}\; \neg\, \mathit{leader}(i) \;\Rightarrow\; \mathit{dead}(i) \;\vee\; \blacklozenge(\exists\, j : i < j : \neg\, \mathit{dead}(j))) \quad .$$

Both Q4 and Q5 refer to the capitulation 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

$$\begin{aligned} \text{Q6} \quad : \quad & (\forall\, i, j :: \mathit{leader}(i) \;\wedge\; \odot\; \neg\, \mathit{leader}(i) \\ & \qquad \wedge\; ((\forall\, k :: \neg\, \mathit{leader}(k)) \;\wedge\; \neg\, \mathit{dead}(i))\; \mathcal{U}\; \mathit{leader}(j) \;\Rightarrow\; i \leq j) \quad . \end{aligned}$$

Informally formulated: given some leader process, i say, its immediate 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 as it may capitulate due to the presence of a better candidate that crashes before becoming a leader.

We may consider Q2 and Q4 as weakened variants of P2 and P3 respectively. This weakening is needed since we now allow crashes. The relationship between Q6 and P4 is more subtle. When processes may not crash Q6 boils down to the corresponding

$$(\forall\, i :: \mathit{leader}(i) \;\Rightarrow\; \Box\, (\forall\, j :: \mathit{leader}(j) \;\Rightarrow\; i \leq j)) \quad .$$

This requirement, however, in the context of the previous protocols 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 capitulates due to the presence of a better candidate one expects that at some time a new (and better) leader emerges. Therefore, 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 leads us to 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 failed process might be a valid successor.

So as to involve failed 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 failed process is present, failed 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 failed. It should be observed that this does not suffice in case a leader crashes, at least one failed 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 failed processes will rejoin the election in absence of a leader. 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 failed to the candidate state, such that a failed process may (re-)join the election spontaneously by identifying itself and starting its timer[3].

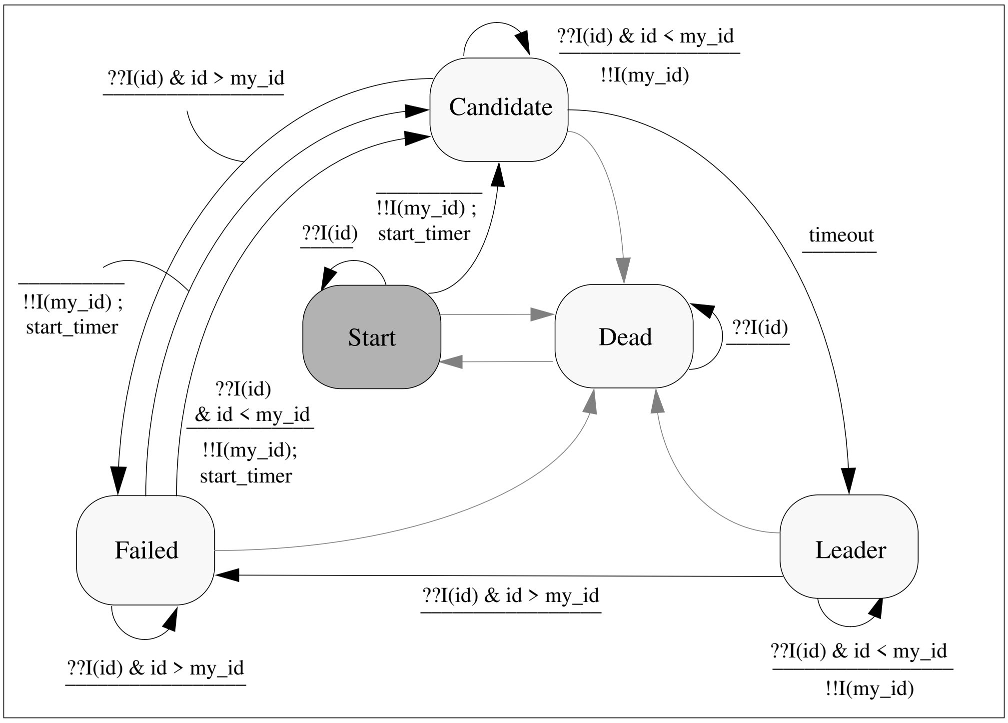

Figure 4: Finite state machine diagram of Protocol 3.

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

[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 failed process, thus forcing it to become starting (or candidate). Another possibility would be to let a failed process regularly check whether a leader is present (see e.g. [GZ86]).

[3]It should be noted that we now have two transitions with equivalent actions, one of which has a true guard from the failed state to the candidate state. These transitions can not 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.

(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 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 $\longrightarrow_{\diamond}$, respectively $\longrightarrow_{\Box}$ of modal transition systems (see e.g. [LT88]).

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, transition labels are omitted when both its associated guard and set of actions are absent.

3.5 Complexity Analysis of the Protocols

3.5.1 Introduction

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 number of bits in a message). The bit complexity of all presented protocols is $\mathcal{O}(\log N)$, where N is the total number of processes. For Protocol 1 we remark that the time complexity is equal to the message complexity.

In this section we analyze the worst case message complexity of our protocols. In our protocols all messages are broadcasted, so each message is received by all processes (except the sender). In a dynamic broadcast protocol, with processes starting up during protocol execution, each process at least has to send one (initial) message to the other processes so as to present itself, so the message complexity is at least $\mathcal{O}(N)$. Due to the dynamic character of the protocol each message needs an answer. If each process answers each message that has been received so far by sending a new message, we may expect a worst case message complexity exponential to N.

3.5.2 Complexity of Protocol 1

The following theorem holds for the message complexity of Protocol 1, where $MC_1^q(N,i)$ represents the number of messages sent by N processes participating in the election, process i being the initial leader. For reasons of simplicity an identity is represented by a positive natural number.

Theorem 3.1 $MC_1^q(N,i) = \frac{1}{2}N^2 + \frac{1}{2}N - \frac{1}{2}i^2 + \frac{3}{2}i - 2.$

Proof: Each process that becomes a candidate sends an initial I-message. For all processes k with $k<i$ this message will be answered by a message $R(m)$ with $m>k$, which will bring process k to the failed state. From this state no messages are sent, so these $i-1$ processes each contribute 1 message to $MC_1^q(N,i)$. In the worst case scenario process i sends $i-1$ R-messages in reaction on these I-messages.

In the worst case scenario all processes k with $k>i$ send their initial I-message, with $I(i+1)$ first, and become candidate before the initial leader replies with its (final) message $R(i+1)$.

Thus process $i+1$ becomes the new leader. But $R(i+1)$ also evokes an I-message from all candidate processes with an *id* greater than $i+1$. If these messages are sent with $I(i+2)$ first, the whole story repeats itself, until finally process N becomes leader. In each "round" the number of participants is reduced by one and the number of reactions on an R-message is maximal. So the scenario described above indeed is the worst case.

Process k ($i<k\leq N$) receives $k-1$ R-messages before it becomes leader. The *id*s of the first $k-2$ R-messages are smaller than k, so $k-2$ times an $I(k)$-message is sent from the candidate state. The *id* of the last R-message is equal to k, which makes k the new leader. All processes, except the final leader, will send an R-message when they capitulate. Together with the initial I-message this leads to a total number of k transmitted messages for processes $i<k<N$ and $N-1$ transmitted messages for process N.

The total number of messages for all processes now becomes

$$MC_1^q(N,i) = (\sum_{k=1}^{i-1} 1) + i + (\sum_{k=i+1}^{N} k) - 1 = \tfrac{1}{2}N^2 + \tfrac{1}{2}N - \tfrac{1}{2}i^2 + \tfrac{3}{2}i - 2 \qquad \square$$

One can easily infer that the worst case message complexity is reached for i=1 or i=2 and is equal to $\frac{1}{2}N^2+\frac{1}{2}N-1$. Contrary, if process N is the initial leader we get $MC_1^q(N,N) = 2N-2$. So, in that case the message complexity reduces to $\mathcal{O}(N)$. Figure 5 illustrates the worst case behaviour of Protocol 1 for $N=4,\ i=1$. Each I-message and R-message is subscripted with either an i (initial message) or a number k, indicating that this message is a reaction on the k-th message transmitted so far. We suppose that a local buffer is empty at the moment the initial message is sent.

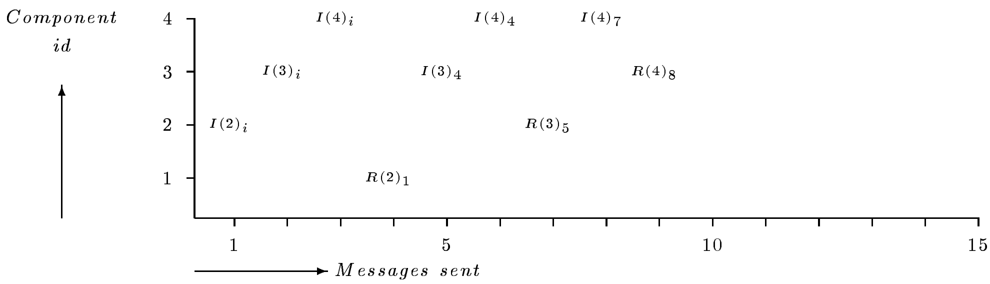

Figure 5: Worst case behaviour of Protocol 1 with queueing.

The message complexity of $\mathcal{O}(N^2)$ 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 with the largest id received upon then, but not processed until so far. In this way a buffer contains at most one I-message at a time. Adopting this tricky buffering mechanism to Protocol 1, reduces the message complexity to $\mathcal{O}(N)$, independent of the initial leader:

Theorem 3.2 $\quad MC_1^s(N) = 2N-2$.

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 failed state. Worst case protocol behaviour is now observed if 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. So, in the worst case $2(N-1)=2N-2$ messages are needed. □

Figure 6 shows this worst case behaviour, with the component with the highest *id* as the initial leader.

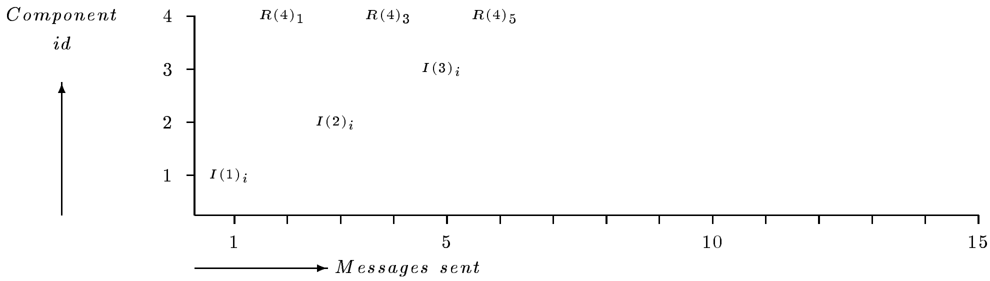

Figure 6: Worst case behaviour of Protocol 1 with smart buffering.

3.5.3 Complexity of Protocol 2

Compared to Protocol 1 we may expect a worse message complexity, because in the candidate state each reception of an I-message with a lower *id* evokes the transmission of a new I-message. In Protocol 1 only the reception of an R-message evoked a new message in the candidate state.

We assume that all processes are in the start state. The worst case message complexity of this protocol is observed when all processes send their initial I-message within a short time interval. To put it in a more quantitative way: all participating processes send their initial I-message within a time interval that is smaller than the timeout interval of a timer in the candidate state. We will also suppose that a process starts with an empty local buffer, local history begins at the moment the initial I-message is sent.

Figure 7 shows the worst case behaviour of Protocol 2 for N=4.

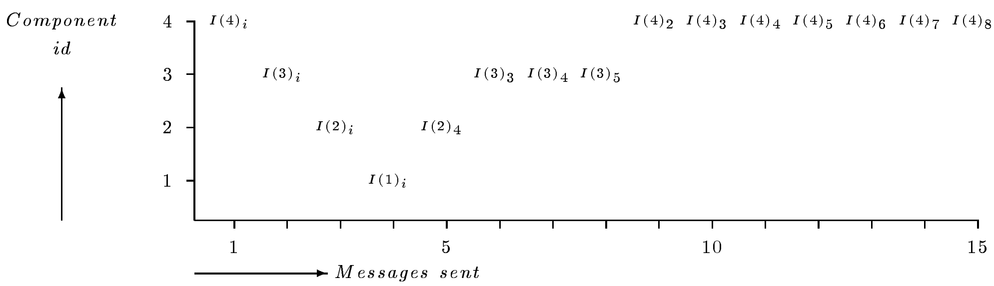

Figure 7: Worst case behaviour of Protocol 2 with simple buffering.

With simple buffering (queueing of all incoming messages) we obtain a complexity exponential to N, whereas for smart buffering this reduces to $\mathcal{O}(N)$. This is stated in the following theorems.

Theorem 3.3 $MC_2^q(N) = 2^N - 1$.

Proof: By induction on N. If N=1 only one initial message is sent, so $MC_2^q(1) = 1$. Now suppose $MC_2^q(N-1) = 2^{N-1} - 1$. In the worst case scenario initial I-messages are sent in order of decreasing *id*s. After the transmission of its initial message process N will first buffer the other $N-1$ initial messages and all replies from processes $2 \ldots N-1$ before it replies by sending an I-message to each of them separately. So we get $MC_2^q(N) = 2MC_2^q(N-1)+1 = 2(2^{N-1}-1)+1 = 2^N-1$. □

Theorem 3.4 $MC_2^s(N) = 2N - 1$.

Proof: Each process transmits an initial I-message. In the worst case all processes except the future leader will have to be brought to the failed state by a separate I-message from a process with a higher *id*. So $N + N - 1 = 2N - 1$ messages are needed. □

Figure 8 shows a worst case behaviour of Protocol 2 with smart buffering for 4 components.

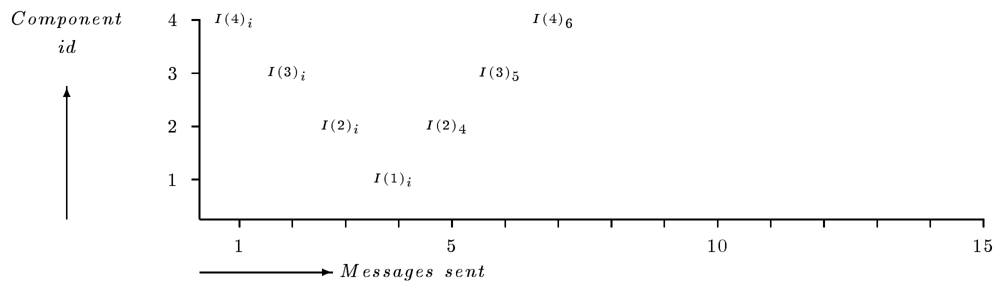

Figure 8: Worst case behaviour of Protocol 2 with smart buffering.

3.5.4 Complexity of Protocol 3

First we consider an election without crashing processes. With 'simple' buffering, the worst case message complexity of Protocol 3 is the same as for Protocol 2. With smart buffering the message complexity increases to $\mathcal{O}(N^2)$. This is stated in the following theorems.

Theorem 3.5 $MC_3^q(N) = 2^N - 1$.

Proof: See Protocol 2. Compared to Protocol 2, there are more situations in which the worst case behaviour occurs. A process that wakes up from the failed state may evoke messages from processes with a higher *id*. □

Theorem 3.6 $MC_3^s(N) = \frac{1}{2}N^2 + \frac{1}{2}N$.

Proof: An initial I-message from a process with a lower *id* causes a transition from the failed state to the candidate state for a process with a higher *id*. This transition is accompanied by the transmission of an I-message. If a leader with a higher id is already present, an extra message is needed to put the process back to the failed state again. This leads to a worst case message complexity of $\sum_{i=1}^{N} i = \frac{1}{2}N^2 + \frac{1}{2}N$. $\square$

Figure 9 shows an example of the worst case behaviour of Protocol 3 with 4 components initially in the start state.

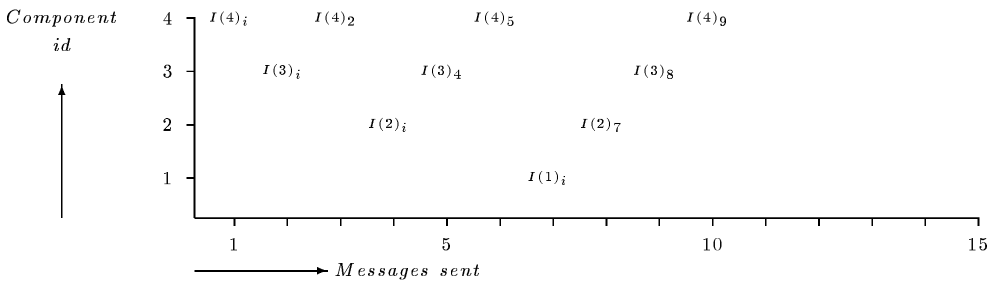

Figure 9: Worst case behaviour of Protocol 3 with smart buffering.

Finally we analyze the complexity in case K processes crash ($0 \leq K < N$). 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 failed 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 failed).

Protocol	buffer	MC
1	queue	$\frac{1}{2}N(N+1) - 1$
1	smart	$2N - 2$
2	queue	$2^N - 1$
2	smart	$2N - 1$
3	queue	$2^N - 1$
3	smart	$\frac{1}{2}N(N+1)$

Table 1: Overview of worst case message complexities of all protocols.

The worst case message complexity involving the crash of K out of N participating processes is given by

Theorem 3.7 $MC_3^c(N, K) = \frac{1}{6}K^3 - \frac{1}{2}NK^2 + (\frac{1}{2}N^2 - \frac{1}{6})K$.

Proof: The worst case scenario is as follows: the leader (process N) crashes, and failed processes become candidate in decreasing order of ids. This leads to a new election with $N-1$ processes. From Theorem 3.6 we know that this requires $\sum_{i=1}^{N-1} i$ messages. If this scenario is repeated for the subsequent crashes of processes $N-1, N-2, \ldots, N-K+1$, we get

$MC_3^c(N,K) = \sum_{k=1}^{K}(\sum_{i=1}^{N-k} i)$. Elimination of the sum constructs leads to the result stated above. $\square$

The results of this section are summarized in Table 1.

4 Verification by Temporal Logic

4.1 Introduction

In the previous section we informally motivated our design decisions. In this section we formally prove that the protocols designed in section 3 satisfy their requirements. That is, we prove that Protocols 1 and 2 satisfy requirements P1 through P4 and Protocol 3 satisfies Q1 through Q6. We, furthermore, prove that for all three protocols unspecified receptions cannot occur. We stress that 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 protocols 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.

In the rest of this section 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 state 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 failed 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.

We first formally define some relevant assumptions about the broadcast mechanism. Let m, m_1, and m_2 be unique messages, that is, both their originator and moment of origination are unique. (It has been shown in [Koy89] that messages need to be uniquely identifiable so as to specify communication mechanisms in temporal logic by axioms like those below.)

Assumption 4.1

$$(\forall\, i :: \text{SEND}_i(m) \;\Rightarrow\; (\forall\, j : i \neq j : \Diamond\, \text{RCV}_j(m))\,) \quad .$$

Assumption 4.2

$$(\forall\, i :: \text{RCV}_i(m) \;\Rightarrow\; \blacklozenge(\exists\, j : i \neq j : \text{SEND}_j(m))\,) \quad .$$

Assumption 4.3

$$\begin{aligned}&(\forall\, i, j :: \text{SEND}_i(m_1) \;\wedge\; \Diamond\, \text{SEND}_j(m_2)\\ &\quad \Rightarrow\; (\forall\, k : k \neq i \wedge k \neq j : \Diamond\,(\text{RCV}_k(m_1) \;\wedge\; \Diamond\, \text{RCV}_k(m_2)))\,) \quad .\end{aligned}$$

Assumption 4.1 states that messages are not lost by the communication network, 4.2 phrases that messages are not spontaneously generated by the network, and 4.3 expresses that the network is order-preserving. Observe that it immediately follows from 4.2 that a process does not receive its own transmitted messages. That is, for all messages m

Property 4.4

$$(\forall\, i :: \text{RCV}_i(m) \;\Rightarrow\; \blacksquare\, \neg\, \text{SEND}_i(m)) \quad .$$

4.2 Verification of Protocol 1

We now start with the proof of the correctness of Protocol 1. We deal with the requirements one by one. As P4, stating that successive leaders are 'better', is the crux to the proofs of P2 and P3, we present the proof of P4 after the proof of P1. The first proof obligation is:

$$\text{P1} \quad : \quad (\exists\, i :: L_i \ \Rightarrow \ (\forall\, j : i \neq j : \neg\, L_j)) \quad .$$

Define predicate Q as follows:

$$Q \ \equiv \ N_l + N_r \leq 1 \quad ,$$

where N_l is defined by $(\#\, i :: L_i)$ and N_r equals $(\#\, i :: R(i) \in Q_i)$. $\#$ denotes 'number of'. By definition, $0 \leq N_r$ and $0 \leq N_l$. It immediately follows $Q \Rightarrow$ P1.

Initially we have assumed

Assumption 4.5

$$\mathcal{I} \ \Rightarrow \ (N_l = 1 \ \wedge \ (\forall\, i :: Q_i = \text{empty})) \quad ,$$

which implies that Q holds initially. The rest of the proof concentrates on establishing

Lemma 4.6

$$Q \ \Rightarrow \ \Box\, Q \quad .$$

From this lemma we may then conclude P1.

Proof: Assume Q holds. By definition Q can only be falsified when either N_l or N_r (or both) increases. We consider an increase of either N_l or N_r by one. Later on we show that considering these cases suffices.

Consider an increase of N_r by 1. So there is one process, j say, that buffers an $R(j)$ message. We infer from the protocol description that only a leader process can transmit R-messages:

Property 4.7

$$(\forall\, i, j :: \text{SEND}_i(R(j)) \ \Rightarrow \ L_i) \quad .$$

According to our definition of broadcasting a sender does not receive its own messages. So, for process j to buffer $R(j)$, there must be another process, i say, which has transmitted this message, and consequently (according to 4.7) L_i holds at transmitting it. A leader i only transmits $R(j)$ $(j \neq i)$ when it capitulates:

Property 4.8

$$(\forall\, i, j : i \neq j : \text{SEND}_i(R(j)) \ \Rightarrow \ \Box\, F_i) \quad .$$

Consequently, a leader transmits only once such a message. From the above, we may now conclude that whenever N_r is increased by one, N_l must be decreased by one.

Now consider an increase of N_l by one. By a similar reasoning as above we prove that this must be accompanied by a decrease of N_r by one. First, it can be inferred from the protocol description that process i can only become a leader after receipt of message $R(i)$. This can be formalized as follows

Property 4.9

$$(\forall\, i :: \neg\, L_i \;\wedge\; \Box \neg\, \mathrm{RCV}_i(R(i)) \;\Rightarrow\; \Box \neg\, L_i) \quad .$$

Furthermore it is quite evident that process i can only perform $\mathrm{RCV}_i(m)$, for some message type m, by extracting m from Q_i,

Property 4.10

$$(\forall\, i :: \mathrm{RCV}_i(m) \;\Rightarrow\; m \notin Q_i) \quad .$$

Considering R-messages the above implies that N_l can only be increased by one after a decrease of N_r by one.

Since an increase of N_l (N_r) by one is coupled by a decrease of N_r (N_l) by one it follows —given that $0 \leq N_l, N_r \leq 1$— that considering the above two cases suffices.
(**End of proof P1**.)

$$\mathrm{P4} \;:\; (\forall\, i,j :: L_i \;\wedge\; \odot\, \neg\, L_i \;\wedge\; (\forall\, k :: \neg\, L_k)\, \mathcal{U}\, L_j \;\Rightarrow\; i < j) \quad .$$

Proof: When a leader never capitulates P4 holds trivially. Consider the case that at some time a leader will capitulate. Assume $L_i \wedge \odot\, \neg\, L_i \wedge (\forall\, k :: \neg\, L_k)\, \mathcal{U}\, L_j$. According to 4.9 j may only become a leader after receipt of $R(j)$. Moreover, R-messages are only transmitted by leader processes (see 4.7). The idea now is to show that process i must have transmitted $R(j)$, and $i{\neq}j$. From the protocol description $i{<}j$ may then be concluded, due to

Property 4.11

$$(\forall\, i,j : i \neq j : \mathrm{SEND}_i(R(j)) \;\Rightarrow\; i < j) \quad .$$

The proof of P4 is as follows. It can easily be verified that $i{\neq}j$ since we have from the protocol description

Property 4.12

$$(\forall\, i :: L_i \;\wedge\; \odot\, \neg\, L_i \;\Rightarrow\; \odot\, F_i) \quad ,$$

Property 4.13

$$(\forall\, i :: F_i \;\Rightarrow\; \Box\, F_i) \quad ,$$

from which it immediately follows

Lemma 4.14

$$(\forall\, i :: L_i \;\wedge\; \odot\, \neg\, L_i \;\Rightarrow\; \Box\, \neg\, L_i) \quad .$$

Furthermore, from the invariance of Q (see proof P1) we have

$$L_i \;\Rightarrow\; \neg\, L_j \;\wedge\; (\forall\, k :: R(k) \notin Q_k) \quad .$$

So, either process i or some successor of i must have transmitted $R(j)$. Since j is the immediate successor of i, i must have sent $R(j)$, and thus (see 4.11) $i<j$.
(**End of Proof P4**.)

$$\text{P2} \quad : \quad \Diamond\, (\exists\, i :: L_i) \quad .$$

Proof: Since initially there is one leader process P2 holds trivially when a leader never capitulates. Therefore consider the case when at some time a leader capitulates. From the protocol it immediately follows that a leader i transmits $R(j)$ at capitulation (see 4.8). We prove that once $R(j)$ is transmitted j will become a leader sooner or later. Formally:

Lemma 4.15

$$(\forall\, j :: R(j) \in Q_j \;\Rightarrow\; \Diamond\, L_j) \quad .$$

We have from the protocol

Property 4.16

$$(\forall\, j :: C_j \;\wedge\; \text{RCV}_j(R(j)) \;\Rightarrow\; \odot\; L_j) \quad .$$

Informally, a candidate j becomes a leader once it receives an $R(j)$ message. By proving

Lemma 4.17

$$(\forall\, j :: R(j) \in Q_j \;\Rightarrow\; \Diamond\, (C_j \;\wedge\; \text{RCV}_j(R(j)))\ ) \quad ,$$

we may —using $4.16 \wedge 4.17 \Rightarrow 4.15$— conclude 4.15. Since transmitted messages are always received and processed at some time (see 4.1) we concentrate on proving that C_j holds on processing $R(j)$. We have that a leader i only transmits $R(j)$ after receipt of an $I(j)$ message with $i<j$. Or,

Property 4.18

$$(\forall\, i, j :: \Box\, \neg\, (\text{RCV}_i(I(j)) \;\wedge\; i < j) \;\Rightarrow\; \Box\, \neg\, \text{SEND}_i(R(j))\ ) \quad .$$

Besides, only candidate and start processes may transmit I-messages.

Property 4.19

$$(\forall\, j :: \text{SEND}_j(I(j)) \;\Rightarrow\; C_j \;\vee\; S_j) \quad .$$

After sending $I(j)$ a start process j becomes a candidate immediately,

Property 4.20

$$(\forall j :: S_j \;\wedge\; \text{SEND}_i(I(i)) \;\Rightarrow\; \odot\, C_j) \quad .$$

A stronger variant of lemma 4.17 is now proven. A process j that receives $R(j)$ at some time remains candidate from sending $I(j)$ until receipt of $R(j)$. Formally,

Lemma 4.21

$$(\forall j :: \text{SEND}_j(I(j)) \;\wedge\; \Diamond\, \text{RCV}_j(R(j)) \;\Rightarrow\; C_j \;\mathcal{U}\; \text{RCV}_j(R(j))\;) \quad .$$

From the protocol we have that a candidate j only leaves the candidate state after receiving $R(i)$ with $j{\leq}i$.

Property 4.22

$$(\forall j :: C_j \;\wedge\; \Box\, \neg\, (\exists\, i : j \leq i : \text{RCV}_j(R(i))) \;\Rightarrow\; \Box\, C_j) \quad .$$

We now prove

Lemma 4.23

$$(\forall j :: \Diamond\, \text{RCV}_j(R(j)) \;\Rightarrow\; \neg\, (\exists\, k : j \leq k : \text{RCV}_j(R(k)))\; \mathcal{U}\; \text{RCV}_j(R(j))\;) \quad .$$

By contradiction. Assume that process j receives $R(k)$ $(j{\leq}k)$ before receiving $R(j)$. This is impossible due to the following lemma.

Lemma 4.24

$$(\forall i,j,k :: \text{RCV}_j(R(i)) \;\wedge\; \Diamond\, \text{RCV}_j(R(k)) \;\Rightarrow\; (j = i \Rightarrow i < k \;\wedge\; j \neq i \Rightarrow i \leq k)) \quad .$$

It immediately follows that 4.24 implies 4.23. Informally, process j receives at most once $R(j)$, and moreover, for any process the parameters of received R-messages form an ascending sequence. Lemma 4.24 can be proven as follows. It is already stated before that only leaders transmit R-messages (see 4.7). A leader i transmits zero or more times $R(i)$ followed by (at most) one time $R(j)$ $(i{<}j)$. So, a single leader generates an ascending sequence of R-messages. From P1 it follows that there is at most one leader at a time. We know from P4 that subsequent leaders are increasing—leaders become 'better'. We may now conclude lemma 4.24 since processes do not receive their own transmitted messages (property 4.4).
(**End of Proof P2**.)

$$\text{P3} \quad : \quad (\forall i :: L_i \;\wedge\; (\exists\, j : i < j : \neg\, L_j) \;\Rightarrow\; \Diamond\, \neg\, L_i) \quad .$$

Proof: The remaining requirement to be proven is P3. The idea is to reformulate P3 in terms of internal states of the protocol, using $\neg\, L_j \equiv S_j \;\vee\; C_j \;\vee\; F_j$. Since failed processes remain failed indefinitely once they become failed (see property 4.13), and since failed processes are 'less' than leaders

Lemma 4.25

$$(\forall\, i, j :: L_i \;\wedge\; F_j \;\Rightarrow\; i > j) \quad ,$$

we do not have to consider failed processes. Of course it remains to prove lemma 4.25. There are only two possible transitions by which a process can become failed

Property 4.26

$$\begin{aligned}(\forall\, j :: \odot \mathcal{J}\, F_j \;\Rightarrow\; &(L_j \;\wedge\; (\exists\, k : j < k : \text{RCV}_j(I(k))) \;\vee \\ &(C_j \;\wedge\; (\exists\, k : j < k : \text{RCV}_j(R(k))) \;) \quad .\end{aligned}$$

Property 4.26 follows directly from the protocol description. Now consider each transition in isolation. In case $L_j \wedge \odot\; F_j$ lemma 4.25 follows directly from the fact that, according to P4, subsequent leaders will be better. In the other case j becomes failed on receipt of $R(k)$, $k{>}j$. From this reception we know that $R(k)$ is transmitted some time ago (see 4.2). From lemma 4.15 we infer that k has (or will) become a leader. In case it has been or still is a leader 4.25 follows immediately from P4. From the invariance of Q (see proof of P1) and lemma 4.15 we deduce that

Lemma 4.27

$$(\forall\, k :: R(k) \in Q_k \;\Rightarrow\; \neg\, (\exists\, i :: L_i)\, \mathcal{U}\; L_k) \quad .$$

In case k is not yet a leader this lemma implies that it will be the next leader, from which —again using P4— lemma 4.25 can be inferred. This concludes the proof of lemma 4.25.

We now continue the proof of P3. According to the fair semantics of transitions each process in the start state will become a candidate eventually. Or,

Property 4.28

$$(\forall\, i :: S_i \;\Rightarrow\; \Diamond\, C_i) \quad .$$

Therefore, it is sufficient to consider the following variant of P3:

Lemma 4.29

$$(\forall\, i :: L_i \;\wedge\; (\exists\, j : i < j : C_j) \;\Rightarrow\; \Diamond\, \neg\, L_i) \quad .$$

The proof of lemma 4.29 is as follows. We have

Property 4.30

$$(\forall\, i :: S_i \;\Rightarrow\; \neg\, C_i\, \mathcal{U}\; \text{SEND}_i(I(i))\;) \quad .$$

That is, a process transmits an $I(i)$ message before becoming a candidate. The crucial property now is

Lemma 4.31

$$(\forall\, i :: L_i \;\wedge\; (\exists\, j : i < j : C_j) \;\Rightarrow\; \Diamond\, (\exists\, k : i < k : \text{RCV}_i(I(k)))\;) \quad ,$$

and since a leader process i capitulates as soon as it receives $I(k)$ ($i<k$) (see properties 4.8 and 4.18) we may conclude from lemmata 4.29 and 4.31 that P3 holds.

It remains, of course, to establish lemma 4.31. Assume $L_i \wedge (\exists\, j : i<j : C_j)$. For i the initial leader the lemma follows quite straightforward. Let i not be the initial leader. Then i has become a leader on receipt of $R(i)$ (see property 4.9). Since messages are broadcasted and j has not itself transmitted $R(i)$, due to

Property 4.32

$$(\forall\, i :: L_i \;\Rightarrow\; \Box\, \neg\, C_i) \quad ,$$

j must have received $R(i)$ (cf. assumption 4.1). Now we have two possibilities, either S_j or C_j holds on receipt of $R(i)$. In both cases j transmits $I(j)$ eventually: in case of S_j to reach C_j and in case of C_j as a reaction on the receipt of $R(i)$. In both cases process i will process $I(j)$ after it has processed $R(i)$, so after i has become a leader.
(**End of Proof P3**.)

We have showed that the Protocol 1 satisfies P1 through P4, and, consequently, conforms to our requirements. Recall that unspecified receptions lead to abnormal termination of the protocol. So, our remaining proof obligation is to prove that unspecified receptions can not occur. For Protocol 1 this boils down to proving that a leader can not receive R-messages. This can easily be verified using that only leader processes transmit R-messages (property 4.7), that at most a single leader exists (P1), and the fact that processes do not receive their own messages (property 4.4). This completes the proof of Protocol 1.

4.3 Verification of Protocol 2

The purpose of this section is to prove that Protocol 2 satisfies requirements P1 through P4, and that no unspecified receptions can occur. We take a similar approach as in the previous section. As P4 is the crux of the proofs of both P2 and P3 (as in Protocol 1), its proof is presented just after the proof of P1.

4.3.1 Timeout Semantics

We first introduce some additional notations. For some protocol state guard TIMEOUT_i for process i evaluates to true whenever i's timeout occurs and the corresponding transition is taken. 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 P1 through P4.

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 transmission of message m, say. This message has to be received (and processed) by all its recipients before the timer may expire. Formally,

Assumption 4.33

$$(\forall\, i :: \text{SEND}_i(m^p) \;\Rightarrow\; \neg\, \text{TIMEOUT}_i^p \;\mathcal{W}\; (\forall\, j : i \neq j : \blacklozenge \text{RCV}_j(m^p))\,) \quad ,$$

where m^p is a unique message. (It has been shown in [Koy89] that messages need to be uniquely identifiable in order to specify communication mechanisms in temporal logic by axioms like 4.33. In this verification we accomplish this by numbering of the messages by the sender. From the context the dependence on the identification of the sender is explicit, so for simplicity this dependence is omitted.) 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 p of m. In the sequel we use p, q as numbers of messages. 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 in stead 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 then additionally require

Assumption 4.34

$$(\forall\, i :: \text{TIMEOUT}_i^p \;\Rightarrow\; (\forall\, j : i \neq j : r_{m^p,j} \notin Q_i)) \quad ,$$

where 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 4.33 and 4.34.

The formal semantics of a non-premature timeout in broadcasting networks is now defined by axioms 4.33 and 4.34. Summarizing, according to 4.33 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 4.34.

4.3.2 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.

The first property states that a timeout can only occur for candidate processes (and not in other states)

Property 4.35

$$(\forall\, i :: \text{TIMEOUT}_i \;\Rightarrow\; C_i) \quad .$$

[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 easily be verified that the presented protocols conform to this principle.

Another property which is used (implicitly) during the verification is that a process is only a candidate once. That is, once a process has left the candidate state it will never become a candidate anymore. This is formulated by

Property 4.36

$$(\forall\, i :: \neg\, C_i \;\wedge\; \blacklozenge C_i \;\Rightarrow\; \Box\, \neg\, C_i) \quad .$$

Furthermore, once a process is in the candidate state and given that it performs a timeout eventually it remains a candidate until this timeout happens,

Property 4.37

$$(\forall\, i :: C_i \;\wedge\; \Diamond\, \text{TIMEOUT}_i \;\Rightarrow\; C_i\, \mathcal{U}\; \text{TIMEOUT}_i) \quad .$$

Using that a candidate i becomes failed on receipt of $I(j)$, $i<j$,

Property 4.38

$$(\forall\, i :: C_i \;\wedge\; (\exists\, j : i < j : \text{RCV}_i(I(j))) \;\Rightarrow\; \odot\, F_i) \quad ,$$

we conclude

Lemma 4.39

$$(\forall\, i :: C_i \;\wedge\; \Diamond\, \text{TIMEOUT}_i \;\Rightarrow\; \neg\, (\exists\, j : i < j : \text{RCV}_i(I(j)))\, \mathcal{U}\; \text{TIMEOUT}_i) \quad .$$

Lemma 4.39 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)—otherwise process i would be forced to the failed state (see 4.38).

One can now infer from 4.33, 4.34, and 4.39 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 $I(i)$.

4.3.3 Proof of Requirements

We now start with proving the requirements one by one. The first proof obligation is:

$$\text{P1} \quad : \quad (\exists\, i :: L_i \;\Rightarrow\; (\forall\, j : i \neq j : \neg\, L_j)) \quad .$$

Proof: From the protocol we immediately deduce that a process can only become a leader after performing a timeout.

Property 4.40

$$(\forall\, i :: \Box\, \neg\, \text{TIMEOUT}_i \;\Rightarrow\; \Box\, \neg\, L_i) \quad .$$

Furthermore, we infer that on occurrence of a timeout a process becomes a leader immediately

Property 4.41

$$(\forall\, i :: \textsc{timeout}_i \;\Rightarrow\; \odot\, L_i) \quad ,$$

and, after just becoming a leader the process has performed a timeout:

Property 4.42

$$(\forall\, i :: \odot\, \mathcal{J}\, L_i \;\Rightarrow\; \textsc{timeout}_i) \quad ,$$

The above three equations give the relation between performing a timeout and becoming a leader.

The idea behind the proof is now as follows. We consider two different cases. In case no leader is present we must prove that it is not possible that two (or more) processes perform a timeout simultaneously, and consequently, become a leader at the same time. This follows directly from the interleaving semantics of our protocol description language which prevents processes to perform transitions, and thus timeouts, simultaneously. The second case we have to consider is the case in which we have a (set of) leader(s) and a new leader appears. Then the proof obligation is to establish that this may not give rise to more than one leader. In the rest of the proof we focus our attention on the latter case.

From the above relation between a timeout and becoming a candidate it immediately follows that it suffices to prove

Lemma 4.43

$$(\exists\, i :: \textsc{timeout}_i \;\Rightarrow\; (\forall\, j : i \neq j : \neg\, L_j)) \quad ,$$

According to property 4.35 a timeout can only occur when a process is in the candidate state. Initially, all processes are in the start state. A process only becomes a candidate after sending an I-message.

Property 4.44

$$(\forall\, i :: \Box\, \neg\, \textsc{send}_i(I(i)) \;\Rightarrow\; \Box\, \neg\, C_i) \quad .$$

In our protocol $\textsc{timeout}_i^p$ is associated to the (initial) transmission of message $I^p(i)$. From assumption 4.33 we infer that each process receives $I^p(i)$. The idea is to refer to the state of the recipient, process j say, at the moment of processing this message and to deduce that, for each possible state, this process j can not be a leader at the occurrence of $\textsc{timeout}_i$. Formally, we have:

Lemma 4.45

$$(\forall\, i, j :: \textsc{rcv}_j(I^p(i)) \;\wedge\; \Diamond\, \textsc{timeout}_i^p \;\Rightarrow\; \Box\, (\textsc{timeout}_i^p \Rightarrow \neg\, L_j)) \quad .$$

We now prove lemma 4.45 for each possible state of the recipient of $I^p(i)$, process j, given that i becomes a leader once (i.e. $\Diamond\, \textsc{timeout}_i^p$). Implicitly we use that process i is still a candidate when j receives $I(i)$.

Property 4.46

$$(\forall\, i :: (\exists\, j :: \textsc{rcv}_j(I^p(i))) \;\wedge\; \Diamond\, \textsc{timeout}_i^p \;\Rightarrow\; C_i) \quad .$$

First, consider the case that j is failed. Once a process is failed it remains failed, and, hence will never become a leader. Thus,

Property 4.47

$$(\forall\, j :: F_j \;\Rightarrow\; \Box\, F_j) \quad .$$

Consequently,

Lemma 4.48

$$(\forall\, i, j :: F_j \;\wedge\; \textsc{rcv}_j(I^p(i)) \;\wedge\; \Diamond\, \textsc{timeout}_i^p \;\Rightarrow\; \Box\,(\textsc{timeout}_i^p \Rightarrow F_j)) \quad ,$$

which concludes the proof for failed processes.

Secondly, consider j to be either a leader or a candidate. Abbreviate $C_j \;\vee\; L_j$ by CL_j. From the protocol specification we directly infer

Property 4.49

$$(\forall\, i, j :: CL_j \;\wedge\; \textsc{rcv}_j(I(i)) \;\Rightarrow\; (j < i \Rightarrow\; \odot\; F_j) \;\wedge\; (j > i \Rightarrow \textsc{send}_j(I(j)))\;) \quad .$$

Property 4.49 suggests a case analysis between $j{<}i$ and $j{>}i$. Consider $j{>}i$ and $\Diamond\, \textsc{timeout}_i^p$. According to 4.49 j replies by sending $I(j)$. According to 4.34 process i is forced to process this message since $I(j)$ is a reply to $I^p(i)$. But, as $j{>}i$ this contradicts with 4.39. Hence, the interesting case is $j{<}i$. Stated otherwise,

Lemma 4.50

$$(\forall\, i, j :: CL_j \;\wedge\; \textsc{rcv}_j(I^p(i)) \;\wedge\; \Diamond\, \textsc{timeout}_i^p \;\Rightarrow\; j < i) \quad .$$

From 4.50, 4.49, and 4.47 we now deduce

Lemma 4.51

$$(\forall\, i, j :: CL_j \;\wedge\; \textsc{rcv}_j(I^p(i)) \;\wedge\; \Diamond\, \textsc{timeout}_i^p \;\Rightarrow\; \Box\,(\textsc{timeout}_i^p \Rightarrow F_j)) \quad ,$$

which concludes the proof for candidate and leader processes.

Finally, consider the case that j is in the start state at the moment of receipt of $I(i)$. From the protocol description it immediately follows that start processes ignore all messages

Property 4.52

$$(\forall\, i :: S_i \;\wedge\; \textsc{rcv}_i(m_1) \;\Rightarrow\; \neg\, \textsc{send}_i(m_2) \;\wedge\; \odot\; S_i) \quad .$$

Distinguish between two cases. In the first case we assume that j remains in the start state until i's timeout occurs. This immediately implies that j is not a leader at the moment i's timeout occurs, and consequently we have

Lemma 4.53

$$(\forall\, i,j :: S_j \;\wedge\; \textsc{rcv}_j(I^p(i)) \;\wedge\; S_j\,\mathcal{U}\,\textsc{timeout}_i^p \;\Rightarrow\; \Box\,(\textsc{timeout}_i^p \Rightarrow S_j)) \quad .$$

In the second case we consider that j has left the start state after processing $I(i)$ and before i performs its timeout, that is $\neg\,(S_j\,\mathcal{U}\,\textsc{timeout}_i^p)$. According to

Property 4.54

$$(\forall\, j :: S_j \;\wedge\; \odot\,\neg\, S_j \;\Rightarrow\; \odot\, C_j) \quad ,$$

j has become a candidate and due to 4.44 must have sent $I(j)$ in order to do so. According to the broadcasting communication i will receive this message. As $I(j)$ is not a reply on $I(i)$, process i is not forced to process this message before performing its own timeout. This suggests the following case analysis. First, consider the case that i processes $I(j)$ before performing its timeout. According to 4.39 this implies that, given that i will perform its timeout once, $i>j$. Due to 4.49 i replies by transmitting $I(i)$, 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 4.39). In the other case i processes $I(j)$ after performing its timeout. But then, by definition j can not be a leader too at the moment i performs its timeout as it is forced, according to 4.34 to wait for the reply of i. So, we conclude

Lemma 4.55

$$(\forall\, i,j :: S_j \;\wedge\; \textsc{rcv}_j(I^p(i)) \;\wedge\; \neg\,(S_j\,\mathcal{U}\,\textsc{timeout}_i^p) \;\Rightarrow\; \Box\,(\textsc{timeout}_i^p \Rightarrow \neg\, L_j)) \quad .$$

Lemmata 4.53 and 4.55 directly imply

Lemma 4.56

$$(\forall\, i,j :: S_j \;\wedge\; \textsc{rcv}_j(I^p(i)) \;\wedge\; \Diamond\,\textsc{timeout}_i^p \;\Rightarrow\; \Box\,(\textsc{timeout}_i^p \Rightarrow \neg\, L_j)) \quad .$$

From lemmata 4.48, 4.51, and 4.56 we deduce (4.45). This completes the proof of P1.
(**End of Proof P1**.)

$$\text{P4} \;:\; (\forall\, i,j :: L_i \;\wedge\; \neg\odot L_i \;\wedge\; (\forall\, k :: \neg\, L_k)\,\mathcal{U}\, L_j \;\Rightarrow\; i<j) \quad .$$

Proof: Assume $L_i \wedge \neg \odot L_i \wedge (\forall\, k :: \neg\, L_k)\,\mathcal{U}\, L_j$, so j is the immediate successor of i. We have that $i{\neq}j$ in an equivalent way as in the proof of P4 for Protocol 1 (see previous section). We now prove that for a leader i it is always the case that leaders in the future will be at least as good as i (note that i may remain a leader for a while).

Lemma 4.57

$$(\forall\, i,j :: L_i \;\Rightarrow\; \Box\,(L_j \;\Rightarrow\; i \le j)) \quad .$$

From lemma 4.57 and $i \neq j$ we immediately deduce P4. The proof of 4.57 is as follows. Assume L_i. In case i never capitulates 4.57 holds trivially. Therefore, consider the case that i capitulates once. Let j be i's successor and assume $i>j$. From 4.34, 4.39, and 4.49 we infer that a process can not become a leader in presence of a better leader or candidate that has received its original I-message:

Lemma 4.58

$$(\forall j :: \mathcal{J} C_j \ \wedge \ (\exists k : j < k : \Diamond (\text{RCV}_k(I(j)) \ \wedge \ CL_k)) \ \Rightarrow \ \neg \Diamond L_j) \quad .$$

The idea of the proof is to show that i can not be succeeded by a smaller process, j say $(i>j)$, as there is always a better candidate or leader process than j that receives $I(j)$—and thus prevents j of becoming a leader.

In order for i, $i>j$, to become a leader i has transmitted $I(i)$. So, i has left the start state before j becomes a candidate. From the following statement which is proven below

Lemma 4.59

$$(\forall j :: (\exists k : j < k : \text{SEND}_k(I(k))) \ \Rightarrow \ \Box (\exists k : j < k : CL_k)) \quad ,$$

we infer that there is still a better process than j, k say, for which CL_k holds. This process receives $I(j)$ and will prevent j of becoming a leader (according to 4.58). This contradicts with j being a successor of i and completes the proof.

It remains to prove lemma 4.59. From the protocol description we infer that after the sending of an I-message the sending process is in either the candidate or leader state. Formally,

Property 4.60

$$(\forall k :: \text{SEND}_k(I(k)) \ \Rightarrow \ \odot \ CL_k) \quad .$$

Moreover, we have that candidates and leaders leave their (combined) state if and only if they receive an I-message with an identity larger than their own identity.

Property 4.61

$$(\forall k :: CL_k \ \Rightarrow \ (\Box \neg (\exists m : k < m : \text{RCV}_k(I(m))) \Leftrightarrow \Box CL_k)) \quad .$$

From property 4.60 we infer that at the next moment process k, $k>j$, transmits $I(k)$, there is a better candidate or leader than j. Furthermore, from 4.61 we infer that as candidates and leaders can only be forced to a state different from leader and candidate by better congeners (as they only leave their state on receipt of $I(m)$ with $m>k$, and as I-messages are only sent by processes that are either candidate or leader) 4.59 holds. This completes the proof of P4. (**End of Proof P4**.)

$$\text{P2} \quad : \quad \Diamond (\exists i :: L_i) \quad .$$

Proof: The proof of this requirement is rather straightforward. As continuously enabled transitions can not be ignored indefinitely (weak fairness assumption) each process in the start state becomes a candidate eventually:

Property 4.62

$$(\forall\, i :: S_i \;\Rightarrow\; \Diamond\, C_i) \quad .$$

Moreover, according to 4.44 a process sends an I-message so as to become a candidate. Consequently, each process sends an I-message sooner or later. Now consider the process with the maximum identity, process $\max_{id}$, say. Due to the finiteness of the set Id this process exists. (We like to stress that the finiteness of Id is crucial for the correctness of Protocol 2, whilst for the correctness of Protocol 1 this is irrelevant.) Once, this process transmits its I-message and becomes a candidate. As there is no 'better' process that can reply it follows from assumptions 4.33 and 4.34 that process $\max_{id}$ can perform its timeout and becomes a leader. Thus, we have that process $\max_{id}$ becomes a leader sooner or later. Furthermore, since leaders can only be succeeded by better processes (see P4), we have

Property 4.63

$$(L_{\max_{id}} \;\Rightarrow\; \Box\, L_{\max_{id}}) \quad .$$

Thus we conclude

Lemma 4.64

$$\Diamond\, L_{\max_{id}} \quad ,$$

which directly implies P2.
(**End of Proof P2**.)

$$\text{P3} \quad : \quad (\forall\, i :: L_i \;\wedge\; (\exists\, j : i < j : \neg\, L_j) \;\Rightarrow\; \Diamond\, \neg\, L_i) \quad .$$

Proof: The idea is to prove P3 along similar lines as in the previous section by first reformulating P3 using $\neg\, L_j \equiv S_j \;\vee\; C_j \;\vee\; F_j$. Once a process becomes failed it remains failed forever (4.47). A process only becomes failed after receipt of an I-message with a larger identity. This follows from (the stronger):

Property 4.65

$$(\forall\, i :: \Box\, \neg\, (\exists\, j : i < j : \mathrm{RCV}_i(I(j))) \;\Leftrightarrow\; \Box\, \neg\, F_i) \quad .$$

From lemma 4.59 and property 4.65 we conclude:

Property 4.66

$$(\forall\, i :: F_i \;\Rightarrow\; (\exists\, j : i < j : CL_j)) \quad ,$$

or, using P1:

Property 4.67

$$(\forall\, i, j : i < j : L_i \;\wedge\; F_j \;\Rightarrow\; (\exists\, k : j < k : C_k)) \quad .$$

Note that it is no longer guaranteed that failed processes are always smaller than the leader process (like in the previous protocol). This is due to the fact that in Protocol 1 only the leader process may force candidates to become failed, whereas in Protocol 2 also candidates may force other candidates to become failed.

From 4.62 we deduce that each process becomes a candidate at some time. Therefore, we may refine P3 (as for Protocol 1) into

Lemma 4.68

$$(\forall\, i :: L_i \;\wedge\; (\exists\, j : i < j : C_j) \;\Rightarrow\; \Diamond \neg L_i) \quad .$$

It remains to establish lemma 4.68. This follows rather straightforward. Assume $L_i \wedge C_j \wedge i<j$. According to 4.44 j has transmitted $I(j)$ so as to become a candidate. This message is received by i when either L_i or F_i holds (otherwise i would not have become a leader). In case L_i, $\Diamond \neg L_i$ follows directly from property 4.49. For F_i we already have $\neg L_i$. This completes the proof of P3.
(**End of Proof P3**.)

Likewise for Protocol 1, it remains to verify that no unspecified receptions can occur. As there is only one message type involved, and as corresponding transitions exist for this message type (for all possible parameters) in all states, and as processes do not receive their own transmitted messages it is evident that no unspecified receptions are possible. This completes the correctness proof of Protocol 2.

4.4 Verification of Protocol 3

The purpose of this section is to prove that Protocol 3 satisfies requirements Q1 through Q6, and that no unspecified receptions can occur. We take a similar approach as in the previous sections.

Like for the previous protocol the timeout mechanism plays a crucial role in establishing the correctness of Protocol 3 with respect to requirements Q1 through Q6. We take as a starting-point the semantics of the timeout mechanism as defined in the previous section (cf. assumptions 4.33 and 4.34).

4.4.1 Timeout Properties

In Protocol 2 a process is only a candidate once (according to (4.36)) and as a timeout can appear at most once the association between, for instance, C_i and $\textsc{timeout}_i$ in a statement like $C_i \wedge \Diamond\, \textsc{timeout}_i$ is unique: the timeout that eventually will occur is the timeout used by i to leave the candidate state referred to by statement C_i. Due to the intrinsic recursive behaviour of Protocol 3 such is no longer true. When stating, for instance, $C_i \wedge \Diamond\, \textsc{timeout}_i^p$ there is no formal relation between the first and second conjunct: process i may be a candidate for a while, leave this state and become a candidate again and then leaving this state on $\textsc{timeout}_i^p$. Stating C_i referring to the first period in the candidate state has no relation at all with $\textsc{timeout}_i^p$. In order to establish such a relation the idea is to refer to the $I(i)$ message on which 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 repeat the timeout properties and reformulate some of them when necessary.

Property 4.69

$$(\forall\, i :: \textsc{timeout}_i \ \Rightarrow\ C_i) \quad .$$

Once a process enters the candidate state by transmission of $I^p(i)$ and the corresponding timeout occurs eventually (i.e. $\Diamond\,\textsc{timeout}_i^p$) 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 4.70

$$(\forall\, i :: \textsc{send}_i(I^p(i)) \ \wedge\ \odot\, C_i \ \wedge\ \Diamond\,\textsc{timeout}_i^p \ \Rightarrow\ C_i\,\mathcal{U}\,\textsc{timeout}_i^p) \quad .$$

As in Protocol 2, a candidate i becomes failed on receipt of $I(j)$ with $i<j$,

Property 4.71

$$(\forall\, i :: C_i \ \wedge\ (\exists\, j : i < j : \textsc{rcv}_i(I(j))) \ \Rightarrow\ \odot\, F_i) \quad ,$$

From properties 4.70 and 4.71 we infer

Property 4.72

$$\begin{aligned}&(\forall\, i :: \textsc{send}_i(I^p(i)) \ \wedge\ \odot\, C_i \ \wedge\ \Diamond\,\textsc{timeout}_i^p \\ &\quad\Rightarrow\ \neg\,(\exists\, j : i < j : \textsc{rcv}_i(I(j)))\,\mathcal{U}\,\textsc{timeout}_i^p) \quad .\end{aligned}$$

4.4.2 Proof of requirements

We now start with proving the requirements one by one. The first proof obligation is:

$$\mathrm{Q1} \quad : \quad (\exists\, i :: L_i \ \Rightarrow\ (\forall\, j : i \neq j : \neg\, L_j)) \quad .$$

Proof: Following an analogous reasoning as for the proof of P1 for Protocol 2 we deduce that the interesting case to prove is

[5]We remark that another possibility would be to equip the C_i predicates with a number as the $\textsc{timeout}_i^p$ 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.

Lemma 4.73

$$(\exists\, i :: \text{TIMEOUT}_i \;\Rightarrow\; (\forall\, j : i \neq j : \neg\, L_j)) \quad .$$

According to 4.69 a timeout can only occur when a process is in the candidate state. A process only becomes a candidate after sending an I-message.

Property 4.74

$$(\forall\, i :: C_i \;\Rightarrow\; \blacklozenge\, \text{SEND}_i(I(i))) \quad .$$

Similarly to the proof of P1 in the previous section the crux of our proof is

Lemma 4.75

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

We now prove lemma 4.75 for each possible state of the recipient of message $I^p(i)$, process j say, given that i becomes a leader once (i.e. $\Diamond\, \text{TIMEOUT}_i^p$).

First consider process j to be either leader, failed, or candidate. For convenience let CLF_j denote $C_j \;\vee\; L_j \;\vee\; F_j$. From the protocol description we immediately infer:

Property 4.76

$$(\forall\, i, j :: CLF_j \;\wedge\; \text{RCV}_j(I(i)) \;\Rightarrow\; (j < i \Rightarrow\; \odot\; F_j) \;\wedge\; (j > i \Rightarrow \text{SEND}_j(I(j)))\;) \quad .$$

Using 4.34, 4.72, and 4.76 we obtain

Lemma 4.77

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

In contrast with Protocol 2, we can not directly conclude 4.75 for candidate, leader, and failed processes from lemma 4.77: in the previous protocol a failed process remains failed indefinitely, whereas —due to its recursive behaviour— in Protocol 3 this is not the case.

So, we have to prove that although process j did not reply on $I^p(i)$ it can not be a leader when TIMEOUT_i^p holds. From 4.76 and 4.77 we infer that, given $\Diamond\, \text{TIMEOUT}_i^p$, we only have to consider processes j for which $j{<}i$. According to 4.76 j becomes failed on receipt of $I^p(i)$. It can only become a leader by transmitting $I^q(j)$ on becoming a candidate. As process i is still being a candidate, according to property 4.70, j is not able to become a leader before i is becoming a leader: j has to wait for i's reply (see timeout semantics) and as $j{<}i$ process i will reply on receipt of $I^q(j)$ thus preventing j becoming a leader. So, we conclude

Lemma 4.78

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

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 recovering j can not become a leader before i is becoming a leader. Note that due to 4.70 i does not crash before becoming a leader. So, crashes of i do not have to be taken into account.

Finally, consider process j to be either start or dead on the moment of processing $I^p(i)$. Let SD_j denote $S_j \ \vee \ D_j$. From the protocol it immediately follows that start and dead processes ignore all messages.

Property 4.79

$$(\forall\, i :: S_i \ \wedge \ \mathrm{RCV}_i(m_1) \ \Rightarrow \ \odot\, S_i \ \wedge \ \neg\, \mathrm{SEND}_i(m_2)) \quad ,$$

Property 4.80

$$(\forall\, i :: D_i \ \wedge \ \mathrm{RCV}_i(m_1) \ \Rightarrow \ \odot\, D_i \ \wedge \ \neg\, \mathrm{SEND}_i(m_2)) \quad ,$$

so j ignores $I^p(i)$. Now the same case analysis as in the proof of P1 of Protocol 2 for start processes can be made and by similar arguments it can be proven that

Lemma 4.81

$$(\forall\, i, j :: SD_j \ \wedge \ \mathrm{RCV}_j(I^p(i)) \ \wedge \ \Diamond\, \mathrm{TIMEOUT}_i^p \ \Rightarrow \ \Box\,(\mathrm{TIMEOUT}_i^p \Rightarrow \neg\, L_j)) \quad .$$

For the sake of brevity we here omit this case analysis. Again, when considering the crash of process j it can also be verified rather easily that after recovering j can not become a leader before i becomes a leader.

From lemmata 4.78 and 4.81 we conclude 4.75. This completes the proof of Q1.
(**End of Proof Q1**.)

$$\mathrm{Q2} \quad : \quad \Diamond\,(\exists\, i :: \Box\,(\neg\, D_i \ \wedge \ (\forall\, j : i < j : D_j))) \ \Rightarrow \ \Box\,\Diamond\,(\exists\, i :: L_i) \quad .$$

Proof: Consider the process with the maximum identity, i' say, for which $\Diamond\,\Box\,(\neg\, D_{i'} \wedge$ $(\forall\, j : i' < j : D_j))$ 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

Lemma 4.82

$$\Diamond\, L_{i'} \quad ,$$

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, $(\forall\, k : i' < k : \Box\, D_k)$. 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

Assumption 4.83

$$\mathcal{I} \;\Rightarrow\; \Diamond (\forall\, k : i' < k : m_k \notin Q_{i'} \;\wedge\; \Box\, D_k) \quad ,$$

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,

Property 4.84

$$(\forall\, i :: \mathcal{J}\; \neg\, L_i \;\Rightarrow\; D_i \;\vee\; \blacklozenge(\exists\, j : i < j : \mathrm{RCV}_i(I(j)))\,) \quad .$$

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.

Property 4.85

$$(\forall\, i, k :: \mathrm{SEND}_i(I(k)) \;\Rightarrow\; i = k) \quad .$$

Furthermore, crashed processes do not transmit messages. That is,

Property 4.86

$$(\forall\, k :: \mathrm{SEND}_k(m) \;\Rightarrow\; \neg\, D_k) \quad .$$

Using 4.83 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:

Lemma 4.87

$$(L_{i'} \;\wedge\; (\forall\, k : i' < k : \Box\, D_k \;\wedge\; m_k \notin Q_{i'}) \;\Rightarrow\; \Box\, L_{i'}) \quad .$$

Secondly, we consider the case that i' is not a leader. Recall that 4.83 holds. From the protocol specification we directly infer that processes that will never crash and are not leader (yet) will become a candidate once.

Property 4.88

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

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 4.33 and 4.34 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. This concludes the proof of Q2.
(**End of Proof Q2**.)

$$\text{Q3} \quad : \quad (\forall\, i :: \neg\,(L_i \ \wedge \ D_i)) \quad .$$

Proof: This follows directly from the definition of finite state machines, where a process can only be in one state 'at a time'.
(**End of Proof Q3**.)

$$\text{Q4} \quad : \quad (\forall\, i,j :: L_i \ \wedge \ \neg\, D_j \ \wedge \ i < j \ \Rightarrow \ \Diamond\, \neg\, L_i \ \vee \ \Diamond\, D_j) \quad .$$

Proof: Assume $L_i \wedge \neg\, D_j \wedge i<j$. Distinguish between two cases: $\Box\, \neg\, D_j$ and $\Diamond\, D_j$. The latter case corresponds to the second disjunct of the conclusion of Q4. Consider $\Box\, \neg\, D_j$. From (4.88) and Q1 we infer that $\Diamond\, C_j$ holds. According to a similar reasoning as for P3 of the previous protocol we observe that it is sufficient to prove:

Lemma 4.89

$$(\forall\, i,j :: L_i \ \wedge \ C_j \ \wedge \ \Box\, \neg\, D_j \ \wedge \ i < j \ \Rightarrow \ \Diamond\, \neg\, L_i) \quad .$$

It remains to establish lemma 4.89. Assume $L_i \wedge C_j \wedge \Box\, \neg\, D_j \wedge i<j$. According to property 4.74 j has transmitted $I(j)$ so as to become a candidate. This message is processed by i after it became a leader—otherwise the message 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 according to 4.76. This completes the proof of Q4.
(**End of Proof Q4**.)

$$\text{Q5} \quad : \quad (\forall\, i :: \mathcal{J}\, \neg\, L_i \ \Rightarrow \ D_i \ \vee \ \blacklozenge(\exists\, j : i < j : \neg\, D_j)) \quad .$$

Proof: According to 4.84 there are only two possible ways in which a leader can capitulate. First, it may spontaneously crash. This corresponds to the first part of the conclusion of Q5. Secondly, leader i capitulates on receipt of an $I(j)$-message with $i<j$. We prove that this corresponds to the second alternative of the conclusion of Q5. From the communication axioms we have that for all (unique) messages m:

Assumption 4.90

$$(\forall\, i :: \text{RCV}_i(m) \ \Rightarrow \ \blacklozenge(\exists\, j : i \neq j : \text{SEND}_j(m))\ ) \quad .$$

Due to property 4.85 $I(j)$-messages can only be transmitted by process j. Furthermore, crashed processes can not transmit messages (due to property 4.86). Thus, we conclude

Lemma 4.91

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

Using 4.84 this concludes the proof of Q5.
(**End of Proof Q5**.)

$$Q6 \quad : \quad (\forall\, i :: L_i \ \wedge \ \odot \neg L_i \ \wedge \ ((\forall\, k :: \neg L_k) \ \wedge \ \neg D_i)\, \mathcal{U}\, L_j \ \Rightarrow \ i \leq j) \quad .$$

Proof: Assume $L_i \wedge \odot \neg L_i \wedge ((\forall\, k :: \neg L_k) \wedge \neg D_i)\, \mathcal{U}\, L_j$. So, j is the immediate successor of leader i and 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:

Property 4.92

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

So, in case a leader capitulates and does not crash it is either a candidate, leader or failed process. From lemma 4.77 it follows that a process can not become a leader in presence of a better candidate, leader or failed process. This implies that j ($j<i$) can not become a leader when i is still in one of these states, which is, according to the premise $\neg D_i\, \mathcal{U}\, L_j$ and the above property the case. This completes the proof of Q6.
(**End of Proof Q6**.)

The remaining proof obligation is the absence of unspecified receptions. As there is only one message type involved, and 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 no unspecified receptions are possible. This completes the correctness proof of Protocol 3.

5 ACP Specifications

In this section the three protocols of section 3 are specified in ACP. We take the Finite State Machine specifications as a starting-point. The ACP specifications are as close to these specifications as possible.

We will give a specification of all separate processes that play a role (protocol processes, buffers, timers, the transmission medium) *and* of the processes that are built from these separate processes. These are a component (the parallel composition of a protocol process, a buffer process and, if applied, a timer process) and the whole system (the parallel composition of all components and the medium). Preceding the protocol specifications a short introduction to ACP is provided.

5.1 Introduction to ACP

ACP, the Algebra of Communicating Processes, is an axiom based mathematical theory for concurrency. ACP has been applied to a large domain of specification problems, ranging from communication protocols, algorithms for systolic systems and electronic circuits up to architectures for Computer Integrated Manufacturing.

This brief introduction is by no means intended to be complete, but merely gives an intuitive notion of what we are dealing with. For a detailed treatment of ACP we refer to [BW90].

ACP starts from a set of objects, called atomic actions, atoms or steps. Atomic actions are the basic and indivisible elements of ACP. In this introduction they will be represented by the symbols a through f. In ACP all atomic actions are elementary processes. Moreover, we have

- δ, deadlock. Deadlock is the state in which there is no possibility to proceed.
- τ, silent step. τ represents the process terminating after some time, without performing observable actions.

Atomic actions may be parameterised with data. There are no strict syntactical constraints: a_d, a^d and $a(d)$ all three denote the atomic action a, parameterised with the data element d.

Processes, in this introduction denoted by the symbols x, y, z, are generated from atomic actions and process terms by means of *operators*. Process names may also be parameterised with data. The most important operators are:

- $\cdot$, sequential composition or product.
  $x \cdot y$ is the process that executes x first and continues with y upon termination of x.
- $+$, alternative composition or sum.
  $x + y$ is the process that first makes a choice between its summands x and y, and then proceeds with the execution of the chosen summand. In the presence of an alternative, δ is never chosen.
  The construct $\sum_{d \in D} x(d)$ is used for the generalised alternative composition $x(d_1) + x(d_2) + \ldots + x(d_n)$, with $d_1, \ldots, d_n$ the elements of D.

- $\|$, parallel composition or merge.
  $x \| y$ is the process that represents the merged execution of x and y.
  The construct $\|_{d \in D}\, x(d)$ is used for the generalised parallel composition $x(d_1) \| x(d_2) \| \ldots \| x(d_n)$, with $d_1, \ldots, d_n$ the elements of D.

- $|$, communication.
  As stated above, $x \| y$ represents the merged execution of x and y. This means that the first action of this composed process is a first action from x or from y or from both. In the last case the two actions from x and y are part of a *communication* between x and y, also called a *synchronization* of x and y. Such a communication has to be defined explicitly by using the communication operator: $a \mid b = c$ means that c is the action that is the result of the communication between the actions a and b.

- ∂_H , encapsulation.
  $\partial_H(x)$ is the process x without the possibility of performing actions from the set of atomic actions H. Algebraically this is achieved by renaming all atomic actions from H in x into δ.

- τ_I , abstraction.
  $\tau_I(x)$ is the process x without the possibility of observing actions from the set of atomic actions I. This is achieved by renaming all atomic actions from I in x into τ.

- θ , priority.
  $\theta(x)$ is the process in which the choice between alternative actions is made according to an ordering on the atomic actions, defined somewhere else. If, in an alternative composition, two atomic actions may be chosen on which an order relation is defined, only the action with the highest priority will be enabled.

- $\triangleleft\ \triangleright$, conditional process.
  The construct $x \triangleleft c \triangleright y$ denotes a conditional process expression. If the boolean expression c evaluates to *true* the process expression reduces to x. If c evaluates to *false* the process expression reduces to y.

Processes are specified by equations like
$x = a \cdot b + c \cdot (e + f)$
$y = (a \cdot b) \| (c \cdot d)$

"Infinite" processes are specified by one or more *recursive* equations. A simple and meaningless example:
$x = a \cdot y + b \cdot z$
$y = c \cdot z$
$z = d \cdot x + e \cdot y$
Possible execution traces of this process are: $a \cdot c \cdot d \cdot b \cdot e \ldots$, $b \cdot d \cdot b \cdot d \ldots$, $a \cdot c \cdot e \cdot c \cdot d \ldots$.

The executable formal specification language PSF [MV90] is based on ACP for its process part. The definition of data in PSF is based on ASF [BHK89].

5.2 Protocol 1

We start with the specification of Protocol 1 from section 3.2.2. First the protocol process and a local buffer process are specified. We will use the following naming convention for the atomic actions involved in the communication between a protocol process, a buffer process and the medium. The transmission of a message is denoted by $send\_XY^i$. X represents the source and Y represents the destination: P for protocol process, B for buffer process or M for medium process. The superscript i denotes the component id. In the same way the reception of a message is denoted by $read\_XY^i$ and the resulting communication action is denoted by $comm\_XY^i$. ID represents the set of component ids. We consider the size of ID to be fixed and finite. M represents the set of messages: $M = \{I(i), R(i) \mid i \in ID\}$.

Specification of the protocol process of component i:

$$\begin{array}{lcl}
Start^i & = & \sum_{m \in M} read_BP^i(m) \cdot Start^i \\
& + & reset_buffer^i \cdot send_PM^i(I(i)) \cdot Candidate^i \\
Candidate^i & = & \sum_{j \in ID} read_BP^i(I(j)) \cdot Candidate^i \\
& + & \sum_{j \in ID \backslash \{i\}} read_BP^i(R(j)) \cdot \\
& & \quad (send_PM^i(I(i)) \cdot Candidate^i \lhd j < i \rhd Failed^i) \\
& + & read_BP^i(R(i)) \cdot Leader^i \\
Leader^i & = & \sum_{j \in ID} read_BP^i(I(j)) \cdot \\
& & \quad (send_PM^i(R(i)) \cdot Leader^i \lhd j < i \rhd send_PM^i(R(j)) \cdot Failed^i) \\
Failed^i & = & \sum_{m \in M} read_BP^i(m) \cdot Failed^i
\end{array}$$

The local buffer process is specified as a queue of unbounded size. The process $Buffer^i$ is parameterised with a message queue q. The queue operations enq (enqueue), $serve$ and deq (dequeue) need no further explanation. The buffer can be reset by the protocol process. This reset is used in order to prevent the processing of messages enqueued before the component enters the election.

$$\begin{array}{lcl}
Buffer^i(q) & = & \sum_{m \in M} read_MB^i(m) \cdot Buffer^i(enq(m,q)) \\
& + & send_BP^i(serve(q)) \cdot Buffer^i(deq(q)) \lhd q \neq empty_queue \rhd \delta \\
& + & read_buffer_reset^i \cdot Buffer^i(empty_queue)
\end{array}$$

The following communications are defined between a protocol process and its local buffer process:

$$send\_BP^i(m) \mid read\_BP^i(m) = comm\_BP^i(m)$$
$$reset\_buffer^i \mid read\_buffer\_reset^i = buffer\_is\_reset^i$$

A component process consists of the encapsulated merge of the protocol process and the buffer process. One component, say l ($l \in ID$), starts in the leader state:

$$Component^l = \partial_{H_1}(Leader^l \parallel Buffer^l(empty\_queue))$$

The other components start in the start state ($i \in ID \backslash \{l\}$):

$$Component^i = \partial_{H_1}(Start^i \parallel Buffer^i(empty\_queue))$$

Definition of the encapsulation set:

$$H_1 = \{read\_BP^i(m), send\_BP^i(m), reset\_buffer^i, read\_buffer\_reset^i \mid i \in ID, m \in M\}$$

The medium process reads a message from a component and sends this message to all other components, thus modelling a broadcast communication. The set ID contains all the component ids, the set IDS is a variable set of component ids.

$$\begin{aligned} Medium &= \textstyle\sum_{i \in ID, m \in M} read\_PM^i(m) \cdot Medium(ID \backslash \{i\}, m) \\ Medium(IDS, m) &= (\|_{i \in IDS}\ send\_MB^i(m)\ ) \cdot Medium \end{aligned}$$

The following communications are defined between a component and the medium process:

$$send\_PM^i(m) \mid read\_PM^i(m) = comm\_PM^i(m)$$
$$send\_MB^i(m) \mid read\_MB^i(m) = comm\_MB^i(m)$$

The complete system consists of the encapsulated merge of all components and the medium:

$$System1 = \partial_{H_2}((\|_{i \in ID}\ Component^i)\ \|\ Medium)$$

Definition of the encapsulation set:

$$H_2 = \{send\_PM^i(m),\ read\_PM^i(m),\ send\_MB^i(m),\ read\_MB^i(m) \mid i \in ID,\ m \in M\}$$

Remark: at this point we notice an important difference between the execution model of the Finite State Machines of section 3 and the ACP execution model. For a transition in an FSM specification the evaluation of a guard and the related action are considered to be atomic: the receiving of a message and the transmission of a reply on this message together form a single atomic event. However, in ACP these are two separate actions. Due to the arbitrary interleaving model, other actions may come in between these two actions. This difference has some influence on the complexity results of section 3. This is discussed in section 5.6.

5.3 Intermezzo: timeout semantics and ACP – part 1

Protocols 2 and 3 make use of a timer, which may generate a timeout. In this section we will discuss the modelling of a timeout in ACP, related to the protocols investigated here.

For certain classes of protocols the correctness of the protocol does not depend on the moment a timeout is raised in relation to other actions in the protocol. For instance, Sliding Window Protocols are robust with respect to premature timeouts. In an ACP specification of these protocols a timeout is modelled by a non-deterministic choice between a timeout action and other enabled actions, see [Bru91]. Other protocols are not robust with respect to premature timeouts. A classical example is the simple PAR protocol, see [Vaa90]. (PAR stands for Positive Acknowledgement with Retransmission.) From section 3 it may be clear that the Leader Election protocols investigated in this document cannot deal with premature timeouts: a timeout may not be enabled before *all* responses to the initial message are generated and processed.

We distinguish three possible approaches to avoid premature timeouts in ACP. The first two approaches are *action oriented*, the last approach is *data oriented*.

1. A timeout action is not enabled as long as certain actions are enabled. This can be modelled with the priority operator θ. The timeout action gets a lower priority than other actions, application of the priority operator prohibits the timeout as long as one of the actions with a higher priority is enabled. This approach is used in [Vaa90].

2. A timeout action is enabled after the execution of certain actions. These actions serve as a kind of *synchronization* for the timeout. Usually this is the only role of these actions, within the specified protocol these actions do not have any other meaning. Therefore we will call them *sync* actions. In [Vaa90] this approach is shortly mentioned for the PAR protocol, in [vW93] it is applied in a PSF specification of the same protocol.

3. In a data oriented approach a timeout action may be enabled if a certain boolean condition is evaluated to *true*. This condition is based on data parameters of the specified system. This requires a specification where state information is put in the data parameters of the process equation(s).

In the remaining part of this section we will investigate the usefulness of the first two alternatives in the realm of the Protocols 2 and 3. In section 6 Protocol 2 and Protocol 3 are captured in a single recursive equation with data parameters. There we will discuss the usefulness of the third alternative.

Application of the priority operator (alternative 1) implies the definition of a set of orderings on actions in which a timeout of component i gets a lower priority than every action that is related to the reply to the initial message from this component. A reply can be made recognizable by labelling the initial I-message with its source and by attaching the same label to all replies to this message.

Two problems arise when this approach is followed. The first problem has to do with the buffering of incoming messages. When queueing is applied, a message in a queue is only related with the $comm\_BP$ action if it is at the head of the queue, otherwise no enabled actions are related with this message. When smart buffering is applied the message in the buffer may be replaced by a better one. By this replacement the label of a message is lost. This kind of buffering problems can be solved by a more complex labelling of the messages. We will not go into the details of such a solution. The second problem has to do with the fact that if the medium is in use (a message has been transmitted to the medium by a component, but has not been buffered by all other components), a $comm\_PM$ action with a reply message to a component i may temporarily be disabled although the timeout of a component i should still be prohibited by this action.

The second problem can only be solved in a rather crude way by placing more restrictions on a timeout action. The timeout of a component i is given a lower priority than every $comm\_MB$ action in order to prevent a temporary blockade of a $comm\_PM$ action. The timeout is also given a lower priority than every $comm\_BP$ action in order to guarantee that every component has had the possibility to react on a message. Finally, the timeout is given a lower priority than every $comm\_PM$ action from a component with an id higher than i in order to ensure that every reply is received by component i before its timeout is enabled. This leads to the following ordering relations:

$timeout^i < comm\_MB^k(m)$,
$timeout^i < comm\_BP^k(m)$
$timeout^i < comm\_PM^j(m)$, with $m \in M,\ i, j, k \in ID, j > i$.

Labelling of messages is not useful any more. The atomic action $timeout^i$ is the result of a communication between the protocol process and the timer process of a component i, see below.

Application of sync actions for the synchronization of a timeout (alternative 2) can be based on the observation that a timeout is permitted when every other component is in the start state (has not yet entered the election), in the failed state (has already lost the election) or has made a transition from the start state to the candidate state after the transmission of the initial message of the component that is waiting for its timeout (is not expected to prohibit the timeout). It is impossible to identify the last set of components without a substantial expansion of the specification. Therefore it is reasonable to focus on a little bit stronger condition which requires that a timeout action is only enabled when all other components are in the start state or in the failed state. This leads to the addition of an action *send_timeout_enable* in the start state and the failed state of a protocol process. A timer process collects these permissions by communicating *read_timeout_enable* actions. Only when all permissions are given a timeout is enabled. This approach leads to two extra actions in the protocol process and a little bit more complicated timer process.

In the case of our Leader Election protocols there is no clear advantage of one alternative above the other, both have their (dis)advantages. In the specification of Protocols 2 and 3 we have chosen to model the timeout semantics cf. alternative 1: a timeout is enabled if certain other actions are disabled. We see this as more close to the dynamic character of the protocols than alternative 2, where a timeout is only enabled if all other components are not actively participating in the election (any more).

5.4 Protocol 2

We continue with the specification of Protocol 2 from section 3. We will give a new specification of the protocol process itself and of the buffer process. We now will use a "smart buffer" in which only the message with the highest id is kept. A timer process, responsible for the generation of a timeout, is also specified. In this protocol we have one message type, $M = \{I(i) \mid i \in ID\}$. Specification of the protocol process of component i ($i \in ID$):

$$
\begin{aligned}
Start^i &= \textstyle\sum_{j \in ID} read\_BP^i(I(j)) \cdot Start^i \\
&\quad + reset\_buffer^i \cdot send\_PM^i(I(i)) \cdot start\_timer^i \cdot Candidate^i \\
Candidate^i &= \textstyle\sum_{j \in ID} read\_BP^i(I(j)) \cdot \\
&\qquad (send\_PM^i(I(i)) \cdot Candidate^i \lhd j < i \rhd stop\_timer^i \cdot Failed^i) \\
&\quad + read\_timeout^i \cdot Leader^i \\
Leader^i &= \textstyle\sum_{j \in ID} read\_BP^i(I(j)) \cdot \\
&\qquad (send\_PM^i(I(i)) \cdot Leader^i \lhd j < i \rhd Failed^i) \\
Failed^i &= \textstyle\sum_{j \in ID} read\_BP^i(I(j)) \cdot Failed^i
\end{aligned}
$$

The local smart buffer process only stores the message with the highest id. The buffer can be reset by the protocol process. The parameter b is used to keep the message with the highest id stored. The function $max(m_1, m_2)$ takes two messages as input and produces the message with the highest id as output.

$$
\begin{aligned}
Buffer^i &= \textstyle\sum_{m \in M} read\_MB^i(m) \cdot Buffer^i(m) \\
&\quad + read\_buffer\_reset^i \cdot Buffer^i \\
Buffer^i(b) &= \textstyle\sum_{m \in M} read\_MB^i(m) \cdot Buffer^i(max(b, m))
\end{aligned}
$$

$$+\ \ send\_BP^i(b) \cdot Buffer^i$$
$$+\ \ read\_buffer\_reset^i \cdot Buffer^i$$

The local timer process is very simple: when a start signal is received the timer waits for a stop signal. If this signal does not appear, a timeout is sent to the protocol process. The waiting for a stop signal and the transmission of the timeout signal is specified as an alternative composition of two process expressions. We will suppose that no start signal is given while waiting for a stop signal or a timeout. As described in the previous section, the timeout semantics will be modelled with the priority operator, see below.

$$Timer^i \quad = read\_start^i \cdot Timer\_s^i$$

$$Timer\_s^i = read\_stop^i \cdot Timer^i + send\_timeout^i \cdot Timer^i$$

The communications between a protocol process and its local timer are defined as follows:

$$start\_timer^i \mid read\_start^i = timer\_started^i$$
$$stop\_timer^i \mid read\_stop^i = timer\_stopped^i$$
$$send\_timeout^i \mid read\_timeout^i = timeout^i$$

In this protocol we assume that there is no leader at the beginning, so all components are initially in the start state:

$$Component^i = \partial_{H_3}(Start^i \ \ \| \ \ Buffer^i \ \ \| \ \ Timer^i)$$

The encapsulation set is defined as follows:

$$H_3 = H_1 \cup \{start\_timer^i, read\_start^i, stop\_timer^i, read\_stop^i, send\_timeout^i, read\_timeout^i \mid i \in ID\}$$

with H_1 as defined in section 5.2.

The medium process is the same as in the specification of Protocol 1.

The semantics of the timeout are modelled with the priority operator θ. This leads to the following specification of the complete system:

$$System2 = \theta \circ \partial_{H_2}((\|_{i \in ID}\ Component^i) \ \| \ Medium)$$

with H_2 as defined in section 5.2. From section 5.3 we recall the order relations for the priority operator θ:

$timeout^i < comm\_MB^k(m)$,
$timeout^i < comm\_BP^k(m)$,
$timeout^i < comm\_PM^j(m)$, with $m \in M,\ i, j, k \in ID, j > i$.

5.5 Protocol 3

In this protocol components may crash. Such a crash has consequences not only for the protocol process, but also for the local buffer process and the timer process. Therefore all component processes need to be reconsidered. In the specification below we will use a simple model of a component crash:

- Only the protocol process has the possibility to crash. The buffer process and the timer process will simply continue (as far as possible) after a crash of the protocol process.

- The "revival" of a component is modelled by the revival of the protocol process. At its revival this process resets the local timer. The local buffer is reset in the start state, which is entered after the revival.
- In the specification of the protocol process a transition from a state to the Dead state is modelled by the atomic action *crash*. The transition from the Dead state to the start state is modelled by the atomic action *revive*. These actions do not communicate with any action from any other process.

In ACP there is no distinction between *must*-actions and *may*-actions (the solid arrows and the dashed arrows from the Finite State Machine Diagram of Protocol 3 in section 3). The plus operator for alternative composition stands for a non-deterministic choice between the alternatives. So we are not able to model this specific property of a process crash in ACP.

Specification of the protocol process of component i:

$$
\begin{array}{lcl}
Start^i & = & \sum_{j \in ID} read\_BP^i(I(j)) \cdot Start^i \\
 & + & reset\_buffer^i \cdot send\_PM^i(I(i)) \cdot start\_timer^i \cdot Candidate^i \\
 & + & crash^i \cdot Dead^i \\
Candidate^i & = & \sum_{j \in ID} read\_BP^i(I(j)) \cdot \\
 & & \qquad (send\_PM^i(I(i)) \cdot Candidate^i \lhd j < i \rhd stop\_timer^i \cdot Failed^i) \\
 & + & read\_timeout^i \cdot Leader^i \\
 & + & crash^i \cdot Dead^i \\
Leader^i & = & \sum_{j \in ID} read\_BP^i(I(j)) \cdot (send\_PM^i(I(i)) \cdot Leader^i \lhd j < i \rhd Failed^i) \\
 & + & crash^i \cdot Dead^i \\
Failed^i & = & \sum_{j \in ID} read\_BP^i(I(j)) \cdot \\
 & & \qquad (send\_PM^i(I(i)) \cdot start\_timer^i \cdot Candidate^i \lhd j < i \rhd Failed^i) \\
 & + & send\_PM^i(I(i)) \cdot start\_timer^i \cdot Candidate^i \\
 & + & crash^i \cdot Dead^i \\
Dead^i & = & revive^i \cdot reset\_timer^i \cdot Start^i
\end{array}
$$

As in the previous protocol the local smart buffer process only stores the message with the highest identity. A crash of the protocol process is not observed by the buffer. However, after the reception of a reset signal the buffer goes to the initial state again. This is already specified in the specification of the buffer process in the previous section. We will not repeat this specification here.

In the timer process in each state the action *read_timer_reset* is added:

$$Timer^i \quad = read\_start^i \cdot Timer\_s^i + read\_timer\_reset^i \cdot Timer^i$$

$$Timer\_s^i = read\_stop^i \cdot Timer^i + send\_timeout^i \cdot Timer^i + read\_timer\_reset^i \cdot Timer^i$$

Definition of the additional communication between the protocol process and the timer process:

$$reset\_timer^i \mid read\_timer\_reset^i = timer\_is\_reset^i$$

In this protocol we assume that there is no leader at the beginning, so all components are initially in the start state:

$Component^i = \partial_{H_4}(Start^i \;\|\; Buffer^i \;\|\; Timer^i)$

Definition of the encapsulation set:

$H_4 = H_3 \cup \{reset\_timer^i, read\_timer\_reset^i \mid i \in ID\}$

with H_3 defined as before.

The medium process is the same as in the specification of Protocol 1 (and 2). The modelling of the timeout is the same as in the specification of Protocol 2. So we get the following definition of the Leader Election protocol:

$System3 = \theta \circ \partial_{H_2}((\|_{i \in ID}\; Component^i) \;\|\; Medium)$

The set H_2 has been defined before.

5.6 Action atomicity and complexity results

We conclude this section with some remarks about the complexity results of section 3 and the execution model of ACP. The complexity analysis in section 3 is based on the atomicity of the action sequence *event-plus-reaction*, e.g. the reception of a message and the transmission of a reply message. This kind of atomicity is common in Finite State Machine formalisms. However, in ACP read actions and send actions are atomic actions themselves. The interleaving model of ACP allows other actions to be executed between a read action and the consecutive send action. This means that, after reading a message from its buffer, a component may have to wait until some actions from other components have been executed before it transmits a reply message. Compared to the FSM model the finer interleaving execution model of ACP introduces the possibility of a *delayed* reaction of a component process, which means that an extra message buffer is introduced within a component. This has a certain influence on the complexity results as derived in section 3.

With simple buffering (queueing of incoming messages) there will be no difference: the extra message buffer can be regarded as an extension of the component buffer queue. However, when we use a smart local buffer we get different results. In Protocol 1 we now get the same worst case message complexity as in the case *without* smart buffering:

Theorem 5.1 *In the ACP interleaving model* $MC_1^s(N,i) = MC_1^q(N,i) = \frac{1}{2}N^2 + \frac{1}{2}N - 1$.

Proof: if the reply on the I-message with the lowest id is temporarily buffered within the protocol process, a message sequence like in Figure 5 is possible. After the reception of the I-message of component $i+1$ the actual leader i can delay its reaction (the transmission of the R-message) until all other I-messages have been sent. □

For Protocol 2 we get a worst case message complexity that is still $\mathcal{O}(N)$:

Theorem 5.2 *In the ACP interleaving model* $MC_2^s(1) = 1$ *and* $MC_2^s(N) = 4N - 5$ *for* $N > 1$.

Proof: The factor 4 comes from the fact that now every component may buffer *two* messages and so may generate two replies on initial messages from components with a lower id. A third reaction may be generated by another message from these components. Together with the initial message this makes 4. The constant -5 comes from the initial values for $MC_2^s(N)$: $MC_2^s(1) = 1$, $MC_2^s(2) = 3$. These initial values can easily be derived. □

In Protocol 3 the worst case message complexity is the same as in the case without smart buffering:

Theorem 5.3 *In the ACP interleaving model* $MC_3^s(N) = MC_3^q(N) = 2^N - 1$.

Proof: The delayed reaction now implies that every component that is about to send a message to the medium cannot be stopped by a message from a component with a higher id until this message is sent. This leads to a worst case behaviour in which every message invokes a message from all components with a higher id. As we have seen before this leads to an exponential worst case message complexity. □

In complexity theory it is a well-known fact that the underlying machine model has a big influence on the complexity of the algorithm [vEB90]. Finite State Machines and ACP both suppose an underlying parallel machine model. The results above show that the *execution model* of a specification formalism sometimes also has a major influence on the complexity of a distributed algorithm.

6 Verification and Validation in ACP

6.1 Introduction

In this section we will explore the power of ACP in the field of verification and validation of the protocols that have been specified in section 5. With *verification* we refer to an algebraic proof of the correctness of a protocol with respect to a set of requirements. If, for whatever reason, we cannot produce such a proof we may try to *validate* a protocol, e.g. by simulation.

For the verification of the protocols we will put our specifications in the following "normal form":

$$\begin{aligned} P(D) = \quad & a_1 \cdot P(D_1) \lhd c_1 \rhd \delta \\ + \; & a_2 \cdot P(D_2) \lhd c_2 \rhd \delta \\ + \; & \ldots \\ + \; & a_n \cdot P(D_n) \lhd c_n \rhd \delta \end{aligned}$$

D denotes a parameter list. D_i , $1 \leq i \leq n$, denotes the same parameter list with substitutions of data terms for some of the parameters. c_i , $1 \leq i \leq n$, denotes a boolean condition, possibly containing variables from D. If a condition is invariantly *true*, a summand may be written as $a_i \cdot P(D_i)$.

The advantage of this normal form lies in the simplification of the calculations that have to be performed in expanding the merge of several processes to a single equation, which can be used for verification and validation purposes. In ACP calculations are performed according to *axioms*, see [BW90]. Before we turn to these calculations we will give a short introduction to the ACP axioms.

6.2 ACP axioms

The axioms of Table 2 form the axiom system for ACP. As before, x, y and z denote process terms, a and b denote atomic actions. We will use the ACP axioms first of all for the *expansion* of the merge of two or more processes. The axiom for the merge operator in Table 2 expands the merge of two process terms to the alternative composition of three terms. The process term $x \mathbin{⌊\!⌊} y$ (x *leftmerge* y) denotes the merge of x and y with the first action from x. The process term $x \mid y$ denotes the communication (synchronisation) between the processes x and y.

The merge axiom can be generalized to the *expansion theorem* for $n \geq 3$:

$$x_1 \parallel \ldots \parallel x_n = \sum_{1\leq i \leq n} x_i \mathbin{⌊\!⌊} (\|_{1\leq j\leq n, j\neq i} \; x_j) + \sum_{1 \leq i < j \leq n} (x_i | x_j) \mathbin{⌊\!⌊} (\|_{1\leq k \leq n, k\neq i,j} \; x_k)$$

From section 5 it is clear that in the specification of the protocols conditional process expressions are frequently used. In [BB92] several axioms are given for conditional process expressions. From these axioms the following identities can be derived.

Lemma 6.1

1. $x \lhd c \rhd x = x$

2. $x \lhd c \rhd y = x \lhd c \rhd \delta + y \lhd \neg c \rhd \delta$

$x + y = y + x$	
$(x + y) + z = x + (y + z)$	
$x + x = x$	
$(x + y) \cdot z = x \cdot z + y \cdot z$	
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	
$x + \delta = x$	
$\delta \cdot x = \delta$	
$x \parallel y = x \mathbin{⌊\!⌊} y + y \mathbin{⌊\!⌊} x + x \mid y$	
$a \mathbin{⌊\!⌊} x = a \cdot x$	
$(a \cdot x) \mathbin{⌊\!⌊} y = a \cdot (x \parallel y)$	
$(x + y) \mathbin{⌊\!⌊} z = (x \mathbin{⌊\!⌊} z) + (y \mathbin{⌊\!⌊} z)$	
$(a \cdot x) \mid b = (a \mid b) \cdot x$	
$a \mid (b \cdot x) = (a \mid b) \cdot x$	
$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \parallel y)$	
$(x + y) \mid z = x \mid z + y \mid z$	
$x \mid (y + z) = x \mid y + x \mid z$	
$a \mid b = \gamma(a, b)$	if γ defined
$a \mid b = \delta$	otherwise

Table 2: ACP axioms.

3. $(x \lhd c_1 \rhd \delta) \lhd c_2 \rhd \delta = x \lhd c_1 \wedge c_2 \rhd \delta$

Proof: elementary, see [Bru]. □

The axioms in Table 3, also from [BB92], are concerned with the merge and encapsulation of conditional process expressions. In the specification of the Protocols 2 and 3 in section 5 the

$(x \lhd c \rhd y) \mathbin{⌊\!⌊} z = (x \mathbin{⌊\!⌊} z) \lhd c \rhd (y \mathbin{⌊\!⌊} z)$
$(x \lhd c \rhd y) \mid z = (x \mid z) \lhd c \rhd (y \mid z)$
$x \mid (y \lhd c \rhd z) = (x \mid y) \lhd c \rhd (x \mid z)$
$\partial_H(x \lhd c \rhd y) = \partial_H(x) \lhd c \rhd \partial_H(y)$

Table 3: Axioms for communication and conditions.

priority operator θ is used to model the desired timeout semantics. In [BW90] the axioms for this operator are given. In this axiomatization an auxiliary operator is used: the binary *unless* operator, denoted by $\lhd$. In order to avoid any confusion between this operator and the left triangle of a conditional process expression, in this paper we will denote the unless operator as $\lll$. The axioms in Table 4 (from [BB92, BW90]) are concerned with the unless operator and the priority operator and with the distributivity of these operators over a conditional process expression. From lemma 6.1 and the axioms for the priority operator the following identity can be derived for the priority operator and the alternative composition of a finite number of conditional process terms:

$a \lll b = a$	if $\neg(a < b)$
$a \lll b = \delta$	if $a < b$
$x \lll y \cdot z = x \lll y$	
$x \lll (y + z) = (x \lll y) \lll z$	
$x \cdot y \lll z = (x \lll z) \cdot y$	
$(x + y) \lll z = x \lll z + y \lll z$	
$\theta(a) = a$	
$\theta(x \cdot y) = \theta(x) \cdot \theta(y)$	
$\theta(x + y) = \theta(x) \lll y + \theta(y) \lll x$	
$x \lll (y \lhd c \rhd z) = (x \lll y) \lhd c \rhd (x \lll z)$	
$(x \lhd c \rhd y) \lll z = (x \lll z) \lhd c \rhd (y \lll z)$	
$\theta(x \lhd c \rhd y) = \theta(x) \lhd c \rhd \theta(y)$	

Table 4: ACP_θ with conditions.

Lemma 6.2

$$\theta(\textstyle\sum_{1 \le i \le n}(a_i \cdot x_i \lhd c_i \rhd \delta)) = \sum_{1 \le i \le n}(a_i \cdot \theta(x_i) \lhd c_i \wedge \neg(\bigvee_{1 \le j \le n \wedge a_j > a_i} c_j) \rhd \delta)$$

Proof: See [Bru] for a proof with $n = 2$. □

Lemma 6.2 states that for an alternative composition of conditional process terms the priority operator can be "translated" to extra conditions on process terms.

It can be foreseen that, in applying the axioms and rules to the process terms of our protocols, we will meet an impassable problem. ACP has no formal semantics of data, so the evaluation of the boolean conditions in the conditional process expressions (which, in our case, are based on the data parameters of the various processes) cannot be formalized. This means that a formal verification in a strict sense is impossible. In the remainder of this section we will give the expansion of the three protocols to a single recursive equation with conditions and we will discuss some requirements which should be met by these equations. We will shortly discuss the necessarily informal verification of the protocols with respect to these requirements.

6.3 Protocol 1

We start with an adapted specification of the processes involved in Protocol 1. By adding states and conditions we will give a specification of each basic process in the normal form as introduced in section 6.1. The merge of these processes will also be expanded to a single equation in this normal form.

First, we give a specification of the protocol process in the desired normal form. The process $P1^i$ has two parameters: ps^i represents a state of the protocol process, j^i represents a component-id. In this specification we distinguish seven states:

- S: the start state.
- B: the buffer is reset, no initial I-message has been sent yet.

- C: the candidate state, the initial I-message has been sent.
- T: an R-message is received by a candidate, but has not been processed yet.
- L: the leader state.
- R: an I-message is received by a leader, but has not been processed yet.
- F: the failed state.

Four states (S, C, L and F) are well-known from previous specifications. The three other states are added in order to get a specification in the desired normal form.

$$
\begin{array}{rl}
P1^i(ps^i, j^i) = & \sum_{m_p \in M} \mathit{read\_BP}(m_p) \cdot P1^i(ps^i, j^i) \lhd ps^i = S \rhd \delta \\
& + \ \mathit{reset\_buffer}^i \cdot P1^i(B, j^i) \lhd ps^i = S \rhd \delta \\
+ & \mathit{send\_PM}^i(I(i)) \cdot P1^i(C, j^i) \lhd ps^i = B \rhd \delta \\
+ & \sum_{j \in ID} \mathit{read\_BP}^i(I(j)) \cdot P1^i(ps^i, j^i) \lhd ps^i = C \rhd \delta \\
& + \ \sum_{j \in ID \backslash \{i\}} \mathit{read\_BP}^i(R(j)) \cdot P1^i(T, j) \lhd ps^i = C \rhd \delta \\
& + \ \mathit{read\_BP}^i(R(i)) \cdot P1^i(L, j^i) \lhd ps^i = C \rhd \delta \\
+ & \mathit{send\_PM}^i(I(i)) \cdot P1^i(C, j^i) \lhd j^i < i \ \wedge \ ps^i = T \rhd \delta \\
& + \ P1^i(F, j^i) \lhd j^i > i \ \wedge \ ps^i = T \rhd \delta \\
+ & \sum_{j \in ID} \mathit{read\_BP}^i(I(j)) \cdot P1^i(R, j) \lhd ps^i = L \rhd \delta \\
+ & \mathit{send\_PM}^i(R(i)) \cdot P1^i(L, j^i) \lhd j^i < i \ \wedge \ ps^i = R \rhd \delta \\
& + \ \mathit{send\_PM}^i(R(j^i)) \cdot P1^i(F, j^i) \lhd j^i > i \wedge \ ps^i = R \rhd \delta \\
+ & \sum_{m_p \in M} \mathit{read\_BP}^i(m_p) \cdot P1^i(ps^i, j^i) \lhd ps^i = F \rhd \delta
\end{array}
$$

The specification of the local buffer process is the same as in section 5. Only the name has been shortened to $B1^i$ and its data parameter now is denoted by q^i in order to give each component queue a unique name.

$$
\begin{array}{rl}
B1^i(q^i) = & \sum_{m \in M} \mathit{read\_MB}^i(m) \cdot B1^i(\mathit{enq}(m, q^i)) \\
+ & \mathit{send\_BP}^i(\mathit{serve}(q^i)) \cdot B1^i(\mathit{deq}(q^i)) \lhd q^i \neq \mathit{empty\_queue} \rhd \delta \\
+ & \mathit{read\_buffer\_reset}^i \cdot B1^i(\mathit{empty\_queue})
\end{array}
$$

The medium process is also specified in a single conditional equation. The merge of the *send_MB* actions (the broadcast to all components except the sender) is expanded to a sum over IDS, a subset of ID. The name of the process has been shortened to M.

$$
\begin{array}{rl}
M(IDS, m) = & \sum_{i \in ID, m_m \in M} \mathit{read\_PM}^i(m_m) \cdot M(ID \backslash \{i\}, m_m) \lhd IDS = \emptyset \rhd \delta \\
+ & \sum_{i \in IDS} \mathit{send\_MB}^i(m) \cdot M(IDS \backslash \{i\}, m) \lhd IDS \neq \emptyset \rhd \delta
\end{array}
$$

On our way to a specification of the whole system we first will derive a specification of the merge of the local buffer processes and the medium. The sequence of component queues $q^i, i \in ID$, is denoted by the parameter Q.

$$BM1(Q, IDS, m) = \partial_{H_1}((\|_{i \in ID} \ B1^i(q^i)) \ \| \ M(IDS, m))$$

Definition of the encapsulation set:

$H_1 = \{send\_MB^i(m), read\_MB^i(m) \mid i \in ID, m \in M\}$.

The expansion of this process leads to the following equation. In this equation the substitution of a new value X for the old value q^i in the sequence Q is denoted by $Q[X/q^i]$.

Lemma 6.3

$$\begin{aligned}
&BM1(Q, IDS, m) = \\
&\qquad \textstyle\sum_{i \in ID,\, m_m \in M} read_PM^i(m_m) \cdot BM1(Q, ID \backslash \{i\}, m_m) \triangleleft IDS = \emptyset \triangleright \delta \\
&\qquad + \textstyle\sum_{i \in IDS} comm_MB^i(m) \cdot BM1(Q[enq(m, q^i)/q^i], IDS \backslash \{i\}, m) \triangleleft IDS \neq \emptyset \triangleright \delta \\
&\qquad + \textstyle\sum_{i \in ID} send_BP^i(serve(q^i)) \cdot BM1(Q[deq(q^i)/q^i], IDS, m) \triangleleft q^i \neq empty_queue \triangleright \delta \\
&\qquad + \textstyle\sum_{i \in ID} read_buffer_reset^i \cdot BM1(Q[empty_queue/q^i], IDS, m)
\end{aligned}$$

Proof: First we look at the merge of the local buffer processes. We name this process $B1(Q)$:

$B1(Q) = \|_{i \in ID}\, B1^i(q^i)$.

As these processes do not communicate with each other, by applying the expansion theorem and the axioms of section 6.2, this merge expands to the alternative composition of three sums over the set ID:

$$\begin{aligned}
B1(Q) = \;& \textstyle\sum_{i \in ID}(\sum_{m \in M} read_MB^i(m) \cdot B1(Q[enq(m, q^i)/q^i])) \\
+ \;& \textstyle\sum_{i \in ID} send_BP^i(serve(q^i)) \cdot B1(Q[deq(q^i)/q^i]) \triangleleft q^i \neq empty_queue \triangleright \delta \\
+ \;& \textstyle\sum_{i \in ID} read_buffer_reset^i \cdot B1(Q[empty_queue/q^i])
\end{aligned}$$

The process $BM1(Q, IDS, m)$ is equal to the encapsulated merge of the processes $B1(Q)$ and $M(IDS, m)$:

$BM1(Q, IDS, m) = \partial_{H_1}(B1(Q) \;\|\; M(IDS, m)\;)$

By applying the axioms and lemma's of section 6.2 and the definition of the encapsulation operator we get a result that is equal to the process expression as stated in lemma 6.3. We will not show the straightforward calculations leading to this result. □

Next we will derive a linear specification of the encapsulated merge of the process $BM1$ and the protocol processes of all components. The process $S1$ is parameterised with PS (a sequence of individual protocol process states ps^i), J (a sequence of component-ids j^i), Q, IDS and m.

$S1(PS, J, Q, IDS, m) = \partial_{H_2}((\|_{i \in ID}\, P1^i(ps^i, j^i)) \;\|\; BM1(Q, IDS, m))$

Definition of the encapsulation set H_2:

$$\begin{aligned}
H_2 = \{&read_BP^i(m),\; send_PM^i(m),\; read_PM^i(m),\; send_BP^i(m),\; reset_buffer^i, \\
&read_buffer_reset^i \mid i \in ID\;,\; m \in M\}
\end{aligned}$$

The expansion of this process leads to the following equation:

Lemma 6.4

$$\begin{aligned}
&S1(PS, J, Q, IDS, m) = \\
&\quad \textstyle\sum_{i \in ID}(\sum_{m_p \in M} comm_BP^i(m_p) \cdot S1(PS, J, Q[deq(q^i)/q^i], IDS, m) \\
&\qquad\qquad \triangleleft serve(q^i) = m_p \;\wedge\; ps^i = S \triangleright \delta \\
&\qquad + buffer_is_reset \cdot S1(PS[B/ps^i], J, Q[empty_queue/q^i], IDS, m) \triangleleft ps^i = S \triangleright \delta)
\end{aligned}$$

$$
\begin{aligned}
+ \ & \textstyle\sum_{i\,\in\,ID}(comm\_PM^i(I(i)) \cdot S1(PS[C/ps^i], J, Q, ID\backslash\{i\}, I(i)) \lhd IDS = \emptyset \wedge ps^i = B \rhd \delta) \\
+ \ & \textstyle\sum_{i\,\in\,ID}(\sum_{j\,\in\,ID} comm\_BP^i(I(j)) \cdot S1(PS, J, Q[deq(q^i)/q^i], IDS, m) \\
& \qquad\qquad \lhd serve(q^i) = I(j) \ \wedge \ ps^i = C \rhd \delta \\
& \qquad + \textstyle\sum_{j\,\in\,ID\backslash\{i\}} comm\_BP^i(R(j)) \cdot S1(PS[T/ps^i], J[j/j^i], Q[deq(q^i)/q^i], IDS, m) \\
& \qquad\qquad \lhd serve(q^i) = R(j) \ \wedge \ ps^i = C \rhd \delta \\
& \qquad + comm\_BP^i(R(i)) \cdot S1(PS[L/ps^i], J, Q[deq(q^i)/q^i], IDS, m) \\
& \qquad\qquad \lhd serve(q^i) = R(i) \ \wedge \ ps^i = C \rhd \delta) \\
+ \ & \textstyle\sum_{i\,\in\,ID}(comm\_PM^i(I(i)) \cdot S1(PS[C/ps^i], J, Q, ID\backslash\{i\}, I(i)) \\
& \qquad\qquad \lhd IDS = \emptyset \ \wedge \ j^i < i \ \wedge \ ps^i = T \rhd \delta \\
& \qquad + S1(PS[F/ps^i], J, Q, IDS, m) \lhd j^i > i \ \wedge \ ps^i = T \rhd \delta) \\
+ \ & \textstyle\sum_{i\,\in\,ID}(\sum_{j\,\in\,ID} comm\_BP^i(I(j)) \cdot S1(PS[R/ps^i], J[j/j^i], Q[deq(q^i)/q^i], IDS, m) \\
& \qquad\qquad \lhd serve(q^i) = I(j) \ \wedge \ ps^i = L \rhd \delta) \\
+ \ & \textstyle\sum_{i\,\in\,ID}(comm\_PM^i(R(i)) \cdot S1(PS[L/ps^i], J, Q, ID\backslash\{i\}, R(i)) \\
& \qquad\qquad \lhd IDS = \emptyset \ \wedge \ j^i < i \ \wedge \ ps^i = R \rhd \delta \\
& \qquad + comm\_PM^i(R(j^i)) \cdot S1(PS[F/ps^i], J, Q, ID\backslash\{i\}, R(j^i)) \\
& \qquad\qquad \lhd IDS = \emptyset \ \wedge \ j^i > i \ \wedge \ ps^i = R \rhd \delta) \\
+ \ & \textstyle\sum_{i\,\in\,ID}(\sum_{m_p \in M} comm\_BP^i(m_p) \cdot S1(PS, J, Q[deq(q^i)/q^i], IDS, m) \\
& \qquad\qquad \lhd serve(q^i) = m_p \ \wedge \ ps^i = F \rhd \delta \\
+ \ & \textstyle\sum_{i\,\in\,IDS}(comm\_MB^i(m) \cdot S1(PS, J, Q[enq(m, q^i)/q^i], IDS\backslash\{i\}, m) \lhd IDS \neq \emptyset \rhd \delta)
\end{aligned}
$$

Proof: We start with the expansion of the merge of the protocol processes $P1^i(ps^i, j^i)$. We name this process $P1(PS, J)$:

$$P1(PS, J) = \|_{i\,\in\,ID} \ P1^i(ps^i, j^i)$$

These processes have no communicating actions, so the merge expands to the alternative composition of a number of sums over the set ID:

$$
\begin{aligned}
P1(PS, J) = \quad & \textstyle\sum_{i\,\in\,ID}(\sum_{m_p \in M} read\_BP^i(m_p) \cdot P1(PS, J) \lhd ps^i = S \rhd \delta) \\
& \ldots \\
+ \quad & \textstyle\sum_{i\,\in\,ID}(\sum_{m_p \in M} read\_BP^i(m_p) \cdot P1(PS, J) \lhd ps^i = F \rhd \delta)
\end{aligned}
$$

Next we consider the encapsulated merge of $P1(PS, J)$ and $BM1(Q, IDS, m)$:

$$S1(PS, J, Q, IDS, m) = \partial_{H_2}(P1(PS, J) \parallel BM1(Q, IDS, m) \ )$$

As in the proof of lemma 6.3, the axioms of section 6.2 and the definition of the encapsulation operator lead to the result as stated in lemma 6.4. For the sake of brevity the vast amount of calculations that goes with this transformation is not shown. □

We now will turn to the verification/validation of Protocol 1. As stated in section 6.2, it is not possible to give a formal verification of the ACP specification of our protocols. The following is a rather short and necessarily informal discussion of how a verification should look like, apart from the restrictions.

In section 3 four requirements have been given for Protocol 1, stated in temporal logic. Here we recall these requirements in natural language:

P1: "There is always at most one leader".

P2: "There will be infinitely often a leader".

P3: "If a component i is the leader and a component j is participating with $j > i$, then i will capitulate sooner or later".

P4: "If j is the successor of i as a leader, then $j > i$".

In ACP there has been gained a lot of experience in verifying concurrent processes in terms of required process behaviour (required actions). Examples of such verifications can be found in [Bae90]. The requirements P1–P4, as stated above, are primarily state oriented. P1 and P2 are concerned with the state of the system at this moment and in the future. P3 and P4 are concerned with transitions from one state to another. In the ACP specifications in this section state information is kept in the data parameters of the process equation. This leads to requirements that are primarily based on statements about these data parameters. Due to the lack of a formal data semantics, this is where the strictly formal ACP road ends and an informal path of natural language reasoning, based on intuitions, begins. We will walk this path for a short distance.

We capture requirement P1 in the following condition R1:

R1: There exists at most one leader. Stated in terms of the data parameters of the process $S1$: the number of leaders in PS is less than or equal to 1.

Requirement P2 is captured as follows. In the specification of the protocol process $P1^i$ we add a summand to the leader state:

$$P1^i(\ldots) = \ldots + \ldots + is\_leader^i \cdot P1^i(\ldots) \triangleleft ps^i = L \triangleright \delta + \ldots$$

The atomic action is_leader^i does not communicate with any other action from any other process. By giving this action a lower priority than any other action in the system we are certain that it only will be chosen if no other action is possible (any more). If the action is_leader^i is chosen, this means that i is the "final" leader that has won the election. According to P4 (see below) we then have that "there will be infinitely often a leader" (P2). If the final leader is the component with the highest id, together with P1 this also implies P3: all eventual leaders with an $id < max(ID)$ apparently have capitulated. Stated more formally:

R2: $\tau_I \circ \theta(S1(initial\ state)) = RS1$ with $RS1 = \tau \cdot is\_leader^{max(ID)} \cdot RS1$

The priority operator now is used in relation with the is_leader^i actions. The abstraction set I contains all actions, except the action is_leader^i. $max(ID)$ stands for the highest id in the set of participating components.

Requirement P4 can be captured by adding a queue SQ of subsequent leader ids to the data parameters of $S1$ and by requiring that SQ forms a strict increasing row with respect to the ordering on the ids:

R3: The queue SQ of subsequent leaders is strictly increasing with respect to the ordering on the ids.

The "verification" of R1 and R3 should imply the addition of R1 and R3 as extra conditions to every action in $S1$. For every action it has to be proved that these conditions are invariantly true from the beginning. The "verification" of R2 can be performed by calculating the required process equality. We will not try to give an informal proof of R1 – R3. Instead, we will point out two possible ways back to a more formal approach:

- turn to a formalism which has a formal semantics of data as well as processes, e.g. the formalism μCRL ([GP91]). This way out has not been investigated, it is left for future research.
- turn to the executable formal specification language PSF ([MV90]). PSF has a formal semantics of both data (based on ASF [BHK89]) and processes (based on ACP). A PSF specification can be simulated on a computer. In this way we get a *validation* of the protocol, rather than a *verification*. This has been carried out for two specifications of Protocol 1 with a few components. One specification was based on the equation for $System1$ in section 5, the other was based on the equation for $S1$ from this section. A number of simulation runs with both specifications all showed the desired behaviour of the protocol.

6.4 Intermezzo: timeout semantics and ACP – part 2

In section 5.3 three alternatives were stated for the modelling of a non-premature timeout. In this section we will look at the third alternative, the data oriented approach.

Once we have a specification of a protocol in the normal form as given in section 6.1, we can model the timeout semantics by the condition under which the timeout action is enabled: in the summand $\ldots + timeout^i \cdot P(D) \triangleleft c \triangleright \delta + \ldots$ we can formulate the condition c according to the timeout semantics. We distinguish three possible alternatives in formulating this timeout condition.

1. We can base this timeout condition on the conditions that enable the actions that have a higher priority than the timeout. The timeout condition becomes *true* iff all these conditions evaluate to *false*. This is the counterpart of modelling the timeout with the priority operator θ, as discussed in section 5.3. Lemma 6.2 makes a formal translation from priorities on actions to conditions on actions possible.
2. The condition on which a timeout is enabled can be based on conditions which are related with specific states of certain constituent processes. Only if these processes are in the desired state(s) the timeout condition becomes *true*. In a certain sense this is the counterpart of the modelling of the timeout with *sync* actions, as discussed in section 5.3.
3. We can base the condition directly on the desired timeout semantics. This means that we try to find the most accurate translation of the timeout semantics as stated in temporal logic into conditions on the data parameters under which the timeout may be enabled.

Although alternative 3 probably gives the most accurate implementation of the desired timeout semantics, in this section we will work out alternative 1. The reason is twofold. First, in using alternative 3 the specification gets very complicated because of the required labelling of messages: the problems mentioned in section 5.3 can be solved only if messages are labelled with a *set* of ids. The operations required on this set will make the specification too complicated. The second reason is that, by choosing alternative 1, we maintain the same approach as with the specifications given in section 5.

6.5 Protocol 2

We will follow the same line as with Protocol 1: first we will give a single equation for the constituent processes of the protocol, then we will derive an equation for the encapsulated merge of these processes.

The specification of the protocol process $P2^i$ looks very much like the adapted specification of the process $P1^i$ in the previous section. By adding states and conditions we get a single equation with several summands. The process $P2^i$ has two parameters: ps^i represents a state of the protocol process, j^i represents a component-id. In this specification we distinguish eight states:

- S: the start state.
- B: the buffer is reset, no initial I-message has been sent.
- I : the initial I-message has been sent, the timer has not been started yet.
- C: the candidate state, the timer has been started.
- T: an I-message is received by a candidate, but has not been processed yet.
- L: the leader state.
- R: an I-message is received by a leader, but has not been processed yet.
- F: the failed state.

Compared to the states of Protocol 1, only the state I is new.

$$
\begin{array}{rll}
P2^i(ps^i, j^i) = & & \sum_{j \in ID} \mathit{read\_BP}^i(I(j)) \cdot P2^i(ps^i, j^i) \lhd ps^i = S \rhd \delta \\
& + & \mathit{reset\_buffer}^i \cdot P2^i(B, j^i) \lhd ps^i = S \rhd \delta \\
& + & \mathit{send\_PM}^i(I(i)) \cdot P2^i(I, j^i) \lhd ps^i = B \rhd \delta \\
& + & \mathit{start\_timer}^i \cdot P2^i(C, j^i) \lhd ps^i = I \rhd \delta \\
& + & \sum_{j \in ID} \mathit{read\_BP}^i(I(j)) \cdot P2^i(T, j) \lhd ps^i = C \rhd \delta \\
& + & \mathit{read\_timeout}^i \cdot P2^i(L, j^i) \lhd ps^i = C \rhd \delta \\
& + & \mathit{send\_PM}^i(I(i)) \cdot P2^i(C, j^i) \lhd j^i < i \ \wedge \ ps^i = T \rhd \delta \\
& + & \mathit{stop\_timer}^i \cdot P2^i(F, j^i) \lhd j^i > i \ \wedge \ ps^i = T \rhd \delta \\
& + & \sum_{j \in ID} \mathit{read\_BP}^i(I(j)) \cdot P2^i(R, j) \lhd ps^i = L \rhd \delta \\
& + & \mathit{send\_PM}^i(I(i)) \cdot P2^i(L, j^i) \lhd j^i < i \ \wedge \ ps^i = R \rhd \delta \\
& + & P2^i(F, j^i) \lhd j^i > i \ \wedge \ ps^i = R \rhd \delta \\
& + & \sum_{j \in ID} \mathit{read\_BP}^i(I(j)) \cdot P2^i(ps^i, j^i) \lhd ps^i = F \rhd \delta
\end{array}
$$

We transform the specification of the smart buffer process of section 5 into a single equation of the desired normal form by adding a default "empty message". If the buffer contains this message it is considered to be empty. This implies that $max(m, empty\_message) = m$ for all incoming messages. We do not consider this empty message to be an element of the message set M. The name of the buffer process has been shortened to $B2^i$, the name of the stored message is now m^i.

$$
\begin{aligned}
B2^i(m^i) = \quad & \textstyle\sum_{m \in M} read\_MB^i(m) \cdot B2^i(max(m^i, m)) \\
+ \quad & send\_BP^i(m^i) \cdot B2^i(empty\_message) \lhd m^i \neq empty\_message \rhd \delta \\
+ \quad & read\_buffer\_reset^i \cdot B2^i(empty\_message)
\end{aligned}
$$

From the specification of process $P2^i$ it is clear that a local timer will only be started in the I-state, after which it will cause a timeout in the C-state or it will be stopped in the T-state. It will never be started again. Therefore we will not give a re-specification of the timer process, we will leave the timer state implicit in the specification of the forthcoming system. The medium process is the same as in section 6.3.

The route to a linear specification of the complete system is the same as in section 6.3. We will not give all intermediate results, but we will state the final result at once in the following lemma. The process $S2$ is parameterised with PS (a sequence of individual protocol process states ps^i), J (a sequence of component-ids j^i), MS (a sequence of messages m^i, kept in the local buffers; m^i can also be the empty message), IDS (a variable set of ids) and a single message m.

$$
\begin{aligned}
& S2(P, J, MS, IDS, m) = \\
& \qquad \theta \circ \partial_{H_3}(\|_{i \in ID} (P2^i(ps^i, j^i) \parallel B2^i(m^i) \parallel Timer^i) \parallel M(IDS, m))
\end{aligned}
$$

with H_3 as defined in section 5. The expansion of this process equation leads to the following equation. The condition for the enabling of the timeout, TO_COND^i, is derived afterwards.

Lemma 6.5

$$
\begin{aligned}
& S2(PS, J, MS, IDS, m) = \\
& \qquad \textstyle\sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S2(PS, J, MS[empty\_message/m^i], IDS, m) \\
& \qquad\qquad\qquad \lhd m^i = I(j) \wedge ps^i = S \rhd \delta \\
& \qquad\qquad\quad + buffer\_is\_reset \cdot S2(PS[B/ps^i], J, MS[empty\_message/m^i], IDS, m) \lhd ps^i = S \rhd \delta) \\
& \quad + \textstyle\sum_{i \in ID}(comm\_PM^i(I(i)) \cdot S2(PS[I/ps^i], J, MS, ID \backslash \{i\}, I(i)) \lhd IDS = \emptyset \wedge ps^i = B \rhd \delta) \\
& \quad + \textstyle\sum_{i \in ID}(timer\_started^i \cdot S2(PS[C/ps^i], J, MS, IDS, m) \lhd ps^i = I \rhd \delta) \\
& \quad + \textstyle\sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S2(PS[T/ps^i], J[j/j^i], MS[empty\_message/m^i], IDS, m) \\
& \qquad\qquad\qquad \lhd m^i = I(j) \wedge ps^i = C \rhd \delta \\
& \qquad\qquad\quad + timeout^i \cdot S2(PS[L/ps^i], J, MS, IDS, m) \lhd TO\_COND^i \wedge ps^i = C \rhd \delta) \\
& \quad + \textstyle\sum_{i \in ID}(comm\_PM^i(I(i)) \cdot S2(PS[C/ps^i], J, MS, ID \backslash \{i\}, I(i)) \\
& \qquad\qquad\qquad \lhd IDS = \emptyset \wedge j^i < i \wedge ps^i = T \rhd \delta \\
& \qquad\qquad\quad + timer\_stopped^i \cdot S2(PS[F/ps^i], J, MS, IDS, m) \lhd j^i > i \wedge ps^i = T \rhd \delta) \\
& \quad + \textstyle\sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S2(PS[R/ps^i], J[j/j^i], MS[empty\_message/m^i], IDS, m) \\
& \qquad\qquad\qquad \lhd m^i = I(j) \wedge ps^i = L \rhd \delta) \\
& \quad + \textstyle\sum_{i \in ID}(comm\_PM^i(I(i)) \cdot S2(PS[L/ps^i], J, MS, ID \backslash \{i\}, I(i)) \\
& \qquad\qquad\qquad \lhd IDS = \emptyset \wedge j^i < i \wedge ps^i = R \rhd \delta \\
& \qquad\qquad\quad + S2(PS[F/ps^i], J, MS, IDS, m) \lhd j^i > i \wedge ps^i = R \rhd \delta)
\end{aligned}
$$

$$+ \quad \sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S2(PS, J, MS[empty\_message/m^i], IDS, m) \\ \lhd\, m^i = I(j) \wedge ps^i = F \rhd \delta)$$

$$+ \quad \sum_{i \in IDS}(comm\_MB^i(m) \cdot S2(PS, J, MS[max(m^i, m)/m^i], IDS\backslash\{i\}, m) \\ \lhd\, IDS \neq \emptyset \rhd \delta)$$

Proof: by lengthy but straightforward calculations, based on the axioms and lemmas of section 6.2. □

In section 5 the timeout semantics was modelled by the definition of a priority relation between certain actions. In lemma 6.2 the relation between the ordering between actions and conditions in a process expression was stated. From this lemma and the action orderings as given in section 5.3 we derive the following lemma concerning the condition for the enabling of the timeout.

Lemma 6.6

$$TO\_COND^i = (IDS = \emptyset) \wedge \\ \bigwedge_{k \in ID}(\neg(m^k \neq empty\_message \;\wedge\; (ps^k = S \vee ps^k = C \vee ps^k = L \vee ps^k = F))) \wedge \\ \bigwedge_{j \in ID, j \geq i}(\neg(ps^j = B \;\vee\; (j^j < j \;\wedge\; (ps^j = T \vee\; ps^j = R))))$$

Proof: We split the proof in three parts, for each of the three order relations we will derive a condition.

1. $timeout^i < comm\_MB^k(m)$ with $m \in M,\; i, k \in ID$.
In the equation of $S2$ there is only one condition under which a $comm\_MB^k(m)$ action is enabled: $IDS \neq \emptyset$. According to lemma 6.2 this leads to the following condition for the enabling of a timeout:

$$C_1^i = \neg(IDS \neq \emptyset) = (IDS = \emptyset)$$

So the first ordering leads to the condition that the medium must be empty before a timeout is enabled.

2. $timeout^i < comm\_BP^k(m)$ with $m \in M,\; i, k \in ID$.
In the equation of $S2$ there are four conditions under which a $comm\_BP$ action is enabled. In each condition it is required that the buffer holds a certain message which is not equal to the empty message. So the second ordering leads to four conditions for the enabling of a timeout:

$$C_{2a}^i = \bigwedge_{k \in ID}(\neg\; (m^k \neq empty\_message \;\wedge\; ps^k = S))$$
$$C_{2b}^i = \bigwedge_{k \in ID}(\neg\; (m^k \neq empty\_message \;\wedge\; ps^k = C))$$
$$C_{2c}^i = \bigwedge_{k \in ID}(\neg\; (m^k \neq empty\_message \;\wedge\; ps^k = L))$$
$$C_{2d}^i = \bigwedge_{k \in ID}(\neg\; (m^k \neq empty\_message \;\wedge\; ps^k = F))$$

$$C_2^i = C_{2a}^i \;\wedge\; C_{2b}^i \;\wedge\; C_{2c}^i \;\wedge\; C_{2d}^i = \\ = \bigwedge_{k \in ID}(\neg\; (m^k \neq empty\_message \;\wedge\; (ps^k = S \vee ps^k = C \vee ps^k = L \vee ps^k = F)))$$

3. $timeout^i < comm\_PM^j(m)$ with $m \in M,\; i, j \in ID, j \geq i$.
In the equation of $S2$ there are three conditions under which a $comm\_PM^i(m)$ action is enabled. According to lemma 6.2 this leads to three conditions for the enabling of a timeout:

$$C_{3a}^i = \bigwedge_{j \in ID, j \geq i}(\neg(IDS = \emptyset \;\wedge\; ps^j = B))$$

$$
\begin{aligned}
C^i_{3b} &= \textstyle\bigwedge_{j\in ID, j\geq i}(\neg(IDS=\emptyset \ \wedge\ j^j<j \ \wedge\ ps^j=T))\\
C^i_{3c} &= \textstyle\bigwedge_{j\in ID, j\geq i}(\neg(IDS=\emptyset \ \wedge\ j^j<j \ \wedge\ ps^j=R))\\
C^i_3 &= C^i_{3a} \ \wedge\ C^i_{3b} \ \wedge\ C^i_{3c} =\\
&= \neg(IDS=\emptyset) \vee \textstyle\bigwedge_{j\in ID, j\geq i}(\neg(ps^j=B \ \vee\ (j^j<j \ \wedge\ (ps^j=T \vee\ ps^j=R))))
\end{aligned}
$$

Finally we get $TO\_COND^i = C^i_1 \ \wedge\ C^i_2 \ \wedge\ C^i_3$ which, after some boolean calculations, leads to the result as stated. □

The requirements for the verification are the same as for Protocol 1. As with Protocol 1 we halt our investigations of the verification of the protocol at this point. With respect to the validation of the protocol we refer to a number of successful simulation runs of a PSF specification of $S2$. The PSF formalism does not provide the priority operator, so $System2$ from section 5 could not be specified and simulated in PSF.

6.6 Protocol 3

The required specification of Protocol 3 will be derived in a few big steps. First we give a specification of the protocol process, then we will give a specification of the whole system. From section 3 it will be clear that the requirements need special attention. We will discuss a revision of the requirements R1–R3 at the end of this section.

The specification of the protocol process $P3^i$ has the same parameters as $P2^i$: ps^i (protocol state) and j^i (a component-id). In the specification we distinguish eleven states:

- S: the start state.
- B: the buffer is reset, no initial I-message has been sent.
- I : the initial I-message has been sent, the timer has not been started yet.
- C: the candidate state, the timer has been started.
- T: an I-message is received by a candidate, but has not been processed yet.
- L: the leader state.
- R: an I-message is received by a leader, but has not been processed yet.
- F: the failed state.
- X: an I-message is received by a failed process, but has not been processed yet.
- D: the dead state.
- A: the component becomes alive again (the revive action has been executed), the timer has not been reset yet.

Compared to the states of Protocol 2 the last three states are new. In the specification below the transition to the dead state is not added to the process term for each separate state S ... X. Instead, a single summand with the action $crash^i$ is added with the condition $\neg(ps^i = D)$.

$$
\begin{array}{rll}
P3^i(ps^i, j^i) = & & \sum_{j \in ID} read\_BP^i(I(j)) \cdot P3^i(ps^i, j^i) \lhd ps^i = S \rhd \delta \\
& + & reset\_buffer^i \cdot P3^i(B, j^i) \lhd ps^i = S \rhd \delta \\
+ & & send\_PM^i(I(i)) \cdot P3^i(I, j^i) \lhd ps^i = B \rhd \delta \\
+ & & start\_timer^i \cdot P3^i(C, j^i) \lhd ps^i = I \rhd \delta \\
+ & & \sum_{j \in ID} read\_BP^i(I(j)) \cdot P3^i(T, j) \lhd ps^i = C \rhd \delta \\
& + & read\_timeout^i \cdot P3^i(L, j^i) \lhd ps^i = C \rhd \delta \\
+ & & send\_PM^i(I(i)) \cdot P3^i(C, j^i,) \lhd j^i < i \ \wedge \ ps^i = T \rhd \delta \\
& + & stop\_timer^i \cdot P3^i(F, j^i) \lhd j^i > i \ \wedge \ ps^i = T \rhd \delta \\
+ & & \sum_{j \in ID} read\_BP^i(I(j)) \cdot P3^i(R, j) \lhd ps^i = L \rhd \delta \\
+ & & send\_PM^i(I(i)) \cdot P3^i(L, j^i) \lhd j^i < i \ \wedge \ ps^i = R \rhd \delta \\
& + & P3^i(F, j^i) \lhd j^i > i \ \wedge \ ps^i = R \rhd \delta \\
+ & & \sum_{j \in ID} read\_BP^i(I(j)) \cdot P3^i(X, j) \lhd ps^i = F \rhd \delta \\
& + & P3^i(B, j^i) \lhd ps^i = F \rhd \delta \\
+ & & P3^i(B, j^i) \lhd j^i < i \ \wedge \ ps^i = X \rhd \delta \\
& + & P3^i(F, j^i) \lhd j^i > i \ \wedge \ ps^i = X \rhd \delta \\
+ & & crash^i \cdot P3^i(D, j^i) \lhd \neg(ps^i = D) \rhd \delta \\
+ & & revive^i \cdot P3^i(A, j^i) \lhd ps^i = D \rhd \delta \\
+ & & reset\_timer^i \cdot P3^i(S, j^i) \lhd ps^i = A \rhd \delta
\end{array}
$$

The (smart) buffer process and the medium process are the same as in the previous section. As with Protocol 2, we will not give a re-specification of the simple timer process, although for this protocol in each state a reset action has been added.

The process $S3$ has the same data parameters as $S2$ in the previous section. So we get

$$
\begin{array}{l}
S3(PS, J, MS, IDS, m) = \\
\qquad \theta \circ \partial_{H_3}(\|_{i \in ID} \ (P3^i(ps^i, j^i) \ \| \ B2^i(m^i) \ \| \ Timer^i) \ \| \ M(IDS, m))
\end{array}
$$

with H_3 as defined before. The expansion of this equation leads to the following equation. The condition TO_COND^i for the enabling of the timeout is the same as in the previous protocol.

Lemma 6.7

$$
\begin{array}{l}
S3(PS, J, MS, IDS, m) = \\
\qquad \sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S3(PS, J, MS[empty\_message/m^i], IDS, m) \\
\qquad\qquad\qquad \lhd m^i = I(j) \ \wedge \ ps^i = S \rhd \delta \\
\qquad\qquad + buffer\_is\_reset \cdot S3(PS[B/ps^i], J, MS[empty\_message/m^i], IDS, m) \lhd ps^i = S \rhd \delta) \\
\quad + \ \sum_{i \in ID}(comm\_PM^i(I(i)) \cdot S3(PS[I/ps^i], J, MS, ID \backslash \{i\}, I(i)) \lhd IDS = \emptyset \wedge ps^i = B \rhd \delta) \\
\quad + \ \sum_{i \in ID}(timer\_started^i \cdot S3(PS[C/ps^i], J, MS, IDS, m) \lhd ps^i = I \rhd \delta) \\
\quad + \ \sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S3(PS[T/ps^i], J[j/j^i], MS[empty\_message/m^i], IDS, m) \\
\qquad\qquad\qquad \lhd m^i = I(j) \ \wedge \ ps^i = C \rhd \delta \\
\qquad\qquad + timeout^i \cdot S3(PS[L/ps^i], J, MS, IDS, m) \lhd TO\_COND^i \ \wedge \ ps^i = C \rhd \delta) \\
\quad + \ \sum_{i \in ID}(comm\_PM^i(I(i)) \cdot S3(PS[C/ps^i], J, MS, ID \backslash \{i\}, I(i))
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad \triangleleft\, IDS = \emptyset \;\wedge\; j^i < i \;\wedge\; ps^i = T \,\triangleright\, \delta \\
&\qquad + timer\_stopped^i \cdot S3(PS[F/ps^i], J, MS, IDS, m) \triangleleft j^i > i \;\wedge\; ps^i = T \triangleright \delta) \\
+\; & \textstyle\sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S3(PS[R/ps^i], J[j/j^i], MS[empty\_message/m^i], IDS, m) \\
&\qquad\qquad \triangleleft\, m^i = I(j) \;\wedge\; ps^i = L \triangleright \delta) \\
+\; & \textstyle\sum_{i \in ID}(comm\_PM^i(I(i)) \cdot S3(PS[L/ps^i], J, MS, ID\backslash\{i\}, I(i)) \\
&\qquad\qquad \triangleleft\, IDS = \emptyset \;\wedge\; j^i < i \;\wedge\; ps^i = R \triangleright \delta \\
&\qquad + S3(PS[F/ps^i], J, MS, IDS, m) \triangleleft j^i > i \;\wedge\; ps^i = R \triangleright \delta) \\
+\; & \textstyle\sum_{i \in ID}(\sum_{j \in ID} comm\_BP^i(I(j)) \cdot S3(PS[X/ps^i], J[j/j^i], MS[empty\_message/m^i], IDS, m) \\
&\qquad\qquad \triangleleft\, m^i = I(j) \;\wedge\; ps^i = F \triangleright \delta \\
&\qquad + S3(PS[B/ps^i], J, MS, IDS, m) \triangleleft ps^i = F \triangleright \delta) \\
+\; & \textstyle\sum_{i \in ID}(S3(PS[B/ps^i], J, MS, IDS, m) \triangleleft j^i < i \;\wedge\; ps^i = X \triangleright \delta \\
&\qquad + S3(PS[F/ps^i], J, MS, IDS, m) \triangleleft j^i > i \;\wedge\; ps^i = X \triangleright \delta) \\
+\; & \textstyle\sum_{i \in ID}(crash^i \cdot S3(PS[D/ps^i], J, MS, IDS, m) \triangleleft \neg(ps^i = D) \triangleright \delta) \\
+\; & \textstyle\sum_{i \in ID}(revive^i \cdot S3(PS[A/ps^i], J, MS, IDS, m) \triangleleft ps^i = D \triangleright \delta) \\
+\; & \textstyle\sum_{i \in ID}(timer\_is\_reset^i \cdot S3(PS[S/ps^i], J, MS, IDS, m) \triangleleft ps^i = A \triangleright \delta) \\
+\; & \textstyle\sum_{i \in IDS}(comm\_MB^i(m) \cdot S3(PS, J, MS[max(m^i, m)/m^i], IDS\backslash\{i\}, m) \\
&\qquad\qquad \triangleleft\, IDS \neq \emptyset \triangleright \delta)
\end{aligned}
$$

Proof: by lengthy but straightforward calculations, based on the axioms and lemmas of section 6.2. □

The correctness requirements for this protocol were given in section 3, stated in temporal logic formulae. We recall these requirements in natural language:

Q1: "There is always at most one leader".

Q2: "If there is a component that never crashes and all better components are crashed for ever, there will be infinitely often a leader".

Q3: "A component cannot be both leader and crashed".

Q4: "If there is a better living component than the leader, eventually this component will crash or the leader will abdicate".

Q5: "The abdication of a leader is caused by a crash of the leader or the existence of a better living component in the past".

Q6: "If a leader abdicates, but does not crash before a new leader emerges, then the identity of the new leader is equal to or higher than the identity of the old leader".

In the following we will try to give a kind of translation of these requirements to ACP requirements. Q1 is the same as P1 and can be captured by R1, as given with Protocol 1. Q3 is obvious: in the process parameter PS each element ps^i can only have one single value. So no component can be both leader and crashed. The requirements Q2, Q4, Q5 and Q6 all contain statements about the behaviour of the system during (subsequent) moments of time. In ACP the only notion we have in this field is a notion of *fairness*, which guarantees that, under certain circumstances, an action will be chosen sooner or later. As already mentioned in section 5, in ACP no difference exists between *may* and *must* transitions. So, in ACP we can never model a component that never crashes (Q2). In an informal way Q5 is obvious: from the specification of $S3$ it is immediately clear that a leader only abdicates after the reception of a message from a better component or on behalf of a crash. We consider Q4 and Q6 as too complex to handle in an ACP setting. Instead, we present a weakened variant of R2: R2$'$ which states that a leader can be observed infinitely often, when abstracting from all other actions and when an is_leader^i action has a lower priority than all actions concerned with message passing.

R2$'$: $\tau_I \circ \theta(S3(initial\ state)) = RS3$ with $RS3 = \sum_{i \in ID} \tau \cdot is\_leader^i \cdot RS3$

As with the previous protocols we will not try to give any verification of Protocol 3 with respect to the requirements. The protocol has been validated by a number of successful simulation runs of a PSF specification of $S3$.

7 Conclusions

In this paper we have designed, specified and verified a series of dynamic leader election protocols in broadcast networks. From this extensive case study in protocol design 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 leader election currently exists in literature. When considering the protocol's correctness this 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.

The protocols are constructed in a step-wise fashion starting from the formal requirement specification. The step-wise approach aids not only in the clarity and conciseness of the protocols, but also —and more importantly— 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.

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

The use of temporal logic for the specification and verification of communication protocols is well-known for almost a decade (see e.g. [Lam82, HO83, SPE84]). 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 (e.g. the addition of timeouts and presence of two kinds of transitions) than traditionally verified protocols.

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.

A specification of the protocols in ACP contains a complete formal description, not only of the various processes but also of the complete distributed behaviour of the protocols. To this extent ACP has more expressive power than state transition diagrams. The protocols in this paper are too large for manual algebraic verification. Automated verification in a related formalism as μCRL is left for future research. PSF simulation runs of the protocols appeared to be very helpful during the various stages of the protocol design.

In general an algebraic verification in ACP consists of a proof that two ACP specifications define the same process, seen from an appropriate level of abstraction. One specification is considered as the requirement specification, while the other serves as the protocol specification. In some cases, as in our LE protocol, it is very hard to provide a requirement specification in ACP. This is due to the fact that such a specification must contain a description of all possible admitted behaviours. This is the main reason why we were not able to give a complete correctness proof in ACP. Instead, we calculated a normal form, which in general is an important step in most ACP proofs. Further research should point out whether there is a way to obtain a requirement specification for this kind of protocols in ACP, or that this problem is intrinsic to ACP.

We think that a combination of the techniques used in this paper may show adequate to give a correctness proof which is completely formal. This would consist of a specification of the complete system (including the communication media) in ACP, followed by a transformation to a normal form in ACP, on which a verification of the requirements using temporal logic is based. It should be studied how to link ACP and temporal logic formally.

In the first instance the construction of the 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 leader election 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 [MV90] 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.

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. This is left for further study.

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. Finally, 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.

References

[AA88] H.H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *IEEE Transactions on Computers*, 37(4):449–453, 1988.

[AG91] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM Journal on Computing*, 20(2):376–394, 1991.

[Att87] H. Attiya. Constructing efficient election algorithms from efficient traversal algorithms. In J. van Leeuwen, editor, *Distributed Algorithms*, LNCS 312, pages 337–344. Springer-Verlag, 1987.

[AvLSZ89] H. Attiya, J. van Leeuwen, N. Santoro, and S. Zaks. Efficient elections in chordal ring networks. *Algorithmica*, 4(3):437–446, 1989.

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.

[BB92] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods, Proceedings Summer School Marktoberdorf 1991*, pages 273–323. Springer, 1992.

[BD87] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic specification.* ACM Press Frontier Series. Addison Wesley, 1989.

[Bru] J.J. Brunekreef. Data oriented process specification. (To appear in autumn 1993).

[Bru91] J.J. Brunekreef. A formal specification of three sliding window protocols. Technical Report P9102b, Programming Research Group, University of Amsterdam, 1991.

[BW90] J.C.M Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[CR79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 22(5):281–283, 1979.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:634–644, 1974.

[DIM93] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election part 1: Complete graph protocols. (Preliminary version appeared in *Proc. 6th Int. Workshop on Distributed Algorithms*, (S. Toueg et. al., eds.), LNCS 579, 167–180, 1992), 1993.

[Fis91] M.J. Fischer. A theoretician's view of fault tolerant distributed computing. In *Fault-Tolerant Distributed Computing*, LNCS 448, pages 1–9. Springer-Verlag, 1991.

[Geh84] N.H. Gehani. Broadcasting sequential processes. *IEEE Trans. on Software Engineering*, 10(4):343–351, 1984.

[Got92] R. Gotzhein. Temporal logic and its applications–a tutorial. *Computer Networks and ISDN Systems*, 24:203–218, 1992.

[Gou93] M.G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25:969–980, 1993.

[GP91] J.F. Groote and A. Ponse. μCRL: A base for analyzing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings of the 3^{rd} Workshop on Concurrency and Compositionality*, pages 125–130. Universität Hildesheim, 1991.

[GZ86] R. Gusella and S. Zatti. An election algorithm for a distributed clock synchronization program. In *Proc. 6th IEEE Int. Conf. on Distributed Computing Systems*, pages 364–371, 1986.

[HO83] B.T. Hailpern and S. Owicki. Modular verification of computer communication protocols. *IEEE Transactions on Computers*, 31(1):56–68, 1983.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[IKWZ90] A. Itai, S. Kutten, Y. Wolfstahl, and S. Zaks. Optimal distributed t-resilient election in complete networks. *IEEE Transactions on Software Engineering*, 16(4):415–420, 1990.

[KKM85] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. In *Proc. 4th Annual ACM Symp. on Principles of Distributed Computing*, pages 163–174. ACM, 1985.

[KMZ84] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proc. 3rd Annual ACM Symp. on Principles of Distributed Computing*, pages 199–207. ACM, 1984.

[Koy89] R.L.C. Koymans. Specifying message passing systems requires extending temporal logic. In B. Banieqbal (et. al.), editor, *Proc. Colloquium on Temporal Logic and Specification*, LNCS 398, pages 213–223. Springer-Verlag, 1989.

[Lam82] L. Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2:175–206, 1982.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[LeL77] G. LeLann. Distributed systems—towards a formal approach. In B. Gilchrist, editor, *Information Processing (vol. 77) (IFIP)*, pages 155–160. North-Holland, Amsterdam, 1977.

[LMW86] M.C. Loui, T.A. Matsushita, and D.B. West. Election in a complete network with a sense of direction. *Information Processing Letters*, 22:185–187, 1986. (see also *Inf. Proc. Letters*, 28:327, 1988).

[LT88] K.G. Larsen and B. Thomsen. A modal process logic. In *Proc. of 3rd Annual Symp. on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988.

[MB76] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[MNHT89] T. Masuzawa, N. Nishikawa, K. Hagihara, and N. Tokura. Optimal fault-tolerant distributed algorithms for election in complete networks with a global sense of direction. In J.-C. Bermond and M. Raynal, editors, *Distributed Algorithms*, LNCS 392, pages 171–182. Springer-Verlag, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification.* Springer-Verlag, New York, 1992.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, VIII:85–139, 1990.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic specification of communication protocols.* Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[Pet82] G.L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Progr. Lang. Syst.*, 4:758–762, 1982.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[SG87] L. Shrira and O. Goldreich. Electing a leader in a ring with link failures. *Acta Informatica*, 24:79–91, 1987.

[SGS84] F.B. Schneider, D. Gries, and R.D. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–16, 1984.

[Sin91] G. Singh. Efficient distributed algorithms for leader election in complete networks. In *Proc. 11th IEEE Int. Conf. on Distributed Computing Systems*, pages 472–479, 1991.

[SPE84] D.E. Shasha, A. Pnueli, and W. Ewald. Temporal verification of carrier-sense local area network protocols. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 54–65, 1984.

[Vaa90] F.W. Vaandrager. Two simple protocols. In Baeten [Bae90].

[vB78] G. v. Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:361–372, 1978.

[vEB90] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (vol. 1)*. Elsevier Science Publishers, 1990.

[vLT87] J. van Leeuwen and R.B. Tan. An improved upperbound for distributed election in bidirectional rings of processors. *Distributed Computing*, 2:149–160, 1987.

[vW93] J.J. van Wamel. Simple protocols. In Mauw and Veltink [MV93].