Consensus Problem in Wireless Ad hoc Networks: Addressing the Right Issues *

Fatemeh Borran
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
1015 Lausanne, Switzerland
fatemeh.borran-dejnabadi@epfl.ch

Ravi Prakash
University of Texas at Dallas
Department of Computer Science
Texas 75080-3021, U.S.A.
ravip@utdallas.edu

André Schiper
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
1015 Lausanne, Switzerland
andre.schiper@epfl.ch

Abstract

Solving consensus in wireless ad hoc networks has started to be addressed in several papers. Most of these papers adopt system models developed for wired networks. These models are focused towards node failures while ignoring link failures, and thus are poorly suited for wireless ad hoc networks. The HO model, which was proposed recently, does not have this drawback. The paper shows that an existing algorithm and the HO model can be used for multi-hop wireless ad hoc networks, if extended with an adequate "implementation". The meaning of "implementation" will become clear from the paper. The description of the "implementation" is augmented with simulation results that validate the feasibility of our approach and provide better understanding of the behavior of realistic wireless environments.

1. Introduction

Ad hoc networks are self-organizing wireless networks that do not rely on a preexisting infrastructure to communicate. Nodes of such networks have limited transmission range, and packets may need to traverse multiple nodes before reaching their destination. Even if the sender and receiver of a packet do not crash, the packet within a wireless network can be lost due to collisions and channel interference. The problems that were already solved for wired communications many years ago, become new challenges in wireless ad hoc environments. Consensus is one of these problems. The importance of consensus is due to the fact that it is a basic building block for solving several other fault-tolerant distributed problems.

*Research funded by the Swiss National Science Foundation under grant number 200021-111701.

Consensus has been extensively studied in traditional networks with various system models. It is now well known that solving consensus deterministically requires some synchrony assumptions [10]. One option is to assume that the (asynchronous) system eventually becomes synchronous, which is called partial synchrony [9]; another option is to augment the (asynchronous) system with failure detectors [5]. Starting from this background, some papers have considered the consensus problem in ad hoc networks. We comment on these papers in Section 2: basically we believe that the approaches suggested are not adequate. The reason is that these papers essentially adopt system models developed for wired and static networks (sometimes with extensions), and these models are not adequate for ad hoc networks. Indeed, the models for wired networks are strongly biased towards node failures to the detriment of link failures. This bias has its root in the FLP paper [10], which assumes process crashes and reliable links. The bias was later strengthened by the failure detector model [5], which also assumes process crashes and reliable links. The bias is so commonly accepted that it is easily overlooked. However, overlooking the bias results in attempts to use solutions for environments where the bias is acceptable, to environments where the bias is unacceptable. This is the case with ad hoc networks, where assuming that links are reliable is clearly inadequate. One may argue that if reliable links are required to solve a problem then there is no work-around, and reliable links need to be implemented on top of lossy links, even if this is expensive in ad hoc networks. But this is not the case for consensus. We know that consensus can be solved in a model in which the distinction between faulty processes and faulty links completely disappears, namely the HO model [7, 12, 6]. This model has no bias, and is, therefore, well suited to handle transient process and link faults. Not only transient link faults (message losses) are frequent in ad hoc networks, but transient process faults can also occur: consider a wireless device that becomes unavail-

able for a while due to a temporary obstacle.

Having said this, we want to stress that the paper does not propose a new consensus algorithm nor a new model for solving consensus. The goal of the paper is to show that an existing consensus algorithm can be used for ad hoc networks, if extended with an adequate "implementation". The meaning of "implementation" will become clear in the next sections. As suggested above, we believe that the right model for consensus in ad hoc networks is the HO model. Several consensus algorithms have been expressed in this model, see [7]. However, out of these algorithms, only two of them genuinely tolerate message loss:[1] the *One-Third-Rule* (*OTR*) algorithm, and the *Paxos/LastVoting* algorithm (*LastVoting* is basically *Paxos* [13] with minor changes). *OTR* seems not adequate, because of its n-n communication pattern (all processes send messages to all). *Paxos/LastVoting* is based on an 1-n communication pattern (communication only between the leader and the other processes). The paper shows that this 1-n communication pattern can nicely be handled in multi-hop networks without any additional overhead for the routing of messages or for election of the leader process. This "implementation" of *Paxos/LastVoting* is completed with simulation results that validate the feasibility of our approach and provide better understanding of the behavior of realistic wireless environments.

The paper is organized as follows. Section 2 presents an overview of the related work. Section 3 presents the consensus algorithm and HO model. Section 4 describes the implementation details. Simulation results are presented in Section 5. Section 6 concludes the paper.

2. Related work

Several papers have addressed the consensus problem in wireless networks. One of the earliest solution to the consensus problem for a cellular network was proposed by Badache et al. [2]. The solution relies on a traditional fixed infrastructure of Mobile Support Station(MSS), and consensus is basically solved among the MSS using the Chandra-Toueg consensus protocol with the failure detector $\Diamond\mathcal{S}$ [5]. The MSS then propagate the decision to the mobile hosts. The solution does not address mobility.

Chockler et al. [8] developed a grid-based consensus algorithm with locally unknown participants in wireless ad hoc networks. The network is divided into a series of non-overlapping grid squares, where each grid square is assumed to be populated. Every node knows a priori its location in the grid. Single-hop consensus is first run for each grid square and, then, all nodes gossip the local decisions. Once a node has received a value for every grid square, it can decide by applying a deterministic function to the set of values received (which requires that every grid square provides a value). Contrary to this solution, we do not require any clustering algorithm, we do not require nodes to know their position, and we do not modify the medium access control (MAC) layer implementation. Moreover the paper makes strong synchrony assumptions (inter-node communication delay are bounded by known constants), nodes are assumed not to crash in the middle of executing a broadcast instruction, and the model does not assume node recovery after a crash. In other words a rather complex system model is considered, in contrast to our very simple model.

Vollset and Ezhilchelvan [14] propose a family of broadcast protocols to be used for solving consensus using randomization. The communication pattern is n-n. Randomization does not lead to efficient consensus algorithms. Moreover, as pointed out in Section 1, the n-n communication pattern does not seem a good choice for multi-hop ad hoc networks. We believe that our 1-n broadcast-convergecast implementation is much more efficient than the general broadcast protocols proposed here.

Finally, Wu et al. [15], propose a consensus protocol for mobile ad hoc networks based on the failure detector $\Diamond\mathcal{P}$. Wu et al. recognize the problem related to the reliable link assumption, but state that complicated design changes would be needed to enable their solution to work with lossy channels. In addition to the issue of using failure detectors in ad hoc networks, the solution has another weakness. It imposes a two-layer hierarchy on the network, where k "predefined" nodes act as clusterheads. Each mobile node is associated with a clusterhead ($k < n$). The solution tolerates up to f faulty nodes, where $f < minimum(k, n/2)$ ($f < k$ because the solution requires one correct clusterhead). Clusterheads are used to reduce the traffic generated for solving consensus. Note that the assumption of predefined clusterheads seems to be in contradiction with the mobility assumption. However, if clusterheads change during the execution, then agreeing on the clusterheads involves solving consensus, which leads to circularity.

To summarize, we believe that the solutions proposed until now have not taken the best approach for ad hoc networks. We believe that the best approach is to "implement" adequately Paxos/LastVoting in a model that handles process faults and link faults in a uniform way.

3. Consensus and HO model

3.1. Consensus specification

We consider a set Π of processes. The consensus problem over a set $\Pi = \{p_1, p_2, ..., p_n\}$ of processes is defined by the following properties:

[1] A problem in the HO model is solved by an algorithm together with a communication predicate. The communication predicate may hide reliable link requirements.

• *Validity:* Any decision value is the initial value of some process.
• *Uniform Agreement:* No two processes decide differently.
• *Termination:* All processes eventually decide.[2]

3.2. The HO model

For solving consensus, we consider the HO (*Heard-Of*) model defined in [7]. The model is based on (asynchronous) rounds. In a round every process first sends messages, then receives messages, and finally changes its state based on the set of messages received. We use the notation $HO(p, r)$ to denote the set of processes from which a message of round r is received by process p (Heard-Of set). Rounds are communication-closed, meaning that a message sent in round r can only be received in round r. An algorithm expressed in the HO model is completed by a predicate over the collection of heard-of sets $(HO(p, r))_{p\in\Pi, r>0}$. For example, predicate $\forall r > 0, \forall p \in \Pi : |HO(p, r)| > \lfloor n/2 \rfloor$ asserts that every heard-of set is a majority set. Consensus is solved in the HO model by a round-based algorithm together with an HO predicate.

In the HO model, there is no distinction between the non reception of some message m by p due to the crash of the sender q, or due a failure of the link between p and q. The model has no bias: it does not need to distinguish process faults from link faults. This makes the model well suited to handle transient process and link faults [12].

3.3. The *Paxos/LastVoting* algorithm

Several consensus algorithms have been expressed in the HO model, see [7]. Out of these algorithms, the *Paxos/LastVoting* algorithm is the most appropriate one:[3] its message complexity is $O(n)$, and it tolerates rounds r in which $HO(p, r)$ is empty for all p, or there are multiple coordinators. The code is given in Algorithm 1. From here on we call the algorithm simply *LastVoting*.

The algorithm consists a sequence of phases ϕ, where each phase has 4 rounds ($4\phi - 3$ to 4ϕ). Each round r consists of a sending step denoted by S_p^r (sending step of p for round r), and of a state transition step denoted by T_p^r. $Coord(p, \phi)$, which denotes the coordinator of p in phase ϕ, is provided by the "implementation" of Algorithm 1, see Section 4. The "implementation" also provides the messages received from the set $HO(p, r)$.

The proof of Algorithm 1 can be found in [7]. The algorithm is always safe even if there are several coordinators per phase. The liveness of algorithm is ensured by a predicate. The predicate given there is sufficient, but not necessary. Termination can occur with a weaker predicate. For example, some process p_x decides at the end of phase ϕ_0 in which the following properties hold:
• All processes consider the same coordinator c_0 in ϕ_0: $\forall p \in \Pi : Coord(p, \phi_0) = c_0$, and
• For a majority of processes p, including p_x, we have $c_0 \in HO(p, 4\phi_0 - 2)$ and $c_0 \in HO(p, 4\phi_0)$, and
• The coordinator hears from a majority of processes in rounds $4\phi_0 - 3$ and $4\phi_0 - 1$: $|HO(c_o, 4\phi_0 - 3)| > \lfloor n/2 \rfloor$ and $|HO(c_o, 4\phi_0 - 1)| > \lfloor n/2 \rfloor$.

This predicate is enough for the purpose of this paper, whose goal is to show experimentally that a clever "implementation" allows Algorithm 1 to solve consensus. What "implementation" means is discussed in the next section.

Algorithm 1 The *LastVoting* algorithm (code of process p).

```
1:  Initialization:
2:    x_p ∈ V, initially v_p                          /* v_p is the initial value of p */
3:    ts_p ∈ ℕ, initially 0
4:    vote_p ∈ V ∪ {?}, initially ?
5:    commit_p a Boolean, initially false
6:    ready_p a Boolean, initially false

7:  Round r = 4φ − 3:
8:    S_p^r:
9:      if Coord(p, φ) ≠ ⊥ then
10:       send ⟨x_p, ts_p⟩ to Coord(p, φ)
11:   T_p^r:
12:     if p = Coord(p, φ) and number of ⟨ν, θ⟩ received > ⌊n/2⌋ then
13:       let θ̄ be the largest θ from ⟨−, θ⟩ received
14:       vote_p := one x̄ such that ⟨x̄, θ̄⟩ is received
15:       commit_p := true

16: Round r = 4φ − 2:
17:   S_p^r:
18:     if p = Coord(p, φ) and commit_p then
19:       send ⟨vote_p⟩ to all processes
20:   T_p^r:
21:     if received ⟨v⟩ from Coord(p, φ) then
22:       x_p := v
23:       ts_p := φ

24: Round r = 4φ − 1:
25:   S_p^r:
26:     if ts_p = φ then
27:       send ⟨ack⟩ to Coord(p, φ)
28:   T_p^r:
29:     if p = Coord(p, φ) and number of ⟨ack⟩ received > ⌊n/2⌋ then
30:       ready_p := true

31: Round r = 4φ:
32:   S_p^r:
33:     if p = Coord(p, φ) and ready_p then
34:       send ⟨vote_p⟩ to all processes
35:   T_p^r:
36:     if received ⟨v⟩ from Coord(p, φ) then
37:       DECIDE(v)
38:     commit_p := false
39:     ready_p := false
```

[2]Usually termination requires only "correct" processes to eventually decide. However, since we assume a model with transient faults (see below), we consider a different termination property.

[3]*LastVoting* is basically *Paxos* [13] expressed in the HO model with minor changes.

4. Implementation of *LastVoting*

4.1. System model

Wireless network: We consider an asynchronous multi-hop wireless network consisting of set of n nodes.[4] We use the terms *node* and *process* interchangeably. Each node in the network has a single wireless transceiver through which it can communicate with other nodes. Due to a variety of reasons (including background noise, terrain, vegetation, etc.), the maximum distance at which a node's transmission can be successfully received may be less than the upper bound on the communication range. Moreover, this distance may change from one transmission to the next. This is different from the unit-disk graph model, and a more realistic representation of wireless propagation characteristics.

Unreliable links and unpredictable delays: When employing MAC layer broadcast, the transmitter does not necessarily know the identities of all nodes within its communication range. Nor does the transmitter know the subset of nodes that successfully received the message. Broadcast communication satisfies the basic *integrity* and *no-duplication* properties guaranteeing that every received message was previously broadcast, and each message is received at most once. However, it is inherently *unreliable*: the receivers do not send any acknowledgment, and the sender does not make any retry attempts to increase the likelihood of message delivery to neighbors. Though MAC layer unicast is described as being reliable (uses acknowledgments), there is no guarantee that a data frame will be forwarded to the intended neighbor. This is due to two reasons. First, the MAC layer buffer may be full when the message arrives, resulting in a buffer overflow. Second, if an acknowledgment is not received following a transmission, the sender makes only a finite number of retry attempts. If all these retries fail, the frame is silently discarded.

So, we assume that the wireless links are *unreliable* and the message communication delay is *unpredictable*.

Good period: *LastVoting* is always safe. To ensure liveness we assume that, from time to time, the system experiences good periods, during which messages are reliably transmitted with the end-to-end (multi-hop) transmission delay bounded by a known constant δ.[5] Note that this condition is sufficient, but not necessary: the property needs to hold only for a "sufficient" number of messages, see for example the predicate in Section 3.3. However, identifying exactly what "sufficient" means here is extraordinary complex, due to the nature of multi-hop networks. Our pragmatic goal is to show experimentally that termination holds in the runs that are generated, and also to evaluate the efficiency of our algorithm.

[4] Actually n needs only to be an upper bound of the number of nodes.

[5] It would be easy to adapt the algorithm to an unknown δ value.

4.2. Implementation model

Figure 1 shows the overall view of our protocol stack. The uppermost layer corresponds to Algorithm 1. However, the heart of our system is Algorithm 2, which contains the main thread that calls Algorithm 1: *in our implementation the sending step S_p^r and the state transition step T_p^r of Algorithm 1 are functions:*

- The sending step S_p^r of Algorithm 1 is a function $S_p^r(s_p, coord_p)$ that takes as input the round number r, the state s_p, the coordinator $coord_p$, and returns the set of message(s) msg to be sent, together with their destination(s) dst (see Algorithm 2, line 14).
- The state transition step T_p^r of Algorithm 1 is a function $T_p^r(msgs, s_p, coord_p)$ that takes as input the round number r, the set of messages received ($msgs$), the state s_p, the coordinator $coord_p$, and returns the new state ns_p (see Algorithm 2, line 32).

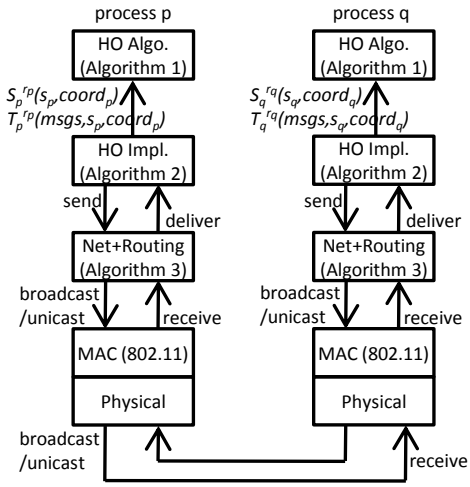


Figure 1. Implementation protocol stack.

Algorithm 2 uses Algorithm 3 as a simple and best-effort broadcast and convergecast algorithm on top of the MAC sub-layer, which typically uses a CSMA/CA-based protocol like IEEE 802.11. Both MAC layer broadcasts and unicasts are used by Algorithm 3: when a message has to be locally broadcast, the MAC layer broadcast primitive is used.

4.3. Algorithm 2: the heart of the system

For every process p, Algorithm 2 has two main roles:

- Elect the coordinator (to be used as a parameter of the S_p^r function)

• For every round r, construct the set of messages received by p (to be used as a parameter of the T_p^r function).

Before discussing these two issues, some general explanations are needed. First, note that Algorithm 2 handles the process state s_p (line 3), the round number r_p (line 7) and the phase number ϕ_p (line 6). Second, Algorithm 2 relies on Algorithm 3 for sending (and receiving) messages (e.g., line 16): the routing implemented by Algorithm 3 is optimized to drop unnecessary messages. Third, Algorithm 2 is designed to ensure fast phase synchronization once a good period has started. Phase synchronization is needed, since when a good period starts, processes can be in different phases (and different rounds). Fast phase synchronization means that processes quickly join the same phase, in order to allow processes to decide. This is done as follows. Each process attaches its current phase ϕ_p and round number r_p to the messages it sends (e.g., line 16). Whenever a process receives a message from some phase $\phi > \phi_p$, it jumps to the first round of that phase (line 31, 12).

Coordinator election: Each process has a priority (e.g. the process identity, line 5), and the process that believes to have the highest priority for some phase ϕ becomes the coordinator for that phase. To be more efficient, the coordinator is restricted to a subset $Contender \subset \Pi$.[6] Initially, every process $p \in Contender$ considers itself as a coordinator (line 4).

At the beginning of each phase ϕ every process p that considers itself to be coordinator sends its identity and priority to all (line 11). This is the only message that Algorithm 2 sends in addition to the message of Algorithm 1. This message is identified by the special round number $null$ (line 11). Each process $p \in \Pi$ that receives a message from phase $\phi \geq \phi_p$ from some process q with higher priority, updates its coordinator and priority (line 22, 28).

After the beginning of a good period, let τ be the time at which the first process starts some phase ϕ_0 (other processes are in earlier phases: with smaller phase numbers). Then at time $\tau + 2\delta$ there is a unique coordinator c for all phases $\geq \phi_0$.[7] However, a unique coordinator c at time $\tau + 2\delta$ is not enough to ensure termination in phase ϕ_0: multiple coordinators between τ and $\tau + 2\delta$ can prevent a decision in phase ϕ_0. So phase $\phi_0 + 1$ is started after 2δ in case c is still in round $4\phi_0 - 3$ (line 40); c is the unique coordinator for the remainder of the good period.

Round message construction: For every round r, Algorithm 2 constructs the set of messages received by process p (to be used as a parameter of the T_p^r function). This is done differently whether $p \in Contender$ or $p \notin Contender$.

[6]The set must be large enough to ensure that it always contains one alive node.

[7]All proofs are in the appendix.

Algorithm 2 Coordinator election and round message construction (code of process p).

```
1:  Initialization:
2:    msgs_p ← ∅                                   /* set of messages received */
3:    s_p ← init_p                                 /* state of the consensus algorithm */
4:    coord_p ← p for p ∈ Contender otherwise ⊥
5:    priority_p ← p's identity for p ∈ Contender otherwise 0
6:    φ_p ← 1                                      /* phase number */
7:    r_p ← 1                                      /* round number */

8:  upon φ_p is updated do
9:    if p ∈ Contender then timer_p ← 0
10:   if p = coord_p then
11:     send (⟨φ_p, null, p, priority_p, −⟩, Π) /* calls Algorithm 3; Π is the
        destination set; message used to elect coordinator */
12:   r_p ← 4φ_p − 3

13: upon r_p is updated do
14:   ⟨msg, dst⟩ ← S_p^{r_p}(s_p, coord_p)        /* calls Algorithm 1 */
15:   if msg ≠ null then
16:     send (⟨φ_p, r_p, p, priority_p, msg⟩, dst)   /* calls Algorithm 3 */

17: upon deliver message ⟨φ, r, q, priority_q, m⟩ do   /* delivered by
    Algorithm 3 */
18:   if φ < φ_p then
19:     ignore message
20:   else
21:     msgs_p ← msgs_p ∪ {⟨φ, r, q, priority_q, m⟩}
22:     if φ = φ_p and priority_q > priority_p then
23:       coord_p ← q
24:       priority_p ← priority_q
25:     if φ > φ_p then
26:       coord_p ← p for p ∈ Contender; ⊥ otherwise
27:       priority_p ← p's identity for p ∈ Contender; 0 otherwise
28:       if priority_q > priority_p then
29:         coord_p ← q
30:         priority_p ← priority_q
31:       φ_p ← φ
32:     ns_p ← T_p^{r_p}({⟨m, q⟩ | ⟨φ_p, r_p, q, −, m⟩ ∈ msgs_p}, s_p, coord_p)
        /* Algorithm 1 is called */
33:     if ns_p ≠ s_p then  /* new state of p is different from its current state */
34:       r_p ← r_p + 1
35:       s_p ← ns_p

36: upon timer_p > 5δ do                           /* timer expires */
37:   coord_p ← p
38:   priority_p ← p's identity
39:   φ_p ← φ_p + 1

40: upon timer_p > 2δ do   /* start new phase if no progress as coordinator */
41:   if p = coord_p and r_p < 4φ_p − 2 then
42:     φ_p ← φ_p + 1

43: upon decide for phase φ_p do
44:   if p = coord_p then
45:     φ_p ← φ_p + 1
```

If $p \notin Contender$, then p does not uses a timer; if $p \in Contender$ then p uses a timer.

Case 1: $p \notin Contender$. In this case p remains in the current round r_p of phase ϕ_p until (1) it receives a message from a larger phase (line 25) or (2) p has received "enough" messages in round r (lines 32 to 35). Note that Algorithm 2 does not know what "enough" means. "Enough" is defined by Algorithm 1: in rounds $4\phi - 3$ and $4\phi - 1$ "enough" is more than $n/2$; in rounds $4\phi - 2$ and 4ϕ "enough" is 1. The solution is for Algorithm 2 to call the T_p^r function whenever a new message is received (line 32): if not enough messages have been received, the T_p^r function does not modify the

state (line 33) and p remains in the same round (in order to wait for more messages).

Case 2: $p \in Contender$. In addition to behaving like an ordinary process (Case 1), p uses a timer, which is reset at the beginning of each phase ϕ_p (line 9). In a good period a round does not take more than δ. So, in addition to the behavior explained under Case 1, p remains in phase ϕ_p until (1) 2δ time units have elapsed (duration of leader election round and round $4\phi - 3$) and p is still in round $4\phi_p - 3$ (line 40), or (2) 5δ time units have elapsed (duration of leader election round and rounds $4\phi - 3$ to 4ϕ) and p is still in phase ϕ_p (line 36).

Optimizations: Algorithm 2 includes two optimizations. The first one is useful when several instances of consensus are running one after the other (e.g. atomic broadcast). When a decision occurs in phase ϕ, the coordinator starts immediately phase $\phi+1$ (line 43) without waiting the timeout for phase ϕ. The second optimization avoids unnecessary coordinator changes. Once some process p is considered to be the coordinator by a majority, it remains the coordinator as long as its messages reach a majority of processes: process $q \in Contender$ that considers p as its coordinator ($priority_q < priority_p$) does not change its coordinator unless its timer expires (line 36).

4.4. The broadcast and convergecast

Algorithm 2 invokes Algorithm 3 when it sends a message in lines 11 and 16. Depending on dst, Algorithm 3 uses diffusion or convergecast in lines 9 and 11: diffusion is used for a message sent by a coordinator (*1 to all*), while convergecast is used for a message sent to the coordinator (*all to 1*). Diffusion messages are identified by the tag MESSAGE (e.g., line 9), while convergecast messages are identified by the tag RESPONSE (e.g., line 11). During diffusion, Algorithm 3 delivers the message that is received for the first time (line 13) to Algorithm 2. During convergecast, the message is delivered only if it reaches its destination (line 21). Algorithm 3 also contributes to an efficient election of the coordinator by discarding messages from contenders that can no more become coordinator.

Diffusion: As all participating nodes are not within communication range of each other, it is not possible for a node to directly communicate with others. Hence, a network-wide message broadcast can be implemented through diffusion. The message source (a coordinator) will broadcast the message locally at the MAC layer (line 9). When node p receives a message from some node q for the first time (line 12), it becomes a child of q (line 15) only if $priority_q > priority_p$ (q wins against p in the election). Then it broadcasts the message at the MAC layer (line 18) except when $priority_q < priority_p$ (q loses against p in the election). When a node receives copies of the same message later, it ignores them. *As a result, an efficient tree rooted at a coordinator is formed.*

Convergecast: The tree constructed during diffusion is used by convergecast, to transport responses to the coordinator, the root of the tree. As a node does not know the identities of all its children it is not possible for the node to determine when it has received responses from all of them. So, a node sends its response to its parent as soon as the node joins the tree. Subsequently, whenever the node receives a response from any child it forwards the received response to its parent.

Algorithm 3 The broadcast and convergecast algorithm (code of process p).

```
 1: Initialization:
 2:   parent_p ∈ Π ∪ {NULL}, initially NULL
 3:   level_p ∈ ℕ, initially 0
 4:   priority_p refers below to the variable priority_p of Algorithm 2

 5: function send (m, dst)                          /* called by Algorithm 2 */
 6:   if dst = Π then
 7:     parent_p := p
 8:     level_p := 1
 9:     locally broadcast ⟨MESSAGE, p, level_p, m⟩
10:   else
11:     unicast ⟨RESPONSE, q, level_p, m⟩ to parent_p

12: upon receive ⟨MESSAGE, root, l, m⟩ from node q with priority
    priority_q for the first time do
13:   deliver ⟨m⟩  /* m delivered to Algorithm 2; after executing lines 17-35 of
      Algorithm 2 execute following lines */
14:   if priority_q > priority_p then
15:     parent_p := q
16:     level_p := l + 1
17:   if priority_q ≥ priority_p then
18:     locally broadcast ⟨MESSAGE, root, level_p, m⟩

19: upon receive ⟨RESPONSE, root, l, m⟩ for the first time do
20:   if p = root then
21:     deliver ⟨m⟩                                /* m delivered to Algorithm 2 */
22:   else
23:     unicast ⟨RESPONSE, root, level_p, m⟩ to parent_p
```

Figure 2 shows an example of broadcast and convergecast protocol in a multi-hop network. During diffusion (tag MESSAGE), since p_2's priority is higher than p_1's, if p_5 receives the message form p_2 before p_1, it ignores p_1's message. Otherwise, it diffuses both, but p_4 becomes its parent and p_2 its grand parent. During convergecast (tag RESPONSE), only path from p_7 to p_2 is followed.

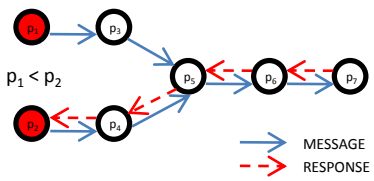


Figure 2. Broadcast *vs.* convergecast.

Gradient-based convergecast: If any node on the path from node p to the root of the tree (i.e., to the coordiantor) is

down, or any link on this path is lossy, p's message may not reach the root. Gradient-based convergecast can increase the probability of responses reaching the root. During diffusion, as a node joins the tree, it sets its level to be one greater than its parent's level (line 16). The root is always at level one (line 8). During convergecast nodes listen to transmissions in the promiscuous mode. If they receive a message from a neighboring node at a higher level they retransmit the message (using MAC layer broadcast). Thus, messages travel from higher level to lower level, with no cyclic forwarding, ultimately reaching the root. Even if the path from the root to a node breaks down after the node has joined the tree, it may be possible for the node's response to reach the root along other gradient-based paths, if such paths exist. This can be done as follows:
1. In line 11, instead of sending the RESPONSE to the parent, locally broadcast the RESPONSE.
2. In line 23, first determine if $l > level_p$. If so, locally broadcast the RESPONSE.

Remark: Note that the underlying network is unreliable. So, whenever a coordinator broadcasts a message there is no guarantee that all the nodes will join the tree and receive the message. Furthermore, messages from all the tree nodes may not reach the root: they may disappear on the way. Yet, the safety property of the *LastVoting* algorithm is never compromised. If the coordinator is able to receive responses from a majority of nodes in round $4\phi - 3$ and, subsequently, acknowledgments from a majority of nodes in round $4\phi - 1$ (not necessarily the same set as in round $4\phi - 3$), it is possible for the coordinator to decide on a value.

5. Simulation

We used JiST/SWANS v1.0.6 [1, 3] wireless network simulator to simulate our algorithm. We consider a $m \times m$ square grid with nodes placed at each intersection as illustrated in Figure 3.

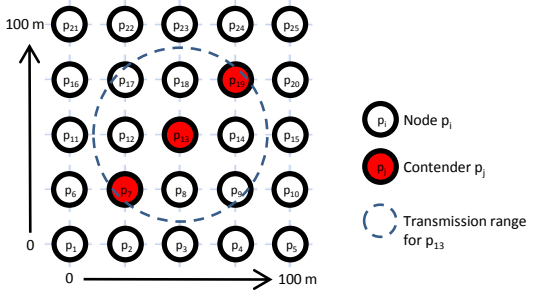


Figure 3. Square grid of size 5×5 in network area $100 \times 100\ m^2$.

We used grid-based placement instead of the random uniform placement only for manageability reasons. For instance, using this placement we can select exactly which nodes belong to the $Contender$ set. Communication between two nodes p_1 and p_2 occurs in an ad hoc manner using unicast/broadcast as defined in the IEEE 802.11b standard [11]. The data rate of the wireless channel is 1 Mbps. All nodes have the same transmission range ($150\ m$). We modify the network area to vary network density and network diameter. Nodes are stationary, except for one case in which we measure the impact of mobility (see Section 5.2.4). We measure the impact of contenders in Section 5.2.3. Each contender starts the algorithm randomly between 0 and 10 milliseconds after simulation start time. The simulation lasts for 100 seconds. Every consensus packet is around 32 bytes. Unless otherwise mentioned, we use the default values defined in the JiST/SWANS simulator.

Note that the IEEE 802.11b MAC layer specification uses CSMA/CA and enforces RTS/CTS/ACK control frames for unicast communication only. Collision control for broadcast is limited to basic collision avoidance carrier sensing, and broadcast is therefore prone to packet collisions. A straightforward approach to reduce collisions is to have nodes wait for a small random amount of time (jitter) before rebroadcasting. Given the consensus algorithm in Section 3.3 and based on broadcast and convergecast protocol (Section 4.4), we are interested in analyzing whether the required liveness condition is provided by Algorithm 2 in wireless ad hoc networks.

5.1. Metrics

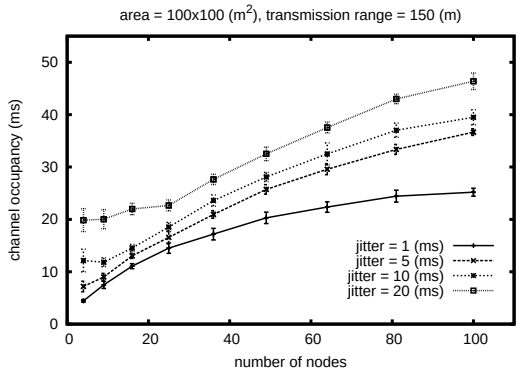
In order to evaluate the performance of *LastVoting* consensus algorithm, several instances of consensus are run one after the other. Each process starts a new instance of consensus with new proposition. A new consensus is started as soon as the decision for current consensus is reached or a message from a later invocation of consensus algorithm is received. In the latter case, the previous decisions can be communicated through piggy-backing.

We have defined two important (and independent) metrics: *consensus latency* and *consensus throughput*. Consensus latency is expressed in terms of average number of phases per consensus from initialization to first decision. Consensus throughput represents how many instances of consensus can be run successfully in simulation time (100 seconds). Note that the time required for one consensus can be calculated from consensus throughput and latency.

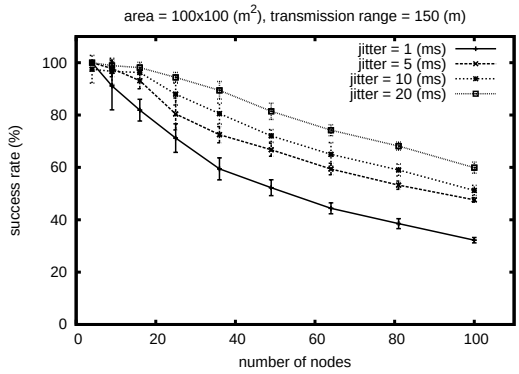
5.2. Results

In this section we present the results of our simulations. We evaluate the performance of our consensus algorithm in

both single and multi-hop networks. There is no process crash or additional packet loss in our simulations.[8] However, to observe the performance of our algorithm in realistic situations, we added a background traffic to the system: every second, each node sends a packet (with the same size as consensus packet) to a random destination. All results of simulations are averaged over 30 independent runs. Due to the many sources of randomness, for instance jitter, the simulation results for ad hoc networks differ from one run to the other. The vertical bars in the graph represent 95% confidence interval for the mean.



(a) Channel occupancy vs. density.



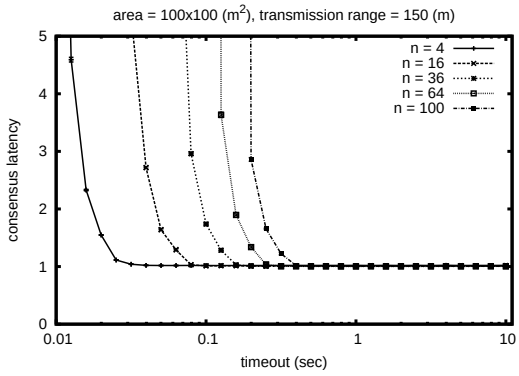
(b) Success rate vs. density.

Figure 4. Impact of density and jitter in channel occupancy and success rate.

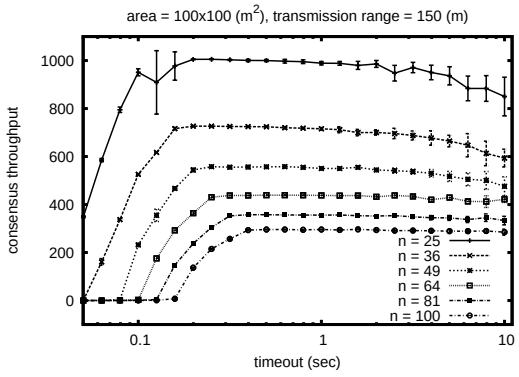
Before running the simulations, we ran a calibration test to examine the behavior of the simulator and our routing algorithm to tune the amount of the jitter. Figure 4(a) shows once a single message is broadcasted, the duration for which the wireless channel remains busy (henceforth, referred to as channel occupancy duration). Note that the same message forwarding algorithm is employed by each node: on receiving a message for the first time, a node rebroadcasts the message after a random wait between 0 and jitter. So, the wireless channel becomes idle either when the message is received by everyone or is completely lost. For instance, for 100 nodes within range of each other, with jitter = 10 ms, channel occupancy is 40 ms. This gives us 80 ms for round-trip time, or 200 ms for one phase of our consensus implementation. Figure 4(b) shows the percentage of nodes that receive the broadcast message. It seems that the value of the jitter is optimal around 10 ms. With 10 ms, at least a majority of processes have received the message and there is almost the same channel occupancy as 5 ms. For the rest of simulations we fix jitter to 10 ms.

[8]The only packet loss is due to collisions and node interferences.

5.2.1. Single-hop scenarios



(a) Consensus latency vs. timeout

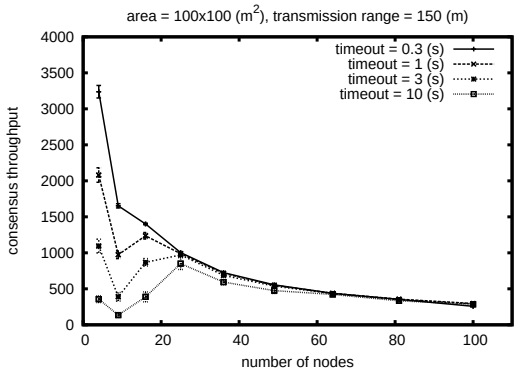


(b) Consensus throughput vs. timeout

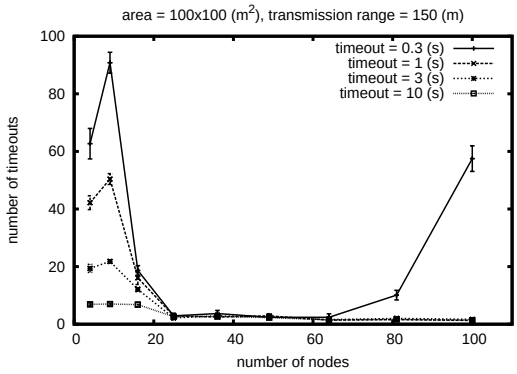
Figure 5. Impact of timeout in consensus latency and throughput in single-hop wireless networks (with single contender).

First, we consider a single-hop network in which all nodes are in communication range of each other. The network area is $100 \times 100\ m^2$. We gradually increased the network density. Only a single node, for example p_1, belongs to the $Contender$ set. We measured the average number of phases per consensus in networks with different node densities (from 4 nodes to 100 nodes) by varying the timeout. The value of timeout refers to 5δ used in Algorithm 2. The ideal value in our scenario is 1 phase per consensus. How-

ever, this value can increase in the presence of packet loss. Figure 5(a) shows how the number of required phases varies with timeout. Logarithmic scales are used in x-axis to better visualize a large range of timeout and emphasize the small timeouts. Beyond a certain value of timeout, the number of phases to terminate consensus remains almost constant (1 phase) as density of the deployment increases. Figure 5(b) shows how consensus throughput varies with timeout for several network densities. Note that the results we have obtained in this simulation based on the timeouts match exactly with our previous results on channel occupancy.



(a) Consensus throughput vs. density



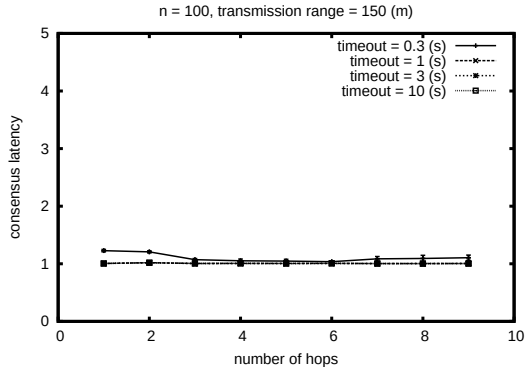
(b) Number of timeouts vs. density

Figure 6. Impact of density in consensus throughput in single-hop wireless networks (with single contender).

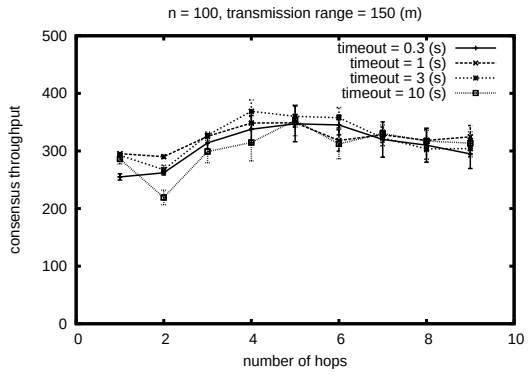
Figure 6(a) shows the impact of density on consensus throughput with different timeouts. In general, by increasing density (number of nodes), the throughput of our algorithm decreases, independent of phase timeout value. This is due to message losses due to increased collisions. The graph shows that there is an optimal value for density. After around 25 nodes, the throughput always goes down. So the algorithm performs less efficiently in the presence of more than 25 nodes per $10000\ m^2$ (single-hop). Although with small number of nodes the throughput is high, the number of timeouts that occur is also high (see Figure 6(b)). For instance, for $n = 4$ the algorithm allows only one message loss while for $n = 100$, 49 losses are allowed in a round (majority set). This explains why for small number of nodes, increasing the timeout reduces performance in Figure 6(a).

5.2.2. Multi-hop scenarios

We now consider multi-hop scenarios where all nodes are not in communication range of each other. To do that we consider 100 nodes distributed in a 10×10 square grid. The transmission range for each node is fixed to $150\ m$. To obtain multi-hop scenarios, we varied the network area from $100 \times 100\ m^2$ (single-hop) to $900 \times 900\ m^2$ (9-hops), and we chose p_1 as the contender ($p1$ is located at the lower left corner of the grid).



(a) Consensus latency vs. area



(b) Consensus throughput vs. area

Figure 7. Impact of number of hops in consensus latency and throughput in multi-hop wireless networks (with single contender).

Figure 7(a) shows the scalability of our algorithm in multi-hop networks. By increasing the network area for 100 nodes, on the one hand we increase the number of hops and on the other hand we decrease the density and, therefore, the probability of message collisions. Figure 7(b) shows the trade-off between number of hops and network density.

From one-hop to four-hops, we decrease the density, so the performance is improved. From six-hops on, since the message must traverse more hops the performance is slightly decreased. So, 100 nodes perform better in five-hops. This gives approximately 20 nodes per hop. This is almost the same conclusion that we had from single-hop scenarios.

5.2.3. Impact of contenders

To see the impact of the contender's position on consensus throughput, we varied the position of the contender from bottom-left corner to the center.[9] We run a Kruskal-Wallis non-paired data test [4] (generalized Wilcoxon Rank Sum test) to determine if the position of the contender influences consensus throughput (null hypothesis: position of the contender does not influence consensus thorughput). The test accepts the null hypothesis with p-value 0.9699. The conclusion is that the throughput of our consensus algorithm is independent of the contender's position. This seems reasonable in single-hop networks. In multi-hop networks, when the contender moves from bottom-left corner to the center of square grid, the number of hops from the contender to the farthest node is reduced while the number of collision is augmented (in center there is 4 times more collision than in corner). So in multi-hop networks, reduced number of hops is compensated by increased number of collisions.

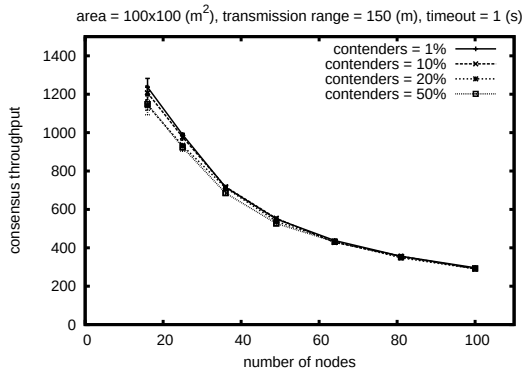


Figure 8. Impact of contenders in consensus throughput in single-hop wireless networks.

In Figure 8, we increased the number of contenders in network of 100 nodes from 1% to 50%. We have fixed timeout to 1 second to eliminate the impact of timeout. It turns out that even the number of contenders has almost not a significant impact on the consensus throughput (p-value = 0.0569 using Kruskal-Wallis test). The explanation is as following: since no process crashes during the simulation, once the process with highest priority is elected as the coordinator, it remains the same as long as the majority of its messages are not lost.

[9] This is enough to explore other possibilities because of the symmetry of square grid.

5.2.4. Impact of mobility

We now measure the impact of node mobility on consensus throughput. We use the random waypoint model with a fixed speed and zero pause time. In this model, nodes select an arbitrary location in the field and move on direct line at constant speed. When they reach the destination, they pick a new destination and so on. Figure 9 shows the robustness of our algorithm against node speed.

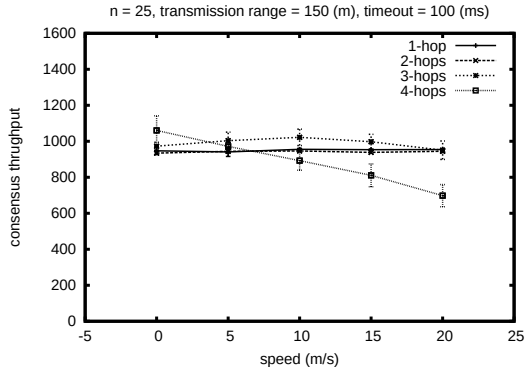


Figure 9. Impact of mobility in consensus throughput.

6. Conclusions

The *Paxos/LastVoting* algorithm expressed in the HO model can potentially solve the consensus problem in wireless mobile networks. *Paxos/LastVoting* is safe by design, but a communication predicate is required to ensure the termination of consensus. We have proposed an appropriate implementation that satisfies the required communication predicate in good periods. We have validated our implementation by running simulations in multi-hop wireless networks. The results of simulations validate the existence of the good periods and confirm that our approach is applicable for realistic wireless networks.

To the best of our knowledge, we are the first to provide the rigorous performance results for consensus in wireless networks. We could not compare our results with Chockler's paper [8] since they do not provide the time unit in their figures. The results in Vollset's paper [14] are far from being efficient (they require around 100 seconds in average for one instance of consensus). Finally, the performance evaluation in Wu's paper [15] is of limited utility since they do not use a realistic MAC layer in their simulations. Although the results of this paper are limited to the simulations, we believe that this approach is applicable in real systems. Our future work is to explore deployment of the system using a network of actual nodes.

References

[1] JiST/SWANS. http://jist.ece.cornell.edu.

[2] N. Badache, M. Hurfin, and R. Macedo. Solving the Consensus Problem in a Mobile Environment. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference, IPCCC'99*, Phoenix, Arizona, USA, February 1999.

[3] R. Barr. *An efficient, unifying approach to simulation using virtual machines*. PhD thesis, Cornell University, Ithaca, NY, May 2004.

[4] J.-Y. L. Boudec. Methods, practice and theory for the performance evaluation of computer and communication systems, 2006. Lecture Notes, EPFL, Switzerland, http://ica1www.epfl.ch/perfeval/lectureNotes.htm.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, Mar. 1996.

[6] B. Charron-Bost and A. Schiper. Harmful dogmas in fault tolerant distributed computing. *ACM SIGACT news*, 38(1):53–61, March 2007.

[7] B. Charron-Bost and A. Schiper. The Heard-Of model: Computing in Distributed Systems with Benign Failures. Technical Report TR, EPFL, June 2007.

[8] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *PODC '05*, pages 197–206, New York, NY, USA, 2005. ACM.

[9] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, Apr. 1988.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, Apr. 1985.

[11] I. W. Group. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Strandard 802.11-1997, New York, NY, 1997.

[12] M. Hutle and A. Schiper. Communication Predicates: A High-Level Abstraction for Coping with Transient and Dynamic Faults. In *IEEE Int Conf on Dependable Systems and Networks (DSN)*, pages 92–101, June 2007.

[13] L. Lamport. The part-time parliament. *ACMTCS*, 16(2):133–169, May 1998.

[14] E. W. Vollset and P. D. Ezhilchelvan. Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in manets. In *SRDS '05*, pages 166–178, Washington, DC, USA, 2005. IEEE Computer Society.

[15] W. Wu, J. Cao, J. Yang, and M. Raynal. Design and Performance Evaluation of Efficient Consensus Protocols for Mobile Ad Hoc Networks. *IEEE Transactions on Computers*, 56(8):1055–1070, August 2007.

APPENDIX

Theorem 1. *Algorithm 2 implements the* LastVoting *predicate in a good period of minimal length* 13δ. [10]

Proof. The proof is based on the following Lemmas. □

Lemma 1. *Let phase ϕ_0 be the largest phase when good period starts at time τ_G. Then, there is some process that starts phase $\phi_0 + 1$ at latest by time $\tau_G + 5\delta$.*

Proof. According to the code of Algorithm 2, all contenders start a timer per phase (line 9). According to the definition of contender set, there is at least one correct process in contender set. This process times out for phase ϕ_0 at latest by time $\tau_G + 5\delta$ (line 36), and starts phase $\phi_0 + 1$ (line 39) at latest by time $\tau_G + 5\delta$. □

Lemma 2. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. Then, process p belongs to the $Contender$ set.*

Proof. From Lemma 1 process p exists. According to the Algorithm 2, a process starts phase ϕ_0 for following reasons, either: (i) it receives a message from another process for phase ϕ_0 (line 31), or (ii) it ends phase $\phi_0 - 1$ by deciding (line 45), or (iii) its timer for phase $\phi_0 - 1$ expires (line 39), or (iv) after 2δ, the coordinator does not receive from a majority set (line 42). The first case is not possible, since p is the first process that starts phase ϕ_0. In the second case, we have $p = coord_p$ which implies $p \in Contender$ by definition. For the two last cases, since p has a timer (line 9) it is already a contender. □

Lemma 3. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. All processes start phase ϕ_0 at latest by time $\tau + \delta$.*

Proof. From Lemma 2 we have $p \in Contender$. According to the Algorithm 2, process p starts phase ϕ_0 by sending a message to all (line 11). Since we are in good period, this message will be received by all processes at latest by $\tau + \delta$. All processes that receive this message start phase ϕ_0. If some process at phase $\phi_0 - 1$ times out, just before receiving this message, it starts phase ϕ_0 on its own before $\tau + \delta$. Thus, all processes start phase ϕ_0 at latest by time $\tau + \delta$. □

Lemma 4. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. All processes have the same coordinator by time $\tau + 2\delta$.*

Proof. According to the Lemma 3, all processes start phase ϕ_0 at latest by time $\tau + \delta$. Assume there is some other process $q \in Contender$ such that $priority_q > priority_p$. Process q starts phase ϕ_0 at time t, $\tau < t < \tau + \delta$, considers itself as coordinator (line 26), and sends its first message for phase ϕ_0 to all (line 11). This message will also be received by all processes at latest by time $t + \delta < \tau + 2\delta$. All processes change their coordinator to q (line 23) before $\tau + 2\delta$. □

Lemma 5. *Let p be the unique coordinator with highest priority that starts phase ϕ_0 at time $\tau > \tau_G$. Algorithm 2 provides the* LastVoting *predicate by time $\tau + 5\delta$.*

Proof. Process p starts phase ϕ_0 by sending its message to all (line 11). All processes receive this message by time $\tau + \delta$ (Lemma 3) and start round $4\phi_0 - 3$ (line 12). Since p is the unique coordinator of phase ϕ_0, no other process executes line 11. Since p is the process with highest priority, all processes accept p as coordinator in phase ϕ_0 (line 29). Since we are in good period, a round do not take more than δ. Algorithm 1 requires four rounds (4δ). In total at latest by time $\tau + 5\delta$ the *LastVoting* predicate is satisfied. □

Lemma 6. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. Let $c \neq p$ with highest priority be the coordinator of phase ϕ_0 that doesn't receive from a majority of processes. Process c starts phase $\phi_0 + 1$ at latest by time $\tau + 3\delta$.*

Proof. From Lemma 3, process c starts phase ϕ_0 at latest by time $\tau + \delta$. From Lemma 4, process c becomes the unique coordinator of phase ϕ_0 at latest by time $\tau + 2\delta$. From the code of Algorithm 2, process c, 2δ after starting phase ϕ_0, finds out that it has not received from a majority of processes (line 40). So, it starts phase $\phi_0 + 1$ at latest by time $\tau + 3\delta$ (line 42). □

Lemma 7. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. The* LastVoting *predicate is satisfied by time $\tau + 8\delta$.*

Proof. Two cases are possible: either p is the process with highest priority or not. In the first case, from Lemma 5, the predicate is satisfied by time $\tau + 5\delta$. In the second case, from Lemma 4, there is a unique coordinator, c, by time $\tau + 2\delta$. Process c starts phase $\phi_0 + 1$ at latest by time $\tau + 3\delta$ according to Lemma 6. In phase $\phi_0 + 1$, process c is the unique coordinator and again according to the Lemma 5 the predicate is satisfied by time $\tau + 8\delta$. □

Analysis: From Lemma 1, we have seen that at most 5δ after τ_G a new phase is started properly. From Lemma 7, we require 8δ to satisfy the *LastVoting* predicate. In total, we need a good period of minimal length 13δ to provide *LastVoting* predicate.

[10] δ is end-to-end multi-hop transmission delay.