Consensus in Wireless Ad hoc Networks *

Fatemeh Borran
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
1015 Lausanne, Switzerland

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

André Schiper
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
1015 Lausanne, Switzerland

Abstract

Solving consensus in wireless ad hoc networks has started to be addressed in several papers. Most of these papers adopt system models similar to those 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 communication layer. The description of the communication layer is augmented with simulation results that validate the feasibility of our approach and provide better understanding of the behavior of realistic wireless environments.

*Research funded by the Swiss National Science Foundation under grant number 200021-111701. This is an updated version of the technical report *Consensus Problem in Wireless Ad hoc Networks: Addressing the Right Issues*

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. Both process and link failures are possible in wireless networks. A failure can be permanent or transient. Packet loss is more frequent than traditional networks because of collisions and channel interference. Since every sent message can collide with other messages, the message complexity of an algorithm is not only a performance issue in wireless networks. Because of its importance in fault-tolerant distributed systems, consensus problem becomes a new challenge in wireless ad hoc networks.

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. These papers essentially adopt system models similar to those 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 Heard-Of (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 unavailable for a while due to a temporary obstacle to signal propagation.

Having said this, 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 communication layer. As suggested above, we believe that the right model for consensus in ad hoc networks is a model that handles process and link faults with the same mechanism, *e.g.*, the HO model. Several consensus algorithms have been expressed in this model, see [7]. Out of these algorithms, only two of them genuinely tolerate message loss: the *One-Third-Rule* (*OTR*) algorithm, and the *Paxos/LastVoting* algorithm (*LastVoting* is basically *Paxos* [13] expressed in the HO model). *OTR* is certainly not adequate, because it is too costly in ad-hoc networks (n-n communication pattern, i.e., in every step all processes send messages to all) *Paxos/LastVoting* is based on the much more economical 1-n communication pattern (communication only between the coordinator and the other processes). The paper shows that an adequate communication layer can nicely handle this 1-n communication pattern in multi-hop networks without any additional overhead for the routing of messages or for election of the coordinator process. The description of the communication layer 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 communication layer. 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 from 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 [15] 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 is not a good choice for multi-hop ad hoc networks. We believe that our 1-n broadcast-convergecast algorithm is much more efficient than the general broadcast protocols proposed here.

Finally, Wu *et al.* [16], 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 a simple approach to solve consensus in ad hoc networks is to extend *Paxos/LastVoting*, an algorithm that tolerates message loss, with an adequate communication layer.

3. Consensus algorithm and HO model

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: Two first properties are *safety* properties and the last one is a *liveness* property.

- *Validity:* Any decision is the initial value of some process.
- *Uniform Agreement:* No two processes decide differently.
- *Termination:* All processes eventually decide.[1]

[1] 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.

3.1. The HO model

For solving consensus, we consider the HO model defined in [7]. The model is very simple, and can be explained in one paragraph. 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, as shown in Section 3.2.

If for some round r, we have $q \notin HO(p, r)$, this means that p did not receive q's message of round r. This can be due to the crash of the sender q, or to a failure of the link between p and q. The model does not need to distinguish between the two cases. This makes the model well suited to handle transient (process and link) faults [12]. Note that transient faults are more general than crash-stop faults.

3.2. The *Paxos/LastVoting* algorithm

The *Paxos/LastVoting* algorithm [7] is the most appropriate algorithm for ad hoc networks (*LastVoting* is basically *Paxos* [13] expressed in the HO model, and is also close to the Chandra-Toueg $\Diamond\mathcal{S}$ [5] consensus algorithm): its message complexity is $O(n)$, and it tolerates rounds r in which $HO(p, r)$ is empty for all p, (i.e., it tolerates message loss). The code is given in Algorithm 1. From here on we call the algorithm simply *LastVoting*.

LastVoting 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 communication layer, see Section 4. The communication layer also provides the messages received from the set $HO(p, r)$.

Before presenting our communication layer, we describe briefly how the *LastVoting* algorithm works: Each process p has a timestamp ts_p which attaches to its proposal x_p. (1) In the first round of every phase, each process sends its proposal and timestamp to its coordinator (line 10). If the coordinator receives proposals from a majority of processes, it sets its $vote$ to the last proposal with the highest timestamp (line 14). (2) In the second round, the coordinator sends its $vote$ to all (line 19). Every process that receives coordinator's vote (line 21), changes its proposal and updates its timestamp. (3) These processes send an ack message to the coordinator in the third round (line 27). If the coordinator receives a majority of acks (line 29), it can decide on its $vote$: (4) the coordinator sends its $vote$ (the decision) to all processes in the last round (line 34), and each process that receives the coordinator's $vote$ decides (line 37).

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

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

7: **Round** $r = 4\phi - 3$**:**
8: S_p^r:
9: **if** $Coord(p, \phi) \neq \bot$ **then**
10: send $\langle x_p, ts_p \rangle$ to $Coord(p, \phi)$
11: T_p^r:
12: **if** $p = Coord(p, \phi)$ **and** number of $\langle \nu, \theta \rangle$ received $> n/2$ **then**
13: let $\overline{\theta}$ be the largest θ from $\langle -, \theta \rangle$ received
14: $vote_p$:= one $\overline{x}$ such that $\langle \overline{x}, \overline{\theta} \rangle$ is received
15: $commit_p :=$ `true`

16: **Round** $r = 4\phi - 2$**:**
17: S_p^r:
18: **if** $p = Coord(p, \phi)$ **and** $commit_p$ **then**
19: send $\langle vote_p \rangle$ to all processes
20: T_p^r:
21: **if** received $\langle v \rangle$ from $Coord(p, \phi)$ **then**
22: $x_p := v$
23: $ts_p := \phi$

24: **Round** $r = 4\phi - 1$**:**
25: S_p^r:
26: **if** $ts_p = \phi$ **then**
27: send $\langle ack \rangle$ to $Coord(p, \phi)$
28: T_p^r:
29: **if** $p = Coord(p, \phi)$ **and** number of $\langle ack \rangle$ received $> n/2$ **then**
30: $ready_p :=$ `true`

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

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 the existence of a phase ϕ_0 in which following predicate holds:

- A majority of processes consider the same coordinator c_0 in ϕ_0: $\exists M \subseteq \Pi$, $|M| > n/2$, $\forall p \in M$: $Coord(p, \phi_0) = c_0$, and
- A majority of processes p receive c_0's message in rounds $4\phi_0 - 2$ and $4\phi_0$: $\exists M \subseteq \Pi$, $|M| > n/2$, $\forall p \in M$: $c_0 \in HO(p, 4\phi_0 - 2)$ and $\exists M \subseteq \Pi$, $|M| > n/2$, $\forall p \in M$: $c_0 \in HO(p, 4\phi_0)$, and
- The coordinator receives the message from a majority of processes in rounds $4\phi_0 - 3$ and $4\phi_0 - 1$:

$|HO(c_0, 4\phi_0 - 3)| > n/2$ and $|HO(c_0, 4\phi_0 - 1)| > n/2$.

4. Communication layer for *LastVoting*

We describe now the communication layer for *LastVoting*. Its role is to ensure the above predicates, which includes the election of a coordinator. We start with the system model, then we explain the architecture of our communication layer, and finally we describe our communication and coordinator election algorithm.

4.1. System model

Wireless network: We consider an asynchronous multi-hop wireless network consisting of set of n nodes.[2] 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*: our algorithm doesn't require any protocol like TCP, unlike [16].

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

Good period: *LastVoting* is always safe. To ensure liveness we assume that, from time to time, unknown to the processes, 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.[3] Note that this is not in contradiction with our previous assumption about unreliable links and unpredictable delays. This is required to overcome FLP impossibility result. Notice that the good period is a realistic system assumption inspired from [9] and already used in [12].

4.2. Architecture

Figure 1 shows the overall view of our architecture. The uppermost layer corresponds to *LastVoting* (Algorithm 1). Algorithm 1 contains two functions S_p^r and T_p^r that are called by the layer beneath, namely Algorithm 2:[4]

- 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 15).
- 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 34).

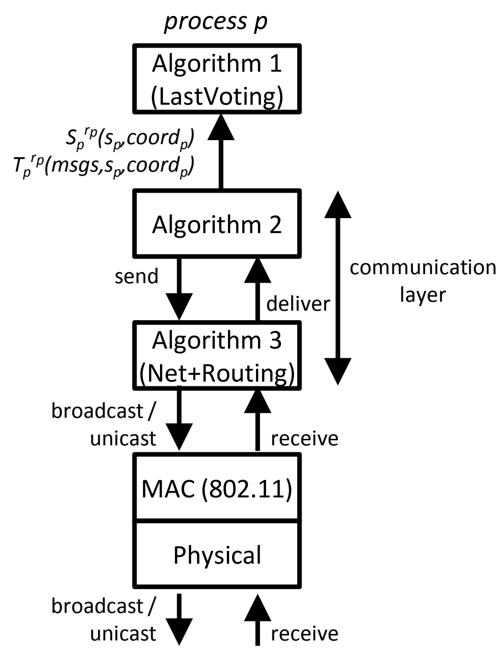


Figure 1. Architecture.

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. For sending a message, Algorithm 2 calls the *send* function of Algorithm 3. Upon reception of a message by Algorithm 3, the *deliver* function of Algorithm 2 is called. 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.

[3] It would be easy to adapt the algorithm to an unknown δ value [9], *e.g.*, using adaptive timeout.

[4] Actually Algorithm 1 does not send the messages in lines 10, 19, 27, 34; it simply defines which messages should be sent to which destinations.

4.3. Algorithm 2: the upper communication layer

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 phase number ϕ_p (line 8) and the round number r_p (line 14). Second, Algorithm 2 relies on Algorithm 3 for sending (and receiving) messages (*e.g.*, line 17): 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 number ϕ_p and round number r_p to the messages it sends (*e.g.*, line 17). Whenever a process receives a message from some phase $\phi > \phi_p$, it jumps to the first round of that phase (line 33, 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 predefined set $Contender \subset \Pi$.[5] 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 messages of Algorithm 1. Each process $p \in \Pi$ that receives a message from phase $\phi \geq \phi_p$ from some process q with higher priority (line 23, 29), updates its coordinator to q and priority to q's priority.

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 $\geq \phi_0$.[6] 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 42); c is the unique coordinator for the remainder of the good period.

[5] The $Contender$ set must be large enough to ensure that all its members are not crashed at the same time.
[6] All proofs are in the Appendix.

Algorithm 2 Upper communication layer: Coordinator election and message reception (code of process p).

1: **Initialization:**
2: $msgs_p \leftarrow \emptyset$ /* set of messages received */
3: $s_p \leftarrow init_p$ /* state of process p */
4: $coord_p \leftarrow p$ for $p \in Contender$; otherwise $\bot$
5: $priority_p \leftarrow p$'s identity for $p \in Contender$; otherwise 0
6: startPhase (1)

7: **function** startPhase (ϕ)
8: $\phi_p \leftarrow \phi$ /* phase number */
9: **if** $p \in Contender$ **then** $timer_p \leftarrow 0$
10: **if** $p = coord_p$ **then**
11: send ($\langle \phi_p, -, p, priority_p, - \rangle, \Pi$) /* calls function send of Algorithm 3; message used to elect coordinator; Π is the destination set */
12: startRound ($4\phi_p - 3$)

13: **function** startRound (r)
14: $r_p \leftarrow r$ /* round number */
15: $\langle msg, dst \rangle \leftarrow S_p^{r_p}(s_p, coord_p)$ /* calls function S of Algorithm 1 */
16: **if** $msg \neq null$ **then**
17: send ($\langle \phi_p, r_p, p, priority_p, msg \rangle, dst$) /* calls function send of Algorithm 3 */

18: **function** deliver ($\langle \phi, r, q, priority_q, m \rangle$) /* delivered from Algorithm 3 */
19: **if** $\phi < \phi_p$ **or** $r < r_p$ **then**
20: ignore message
21: **else**
22: $msgs_p \leftarrow msgs_p \cup \{\langle \phi, r, q, priority_q, m \rangle\}$
23: **if** $\phi = \phi_p$ **and** $priority_q > priority_p$ **then**
24: $coord_p \leftarrow q$
25: $priority_p \leftarrow priority_q$
26: **if** $\phi > \phi_p$ **then**
27: $coord_p \leftarrow p$ for $p \in Contender$; $\bot$ otherwise
28: $priority_p \leftarrow p$'s identity for $p \in Contender$; 0 otherwise
29: **if** $priority_q > priority_p$ **then**
30: $coord_p \leftarrow q$
31: $priority_p \leftarrow priority_q$
32: **forall** $r' \in [r_p, r)$ **do** $s_p \leftarrow T_p^{r'}(\{\langle m, q \rangle | \langle \phi_p, r', q, -, m \rangle \in msgs_p\}, s_p, coord_p)$ /* calls function T for all intermediate rounds */
33: startPhase (ϕ)
34: $ns_p \leftarrow T_p^{r}(\{\langle m, q \rangle | \langle \phi_p, r, q, -, m \rangle \in msgs_p\}, s_p, coord_p)$ /* calls function T of Algorithm 1 */
35: **if** $ns_p \neq s_p$ **then** /* new state of p is different from its current state */
36: $s_p \leftarrow ns_p$
37: startRound ($r + 1$)

38: **upon** $timer_p > 5\delta$ **do** /* timer expires */
39: $coord_p \leftarrow p$
40: $priority_p \leftarrow p$'s identity
41: startPhase ($\phi_p + 1$)

42: **upon** $timer_p > 2\delta$ **do** /* start new phase if no progress as coordinator */
43: **if** $p = coord_p$ **and** $r_p < 4\phi_p - 2$ **then**
44: startPhase ($\phi_p + 1$)

45: **upon** decide for phase ϕ_p **do**
46: **if** $p = coord_p$ **then**
47: startPhase ($\phi_p + 1$)

Message reception: 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 $p \notin Contender$. If $p \notin Contender$, then p does not use 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 26) or (2) p has received "enough" messages in round r (lines 34 to 36). 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 34): if not enough messages have been received, the T_p^r function does not modify the state (line 35) 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 coordinator election round and round $4\phi - 3$) and p is still in round $4\phi_p - 3$ (line 42), or (2) 5δ time units have elapsed (duration of coordinator election round and rounds $4\phi - 3$ to 4ϕ) and p is still in phase ϕ_p (line 38).

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 45) 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 38). Finally, another optimization – not shown in Algorithm 2 but is considered in our simulations – is the following: the coordinator, on starting a new phase (lines 44 and 47), does not need to send an additional message to all (line 11), because there is a unique coordinator. This additional message has to be sent when there is no unique coordinator: either timer has expired (line 41) or a message from higher phase is received (line 33).

4.4. The lower communication layer: broadcast and convergecast

Algorithm 2 invokes Algorithm 3 (lower communication layer) when it sends a message in lines 11 and 17. 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 messages 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

Algorithm 3 Lower communication layer: broadcast and convergecast algorithm (code of process p).

1: **Initialization:**
2: $parent_p \in \Pi \cup \{\text{NULL}\}$, initially NULL
3: $level_p \in \mathbb{N}$, 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** $dst = \Pi$ **then**
7: $parent_p := p$
8: $level_p := 1$
9: locally broadcast $\langle MESSAGE, p, level_p, m\rangle$
10: **else**
11: unicast $\langle RESPONSE, q, level_p, m\rangle$ to $parent_p$

12: **upon** receive $\langle MESSAGE, root, l, m\rangle$ from node q with $priority_q$ for the first time **do**
13: deliver (m) /* calls Algorithm 2 */
14: **if** $priority_q > priority_p$ **then**
15: $parent_p := q$
16: $level_p := l + 1$
17: **if** $priority_q \geq priority_p$ **then**
18: locally broadcast $\langle MESSAGE, root, level_p, m\rangle$

19: **upon** receive $\langle RESPONSE, root, l, m\rangle$ for the first time **do**
20: **if** $p = root$ **then**
21: deliver (m) /* calls Algorithm 2 */
22: **else**
23: unicast $\langle RESPONSE, root, level_p, m\rangle$ to $parent_p$

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. Therefore, each 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.

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.

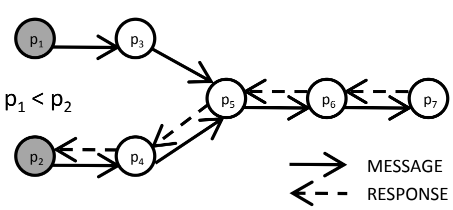


Figure 2. Broadcast *vs.* convergecast.

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

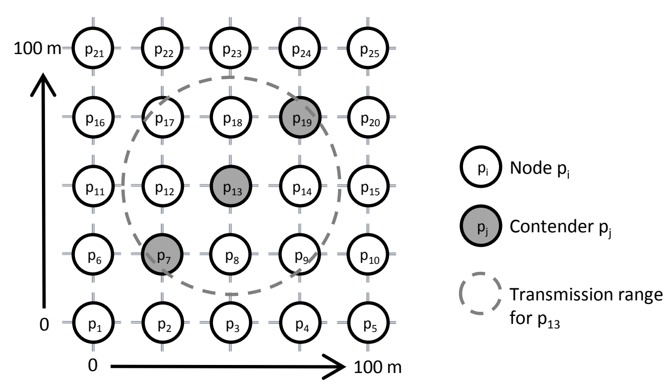


Figure 3. Square grid of size 5×5 in network area $100 \times 100\, m^2$.

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 5.2.5). We measure the impact of location and number 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.

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 3.2 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 and 3 in wireless ad hoc networks. Note that the good and bad periods are not simulated in our scenarios: frequent collisions and node interference during some time interval corresponds to a bad period; otherwise we have a good period.

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.

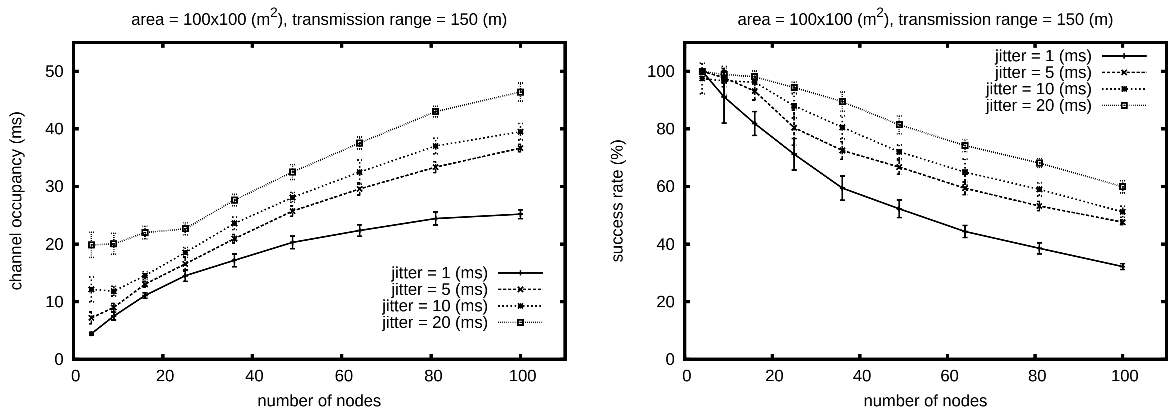
We have defined two (independent) metrics: *consensus latency* and *consensus throughput*. Consensus latency is expressed in terms of average number of phases per consensus from initialization to first decision. Consensus throughput represents how many instances of consensus can be run successfully in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

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. In these scenarios no process crashes and no packet is explicitly dropped: the only source of failure is the collisions and node interferences.[7] However, to observe the performance

[7]Considering only message loss does not make consensus easier to solve: consensus is impossible to solve in a synchronous system with lossy links [14]. To solve consensus, message loss must be restricted.

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. We have noticed that increasing background traffic only reduces the throughput of our algorithm slightly (the corresponding graphs are in the Appendix). 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.

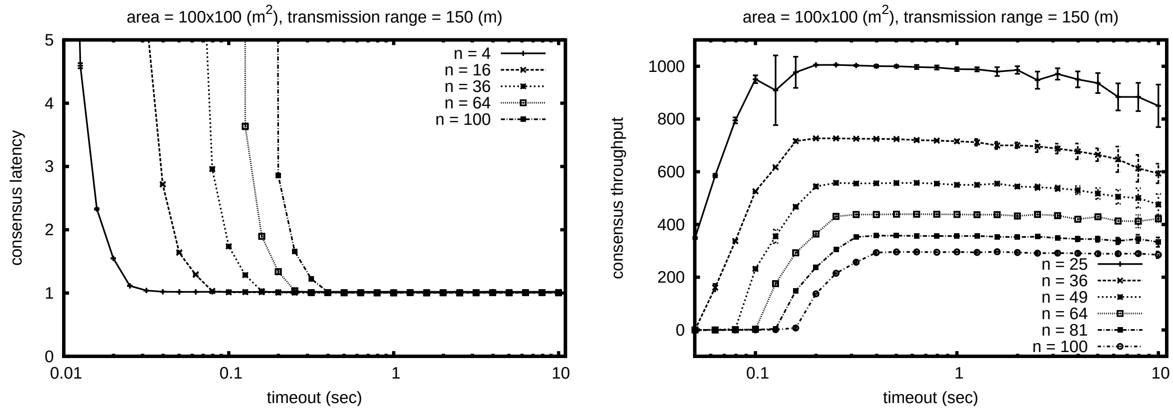
(b) Success rate *vs.* density.

Figure 4. Impact of network 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.

5.2.1. Single-hop scenarios

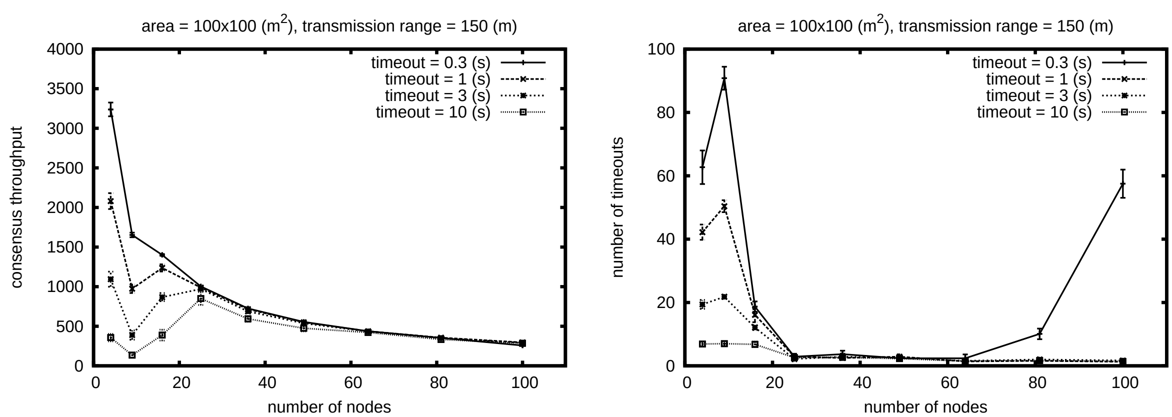
First, we consider a single-hop network in which all nodes are in communication range of each other. The network area is $100 \times 100\ m^2$. We gradually increased the network density. Only a single node, for example p_1, belongs to the $Contender$ set. We measured the average number of phases per consensus in networks with



(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.

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. However, 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

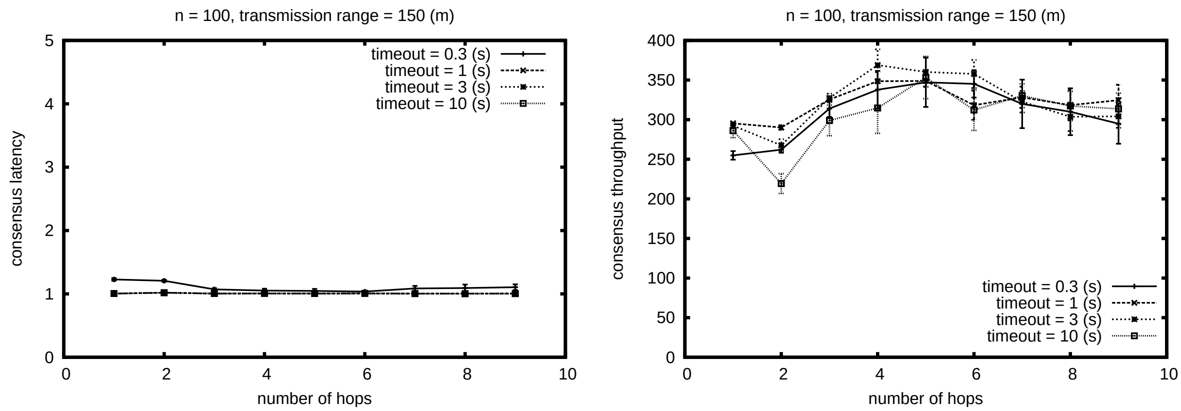
Figure 6. Impact of network density in consensus throughput in single-hop wireless networks.

Figure 6(a) is just another representation of Figure 5(b) to better visualize the impact of network density. In general, by increasing density (number of nodes), the throughput of our algorithm decreases, independent of phase timeout value. This is because of 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 the performance in Figure 6(a).

5.2.2. Multi-hop scenarios

We now consider multi-hop scenarios where not all the nodes are 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 coordinator ($p1$ is located at the lower left corner of the grid).



(a) Consensus latency *vs.* network diameter

(b) Consensus throughput *vs.* network diameter

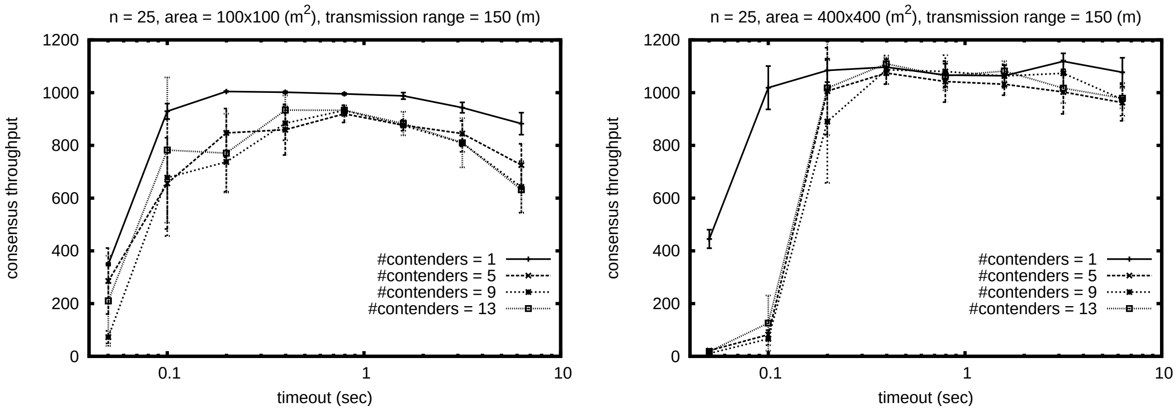
Figure 7. Impact of network diameter in consensus latency and throughput in multi-hop wireless networks.

Figure 7(a) shows the scalability of our algorithm in multi-hop networks. By increasing the network area for 100 nodes, on 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. Additional graphs are in the Appendix.

5.2.3. Impact of location and number of contenders

To see the impact of the contender's position on consensus throughput, we varied the position of the contender (coordinator) from bottom-left corner to the center.[8] 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 throughput). The test accepts the null hypoth-

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



(a) Single-hop network (b) Multi-hop network

Figure 8. Impact of contenders in consensus throughput.

esis 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.

In Figure 8, we increased the number of contenders in a network of 25 nodes from 1% to 50%. The figure confirms that for large enough timeout, the number of contenders does not have an important impact on the consensus throughput. In fact, once the process with highest priority is elected as the coordinator, it remains the same as long as a majority of its messages are not lost.

To better understand how the crash of the coordinator influences our algorithm, we ran a simulation in which the contender with the highest priority crashes and recovers with frequency $1/t$. We noticed that for $t \gg timeout$ (which is reasonable assumption) the consensus throughput is almost the same as the case in which the contender never crashes (the corresponding graphs are in the Appendix).

5.2.4. Impact of message loss

We now consider scenarios in which a node on receiving a message can discard it with probability p (uniform distribution). This simulates the loss of the message during its passage through the network. There is one coordinator located at the lower left corner of the grid. Figure 9 shows the sensitivity of our algorithm to p and confirms the ability of our algorithm to tolerate a minority message

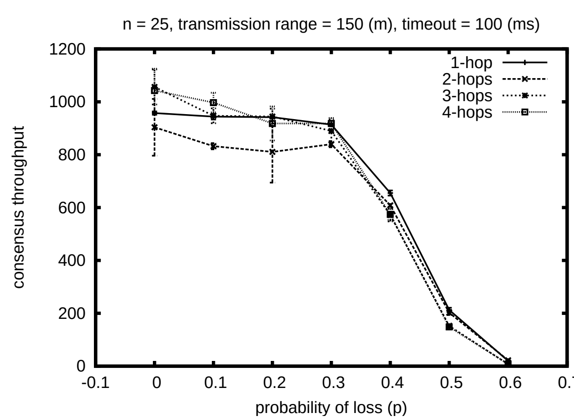


Figure 9. Impact of message loss in consensus throughput.

loss. Additional graphs are in the Appendix.

5.2.5. Impact of mobility

Finally, we measure the impact of 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 destination in the field and move directly towards it at constant speed. When they reach the destination, they pick a new destination and so on. Figure 10 shows the behavior of our algorithm with node speed. The coordinator is located at the lower left corner of the grid. Note that when the network diameter is 2 (network area is 200×200), a majority of nodes (13 nodes) are in communication range with the coordinator, which explains why there is no difference between 1-hop and 2-hop scenarios.

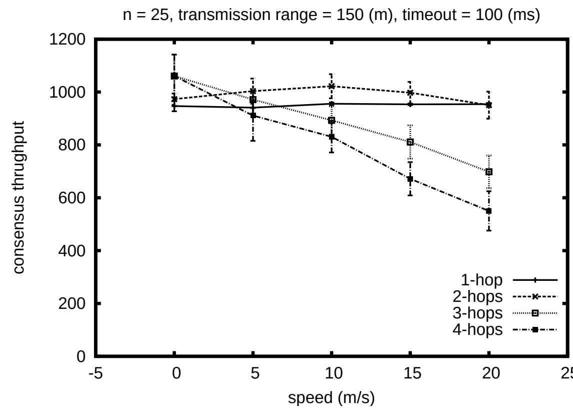


Figure 10. Impact of mobility in consensus throughput.

6. Conclusion

The *Paxos/LastVoting* algorithm extended with an adequate communication layer 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.

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 [15] 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 [16] is of limited utility since they do not use a realistic MAC layer in their simulations. Although the results of our 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.

Acknowledgments We would like to thank Nuno Santos for his comments to improve the paper.

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. *J. 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. *J. 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. *J. 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 *DSN07*, pages 92–10. IEEE, June 2007.

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

[14] N. Santoro and P. Widmayer. Time is not a healer. In *Proceedings of the 6th Symposium on Theor. Aspects of Computer Science*, pages 304–313, Paderborn, Germany, 1989.

[15] 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.

[16] 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

A. Proofs

Theorem 1. *Algorithm 2 implements the liveness predicate of* LastVoting *in a good period of minimal length* 13δ. [9]

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 process (that is up) in contender set. This process times out for phase ϕ_0 at latest by time $\tau_G + 5\delta$ (line 38), and starts phase $\phi_0 + 1$ (line 41) 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 33), or (ii) it ends phase $\phi_0 - 1$ by deciding (line 47), or (iii) its timer for phase $\phi_0 - 1$ expires (line 41), or (iv) after 2δ, the coordinator does not receive from a majority set (line 44). 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 $\tau + \delta$. □

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

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 (line 33), $\tau < t < \tau + \delta$, considering itself as coordinator (line 27), 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 24) 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 liveness predicate of* LastVoting *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 30). Since we are in good period, a round does not take more than δ. Algorithm 1 requires four rounds (4δ). In total at latest by time $\tau + 5\delta$ the liveness predicate of *LastVoting* 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 42). So, it starts phase $\phi_0 + 1$ at latest by time $\tau + 3\delta$ (line 44). □

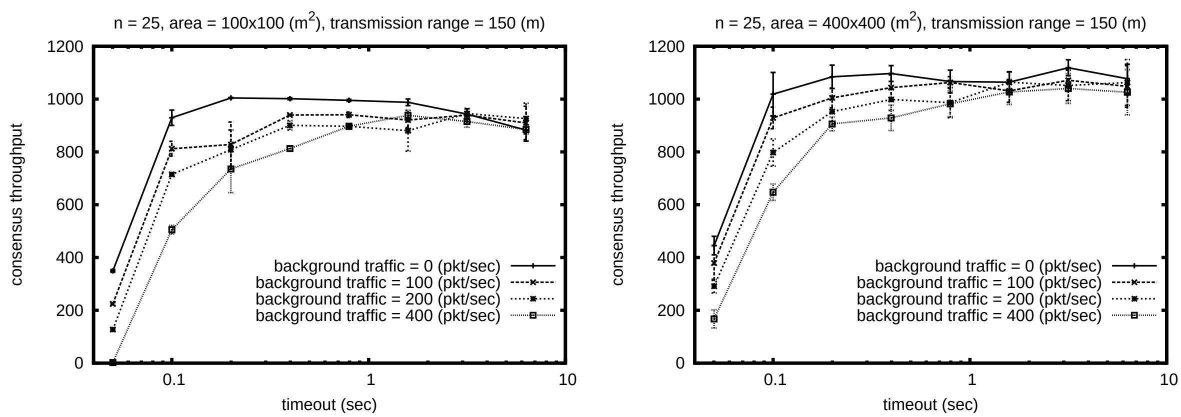
Lemma 7. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. The liveness predicate of* LastVoting *is satisfied by time $\tau + 8\delta$.*

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 liveness predicate of *LastVoting*İn total, we need a good period of minimal length 13δ to provide the predicate.

B. Additional results

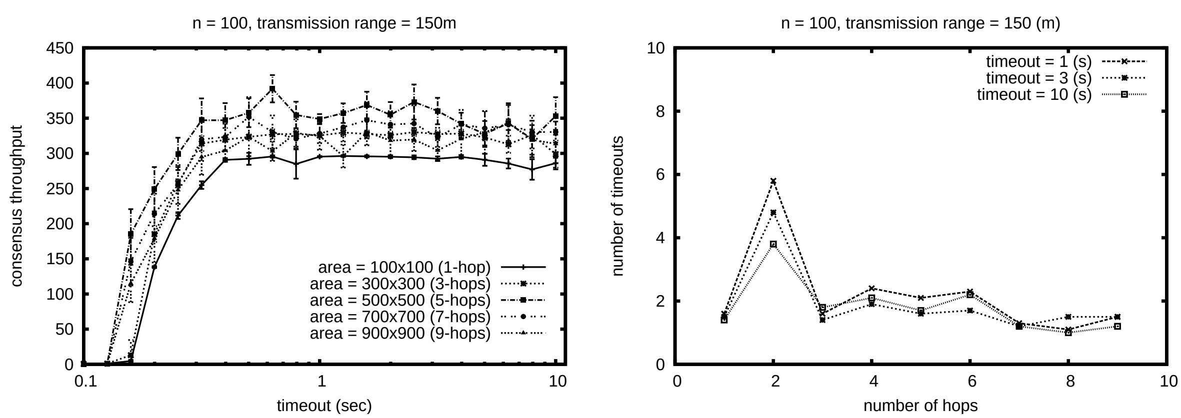
B.1. Impact of background traffic



(a) Single-hop network (b) Multi-hop network

Figure 11. Impact of background traffic in consensus throughput.

B.2. Multi-hop scenarios

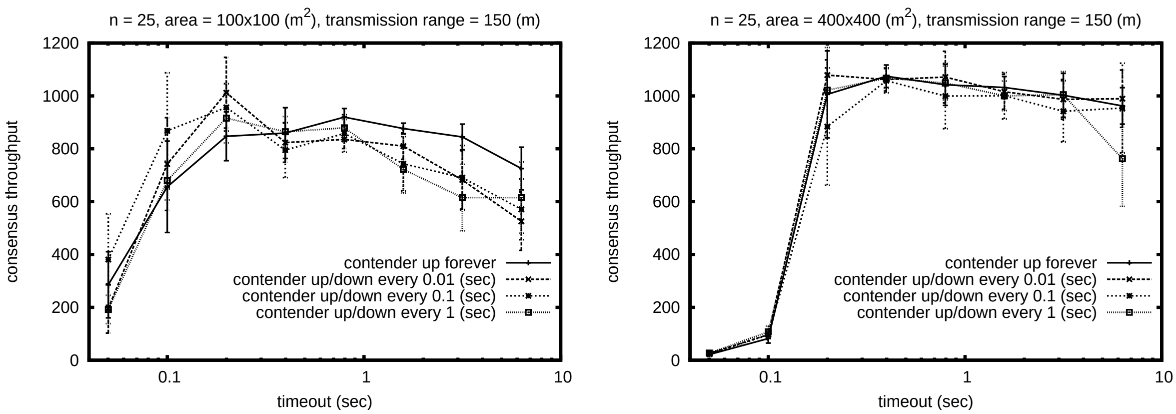


(a) Consensus throughput *vs.* timeout (b) Number of timeouts *vs.* network diameter

Figure 12. Additional multi-hop scenarios.

B.3. Impact of coordinator crash

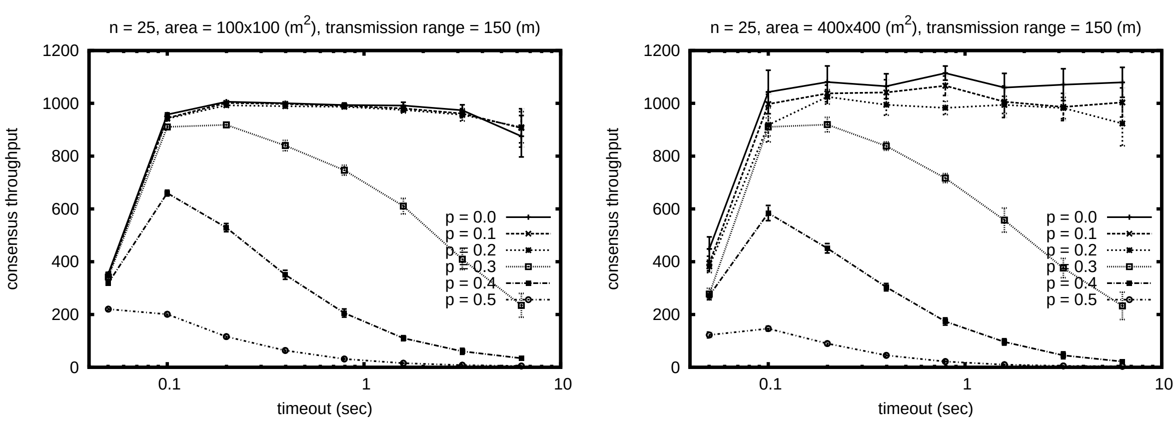
B.4. Impact of message loss



(a) Single-hop network

(b) Multi-hop network

Figure 13. Impact of coordinator crash in consensus throughput.



(a) Single-hop network

(b) Multi-hop network

Figure 14. Impact of message loss in consensus throughput.