

Proseminar Verteilte Algorithmen

Elections in a Distributed Computing System

Autor: Johannes Gilger

Betreuender Lehrstuhl:
Lehrstuhl für Informatik II
RWTH Aachen

Betreuer: Martin Neuhäuser

Aachen, January 20, 2007

Contents

1	Introduction	2
2	Leader election algorithms	2
2.1	Assumptions about our communication system.	2
2.2	The Bully Election Algorithm	4
2.3	The Invitation-Algorithm	7
2.4	Practical considerations.	11
3	Leader election in consumer products	12
3.1	Leader election in the IEEE 1394 Serial Bus (Firewire)	12
4	Conclusion	12
5	Acknowledgment	13

1 Introduction

A short introduction to leader election

Having more than one computing entity work on a shared problem has been common since the very early days of computing. The main reasons to use more than one computer are obvious: Technical limitations of building everything into one machine, reliability increase by using more than one single point, which may fail more easily, and of course extendability. If you have a network, a problem and an algorithm for a problem it is very likely that you can vary the number of nodes which work on it, may it be because you add machines or because machines fail. In a perfect world everything works as it should. One node would be called leader, its id would manually be programmed into the nodes and the system would be started. In our world we rarely have a constant topology, that is why we need algorithms to elect a leader in case the old topology changes, the leader fails or has been stopped by a higher authority or simply does not fit the task anymore. This is where leader election algorithms step in.

Leader election algorithms have to deal with mutual exclusion. Just like a presidential election, there can only be one elected candidate (per country) after the election. The algorithms have to make sure that the leader of a group is known to all the group members. Also it has to take action in the event of a leader failing, which can happen because it simply stops working or because the communication link with it ceases to exist. Leader election algorithms vary depending on the circumstances under which they are called. This means that their complexity increases as the communication system becomes less dependable. There are scenarios and algorithms which almost match our definition of a perfect world. They do not act on the assumption of communication failures and simply assume that a node which does not respond in a timely manner is down. We can use these protocols where failure of the communication system is unlikely, and if it ever occurred, our leader election would be useless anyway. I will present two leader election algorithms, introduced by Molina in [1] in detail and talk about leader election in general.

2 Leader election algorithms

2.1 Assumptions about our communication system.

Before we can start talking about algorithms we first have to agree on a nomenclature for the nodes, their state-vector and the messages [1]. $S(i)$ identifies node i . This identification number is unique throughout the system. The state-vector is nothing more than a predefined set of variables each nodes stores. Think of integers, Strings and boolean values. Those

values are identified by $S(i) \cdot x$ where x may be one of the following: s, c, d, g, counter, Up, h. So $S(i) \cdot d$ means: “The value of the variable d (task description) of node i”. We assume that the state-vector is stored in safe storage cells (see Assumption 4). For easier handling the id of the nodes, we further assume that our system contains a fixed number of nodes n and that the nodes are numbered from $1 \dots n$.

$S(i) \cdot s$ The status of the node, should be one of the following: “Down”, “Election”, “Reorganization” or “Normal”.

$S(i) \cdot c$ The id of the coordinator according to node i .

$S(i) \cdot d$ Definition of the task being performed.

For the invitation election algorithm, which is discussed later, we need the following additional properties:

$S(i) \cdot g$ The unique group id which node i considers itself part of.

$S(i) \cdot counter$ A counter how many groups have been formed by i .

$S(i) \cdot Up$ The list of nodes which $S(i)$ believes to be in operation.

$S(i) \cdot h$ The id of the node which sent the last “halt” message to node i .

Now we have to make a few assumptions about our communication system (or network) because otherwise we would get lost in a lot of details and unimportant factors which only need to be considered in special scenarios and would go beyond the scope of this document [1].

Assumption 1: All nodes use the same algorithms for communication and election.

Assumption 2: There are no software bugs in the operating systems of the nodes.

Assumption 3: There are no random messages, i.e. messages which appear having been sent by a certain node M although they have not.

Assumption 4: The nodes are equipped with safe storage cells, nonvolatile memory which contains the state vector and is implemented with a double-write method. This memory is not flushed when a node is restarted.

Assumption 5: Failure of a node does not impede the integrity of its operating system and algorithms, only contents of nonsafe memory can be lost.

Assumption 6: There are no transmissions errors. Messages may be lost, but if they do arrive at their destination the receiver can be sure they are intact, i.e. as sent.

Assumption 7: Messages which are not discarded are processed by the receiving node in the exact order they arrived.

Assumption 8: The communication system does *not* fail. If a node does not receive an acknowledgement for a message it sent to another node within a given time it can be sure that the node failed.

Assumption 9: Nodes never pause but respond to messages in a timely fashion. Not responding fast enough is considered a failure, upon which the node must be reset.

Molina describes two additional assertions which are used when analysing the algorithms. Assertion 3 is an extension of the 1st and 2nd which is necessary for verifying networks with separated groups. We can think of Assertion 1 and 2 stating basically the same as Assertion 3, just fixed to a single non-changing group.

Assertion 1: If two or more nodes $S(i)$ and $S(j)$ are in the State “Reorganization” or “Normal” they must have the same coordinator $S(i) \cdot c = S(j) \cdot c$. If they are in the state “Normal” they must also have the same task or task-description ($S(i) \cdot d = S(j) \cdot D$).

Assertion 2: After an election there has to be a coordinator, $S(i) \cdot s = \text{”Normal”}$ and $S(i) \cdot c = i$. For the other nodes in the system for which $S(j) \cdot s = \text{”Normal”}$, $S(j) = i$ must be true.

Assertion 3: Derived from Assertion 1 and Assertion 2 in the context of groups. If two (or more) nodes are normal or reorganizing *and* are the same group then they must also share the same leader. If they both are normal and in the same group they also must share the same task.

Note that I will sacrifice this nomenclature later when I discuss the actual implementation for the sake of understandability.

2.2 The Bully Election Algorithm

The Bully Election Algorithm is in theory a rather simple approach to leader election. Simply put, the node with the highest identification number (id) wins the election by forcing every other node to accept it as its leader. In this kind of election fairness is not important. For this algorithm to work we have to make sure that Assumptions 1-9 hold. I will try to communicate a rough idea of how the algorithm works by giving a short outline:

1. Node i notices that its leader does not respond anymore (it has been checking the leader's state in regular intervals).
2. It tries to contact all the nodes with a higher id number than itself.
3. If a node j with a higher number responds, node i gives up trying to be the leader because it knows node j will start an election process of its own and will eventually be elected.
4. If no node $S(j), j > i$ responds, node i first considers itself coordinator ($S(i) \cdot c = i$) and then sends a "halt" message to the nodes with lower ids than itself, which forces them to halt and restart. Node i then sends a message which proclaims itself as the new coordinator to these nodes. The nodes being check that the "halt" and the "I am the new coordinator" messages originated from the same node and then switch into reorganization mode (this is where the safe memory is important).
5. The process of reorganization means that the coordinator distributes the new task/algorithms for the working nodes.

Figure 1 gives an idea how an election with the Bully-Algorithm might look like. The Bully Algorithm does not work when Assumptions *A8* and *A9* do not hold since it assumes that the network with the nodes is a structure which in itself is stable and the only error which may occur is a failing node. Molina's Bully-Algorithm fails for example when a node pauses for a certain amount of time. For a more realistic and therefore unstable network topology one should use the Invitation-Algorithm for leader election which I will present in the next section.

Listing 1 shows the important part of the Bully Algorithm, the election procedure. I will highlight the important aspects of this procedure, covering them as we move along the source code. This algorithm was modified to some extent from Molina's Algol-like pseudo-code to a more modern appearance (somewhat similar to Java). It employs some methods that differ from the usual understanding of Java. `Node.AreYouThere()` means that message "AreYouThere" is being sent to the node identified by "Node", with possible parameters. How this works in detail is not important. It is important though that messages are received by the node, put in a queue (probably in the safe memory) and then processed in order. This is why a procedure of a node can send a message to itself and exit immediately after that. The "state vector" is split up into variables in the system. These are the properties of the node. Task and status are contained in strings for example, while "id" and "haltFrom" (the variable indicating who sent the last "halt" message) are integers. I also assume that there is some way for a call to notice that the response coming back is not "Ok" (maybe the communication subsystem itself sends a different message if it has not been able to reach the node). After being invoked, the procedure tries to reach the nodes with a higher

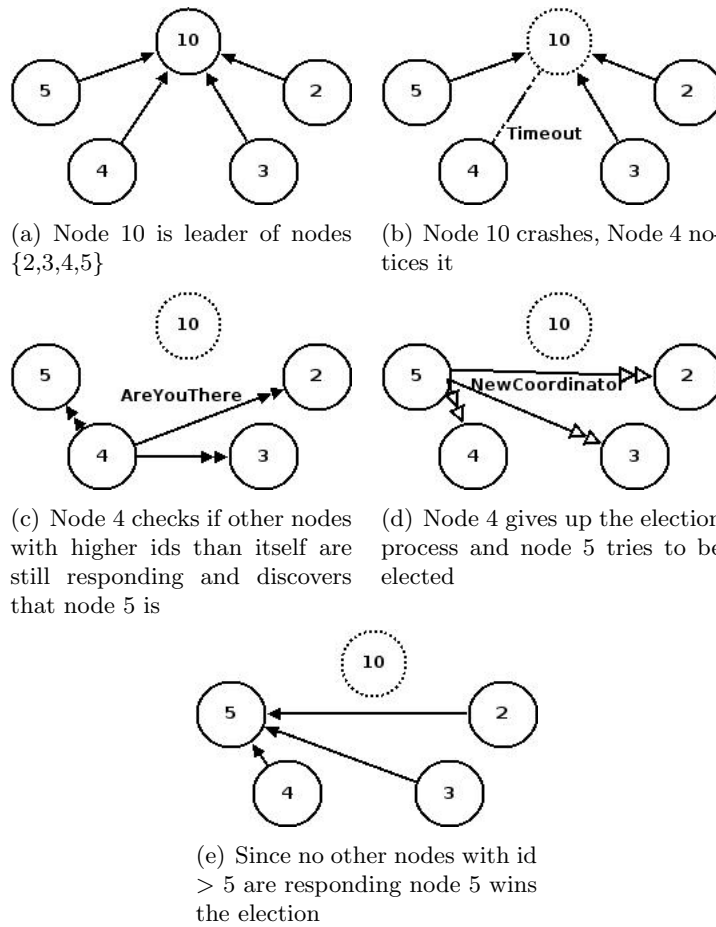


Figure 1: The Bully Election Algorithm (simple example)

id than itself ("id+1, ..., n"). It is necessary that the number of nodes in the system is known ("nodesTotal") for this to work. If one of the called nodes responds, the procedure exits. If no node responded, node "id" stops itself and sets its status to "Election" and the id of the node which halted it (haltFrom) to "id", itself. Then it halts all the nodes with lower ids ("id-1, ..., 1"). Those which responded are flagged "true" in the array "up". After having done that, the node switches to "Reorganization" status. The next thing to do is to announce itself as the coordinator to all the active nodes in "up" by sending them the NewCoordinator message (upon which each node checks if the NewCoordinator message originated from the same node as the previous Halt message). If one of these nodes has failed since the last check, the call NewCoordinator times out and the whole election is restarted. If not, and all nodes have received the notification of the new coordinator, they are all sent the Ready message, which distributes the new task description to the nodes. Nodes receiving the Ready message switch to "Normal" state.

If all the nodes have received the Ready message, the coordinator ("id") goes into "Normal" state itself.

Listing 1: The Bully Election Procedure

```

1 private Election() {
2   int haltFrom, nodesTotal;
3   String status, task;
4   boolean[] up;
5
6   for (int node=id+1; node<nodesTotal; node++) {
7     if (node.AreYouThere(id) == "Yes") { exit 0; }
8   }
9   self.Stop();
10  status = "Election";
11  haltFrom = id;
12  for (int node=id-1; node>0; node--) {
13    if (node.Halt(id) == "Ok") { up[node] = true; }
14  }
15  coordinator = id;
16  status = "Reorganization";
17  for (int node=0; node<= up.length; node++) {
18    if (up[node] == true)
19      if (node.NewCoordinator() != "Ok") {
20        self.Election(); exit 0; }
21    if (node.Ready(task) != "Ok") {
22      self.Election(); exit 0; }
23  }
24  }
25  status = "Normal";
26  }

```

I will try to give a short overview of the rest of the procedures. `AreYouThere` can be considered the equivalent of a "ping" to a node, `AreYouNormal` asks if the node's status is "Normal", `Halt` prepares the node for an election, `Recovery` is the first procedure a node calls when starting, it invokes an election. `Check` is called by the coordinator periodically to see if its child nodes are normal, if not it starts an election. `Timeout` is called after a certain amount of time in which a node has not heard anything from its coordinator. It first checks the coordinator and if it discovers it being down, starts an election.

2.3 The Invitation-Algorithm

The Invitation-Algorithm is a more sophisticated election algorithm. In our case we employ it when assumptions Assumption 8 and Assumption 9 do not hold anymore. As soon as we accept that, we also have to acknowledge that in an environment where Assumption 8 and Assumption 9 do not hold, Assertion 1 and Assertion 2 may not be satisfied. The most simple example

is a partition of the network. In this case we have two different leaders at the same time, albeit in different partitions and Assertion 2 is violated since not all of the nodes have $S(j) = i$, meaning the same leader. We do however (for the simplicity of the algorithm) have to make sure that Assumption 8 and Assumption 9 hold in the subsets of the partitioned network, i.e. that there are no nodes which can only send or receive messages. The principle

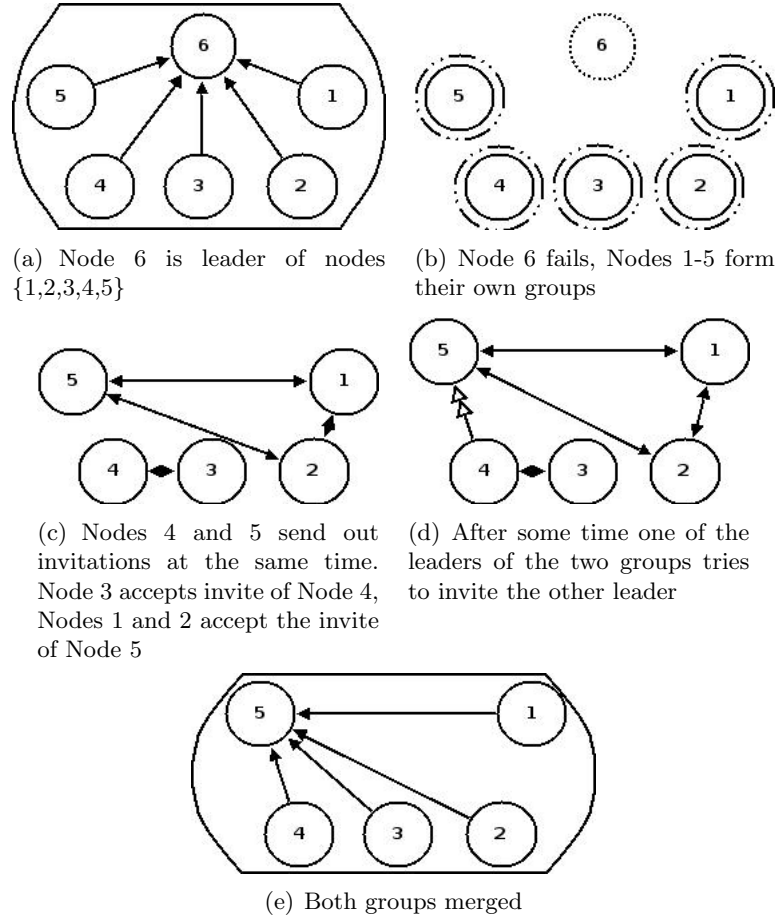


Figure 2: The Invitation-Algorithm (example presented in [1])

the Invitation-Algorithm relies on is the concept of groups. These groups are formed by one or more nodes, each having its own coordinator. When two groups want to connect they call a merge after which there is only one coordinator for the union of both groups. Before the merge actually takes place, one coordinator invites the other group's coordinator which then may accept or decline the offer to join. When a node does not receive a reply from its coordinator anymore it assumes the coordinator is down and forms a new group with only itself in it (Figure 2(b)) with the purpose of trying to merge with other groups when they become available. This is illustrated in

Figure 2. More than two groups merge by sequentially merging two groups. Usually it should be avoided that more than one group coordinator sends out an invitation at the same time, but in reality like in the example (Figure 2(c)) this sometimes happens anyway. This does not affect the algorithm since the groups will eventually merge and the order is not important.

Stoller pointed out a contradiction in Molina's Invitation algorithm in respect to his Assertion A4 in [7]. Essentially, Stoller gives the scenario of three nodes, 1, 2 and 3. Node 1 can communicate with 2 (bidirectional of course) and 2 with 3, but 1 not with 3. Assertion A4 however states that if there is a set R of nodes which all can communicate which each other (and there is no superset to R) then the leader of every node in R is in R itself. When thinking about it for a minute it should be quite obvious that this is not always true. The author gives a scenario in which node 2 (the one which can communicate with both node 1 and 3) eventually forms a group with node 1, with node 1 being the leader. Node 3 is part of a singleton group, even though there is a superset of that node ($\{2, 3\}$, which can communicate with each other), which in turn means that node 2 should therefore have a leader which is in $\{2, 3\}$. This is where the assertion is violated. In the paper, an alternate version of the assertion and the algorithm, employing graphs and clique covers, is given.

I will present the important procedures in the Invitation Election algorithm in the same Java-Like language I used before with the Bully Election Algorithm.

Listing 2: The variables in the Invitation Procedures

```

1 String status, task;
2 int id, groupid, counter, node, seconds, setMax;
3 boolean[] neighbours, coordinators;

```

These variables are mentioned to clarify the following code.

Listing 3: The Invitation Check Procedure

```

1 private Check() {
2   if (status == "Normal" && coordinator == id) {
3     for (node=1; node<nodesTotal; node++) {
4       if (node.AreYouCoordinator() == "Yes") {
5         coordinators[node] = true; }
6     }
7     for (int i=n; i>=1; i--) {
8       if (coordinators[i] == true) {
9         setMax = i; } }
10    if (id < setMax) { sleep setMax-id; }
11    self.Merge(coordinators);
12  }
13 }

```

Listing 3 shows the Check procedure, which is called in regular intervals by the operating system of the nodes. If the node considers itself its coordinator, it first calls all the nodes in the system, sending them the AreYouCoordinator message. If a node answers that it is, it is added to (or rather flagged in) the array coordinators. It then determines the coordinator in this array with the highest id number, if the id found is bigger than its own it waits an amount of time which should give the other coordinator the chance to call a merge first. After that it starts the Merge() itself, supplying the coordinators array as a parameter.

Listing 4: The Invitation Merge Procedure

```

1 private Merge (boolean [] coordinators) {
2   status = "Election";
3   self.Stop();
4   counter++;
5   groupid = id . counter;
6   coordinator = id;
7   for (node=1; node<nodesTotal; node++) {
8     if (coordinators[node]) {
9       node.Invitation(id, groupid);
10    else if (neighbours[node]) {
11      node.Invitation(id, groupid); }
12  }
13  neighbours = [];
14  sleep(seconds);
15  status = "Reorganization";
16  /* Here the reorganization takes place.
17   * After that the "neighbours" array
18   * contains the nodes which
19   * accepted the invitation. */
20  for (node=1; node<neighbours.length; node++) {
21    if (neighbours[node]) {
22      if (node.Ready(id, groupid, task) != "Ok") {
23        self.Recovery(); } } }
24  status = "Normal"
25  }

```

The Merge procedure is presented in Listing 4. It is the centerpiece of the invitation algorithm, called by the aforementioned Check procedure with the purpose to unite (i.e. merge) the group the node that is calling the merge is in with other groups. It has a single argument, which is an array indicating which nodes in the system that are reachable (i.e. reply to messages) are coordinators, so that those coordinators can be invited. The array “coordinators” is different from the array “neighbours” because “coordinators” contains the coordinators in the system which obviously are not part of the own group because they were not reachable during the last election or because they simply decided not to accept the invitation last time. The

array “neighbours” in contrast contains the members of the current group (of which only one is a coordinator).

The node goes through the usual initiation for the election. It sets its status to “Election” and stops itself. Then it increases the counter by 1. The counter is something we need to guarantee unique group id numbers. The group-id is calculated by appending the counter to the id of the node (the . operator in my example does just this). In the next few lines the algorithm goes through the two arrays (they have both the length of “nodesTotal” because the number of nodes in the system is constant), the coordinators array which it was supplied with and the neighbours (which are different, as explained before). It sends out an invitation to all the nodes which are supposed to be up after which it waits for a certain amount of time for replies to the invitations. After setting status to “Reorganization” the reorganization itself takes place. This part of the procedure was intentionally left out by Molina. What is important is that after this reorganization, the array neighbours contains all the nodes which are in the newly formed group (i.e. which accepted the invitation and were flagged in the neighbours array) and the variable “task” contains the new task which will be distributed to the group members (including the list of neighbours). The last iteration sends the Ready message to all the nodes in the node group, containing the id of the coordinator, the group-id and their new task.

2.4 Practical considerations.

Looking at the implementation of the Bully and the Invitation Algorithm by Molina it is quite obvious that even the pseudocode (which was supposed to be simple) is difficult to read and does not translate very good to the real world. The first things that comes to mind when looking at the code is that all the nodes have the same set of methods and data fields, yet are independent objects. This very much resembles the structure of object-orientation. If we were to simulate a leader election algorithm in software (for testing and analyzing it) we could therefore do it very intuitively in an object-oriented programming language. A node inherits all the methods it needs to work. Because it is quite obvious which node called a method like Ready(), the cumbersome syntax *Ready(i, j, x)* could be abbreviated by calling *nodej.Ready(x)* and so on.

Leader election algorithms sometimes face the Byzantine Generals problem [4]. A node *i* can never be sure that a message it sent was received by node *j* and the other way around. For the Bully algorithm in our example we avoided that problem by making our network fail-safe by definition. This is as if the two generals were talking on the phone and ending their conversation with ”over-and-out”. With the Invitation-Algorithm where we do not take Assumption 8 and Assumption 9 for granted we are in the byzantine situation. However in our case we accept that we cannot reach certain nodes

which is not that important to us (unlike it is to the two generals who must attack simultaeneously).

3 Leader election in consumer products

3.1 Leader election in the IEEE 1394 Serial Bus (Firewire)

It might not be widely know it but there are leader election algorithms being performed by personal computers (and their peripherals) constantly. I would like to take the FireWire Serial Bus (IEEE 1394) as an example for such a case. It is a bus for connecting high-speed peripheral devices to the computer, which then automatically reconfigure themselves to adapt to the changed topology in the connection chain. The protocol responsible for automatically connecting the FireWire devices on a software level is specified in [2]. The terminology here is to call the devices connected to the bus “nodes” which have one or more “ports” which connect them to other devices. Each port can have only one connection. A node with only one connection is called a “leaf” and a node with more than one a “branch”. The protocol is called into action when a connection is added or removed (e.g. when you plug in a new FireWire device into your computer). The protocol creates a tree in the network and the root of that tree will be the leader. Nodes start sending their neighbours parent-requests. The one node which received parent-requests from all of its neighbours considers itself the leader. Sometimes it may happen that two nodes which received parent-requests from all their neighbours but one send each other a parent request at the same time. This scenario is described as “root-contention” [6]. They then both draw back their parent request and wait for a random amount of time. After the time has elapsed they check if there is an incoming parent request and if so accept it and are elected root, and if not they send out their own parent request again. Even if they pick the same “random” time, some node will eventually be the leader. The formal verification of this is given by Stoelinga and Vaandrager.

The protocol for processor failures in the Amoeba Distributed Operating System [3] is another good example of a classical leader election algorithm as described by Molina. In fact, it is loosely based on the Invitation Election Algorithm.

4 Conclusion

We looked at two different leader election algorithms in this paper. They were introduced by Molina in 1982 as a theoretical approach to the problem of mutual exclusion in a distributed system. Although the algorithms may seem overly simple (and the assertions for them to work rather unrealistic)

there are a lot of real scenarios where these algorithms do exactly what they are needed for. The example of the algorithm for the IEEE 1394 serial bus shows that for most systems, leader election algorithms do not have to be complicated in order for them to be effective as the algorithm can be naively started again if an election failed, resetting the system. The speed at which a new leader is elected (in the case of Firewire only a few milliseconds) is hardly noticeable to humans. It is not required to keep a system in a working state all the time. This however does not apply to medical or military applications or aviation. For these environments the algorithms have to deal with the possibility of node or communication failure and still handle the election leaving the system in a usable state during the time. The phrase “eventually a leader is elected” is unacceptable.

Considering consumer products a really important field for leader election has been opened by the development of wireless networks, where nodes communicate via radio waves. This in itself is not that groundbreaking or new, but what is more interesting is that today a big range of consumer mobile devices are equipped with numerous communication channels (WiFi, Bluetooth), most of which are capable of communicating not only with a base-station in a centralized way but possess the ability of inter-device connection, or peer-to-peer connectivity. It is not hard to see where this leaves us. What could be more random than a network which topology changes as people enter or leave certain areas [8], decide to connect to or disconnect from a cloud of devices [5], and nodes may not be able to communicate directly but must use a path of other participating nodes. Peer-to-peer (P2P) networks in general call for leader election. The vast amount of users in P2P networks with unreliable links or surfing-habits makes a solid and fast-reacting leader election algorithm mandatory. After all, it is not hard to see that we can not reset the whole system into reorganization mode every time one of a few hundred thousands users decides to join or leave. This is where distributed leader election will play a role.

The future will bring us even more networked environments. A lot of those will be thanks to the quick growth and easy availability of the Internet as well as the increasingly small size and low prices for mobile computers and the multitude of communication systems of those devices.

5 Acknowledgment

I would like to thank the people who proof-read this document and pointed out things that were confusing to the casual reader. I would also like to thank my advisor M. Neuhäüßer for the kind support.

References

- [1] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE TRANS. COMP.*, 13(1):48–59, 1982.
- [2] IEEE. *IEEE 1394 Standard for a High Performance Serial Bus*, 1995. <http://standards.ieee.org/catalog/olis/busarch.html>.
- [3] MF Kaashoek and AS Tanenbaum. Group communication in the Amoeba distributed operating system. *Distributed Computing Systems, 1991., 11th International Conference on*, pages 222–230, 1991.
- [4] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
- [5] N. Malpani, J.L. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 96–103, 2000.
- [6] M. Stoelinga and F. Vaandrager. Root contention in IEEE 1394. *Lecture notes in computer science*, pages 53–74.
- [7] S.D. Stoller. Leader Election in Asynchronous Distributed Systems. *IEEE Transactions on Computers*, 49(3):283–284, 2000.
- [8] K. Weniger and M. Zitterbart. IPv6 Autoconfiguration in Large Scale Mobile Ad-Hoc Networks.