A New Leader Election implementation

Hans Svensson
Dept. of Computing Science
Chalmers University of Technology
Gothenburg, Sweden
hanssv@cs.chalmers.se

Thomas Arts
IT University of Gteborg
Gothenburg, Sweden
thomas.arts@ituniv.se

Abstract

In this article we introduce a new implementation of a leader election algorithm used in the generic leader behavior known as gen_leader.erl. The first open source release of the generic leader [6] contains a few errors. The new implementation is based on a different algorithm, which has been adopted to fulfill the existing requirements. The testing techniques used to identify the errors in the first implementation have also been used to check the implementation we propose here. We even extended the amount of testing and used an additional new testing technique to increase our confidence in the implementation of this very tricky algorithm. The new implementation passed all tests successfully. In this paper we describe the algorithm and we discuss the testing techniques used during the implementation.

Categories and Subject Descriptors D.2 [*Software Engineering*]

General Terms Algorithms, Verification

Keywords Erlang, leader election, distributed systems, implementation

1. Introduction

Many distributed applications are easy to implement if one has one dedicated process to administer certain tasks. For example, one process could poll all attached hardware devices to determine the configuration of a distributed system, whereafter the other nodes may then consult this process for the configuration information. More generally, it is often useful to have a server process that is in charge of keeping a consistent view of an aspect of the system state. All nodes in the distributed system consult that server process if they want information about the system state or if they want to update the system state.

A dedicated server provides an easy way to introduce consensus, synchronization and resource allocation in a distributed system. The disadvantage with this solution is, though, that one introduces a single point of failure in the system. In a fault-tolerant setting, at least one stand-by node needs to be introduced. Taking that thought one step further, several stand-by nodes may be introduced, since that provides an even better protection against faults. With either one or more stand-by nodes, each stand-by node has the problem of detecting when to become the active node. In fact, the primary node (the one that is assumed to run the dedicated server if nothing goes wrong) also has the problem to determine whether it can actually take that role. This is caused by the fact that when this primary node starts, one of the stand-by nodes may already have decided that the primary node is dead and that it should run the server instead.

This problem of having several nodes competing to perform one central task is well-known and described in literature as *the leader election problem*. A solution to this problem is an algorithm that when its execution terminates, guarantees that a single node is designated as a leader and every node knows whether it is a leader or not. The leader is then assigned the role of the above described dedicated server.

At least since the early seventies leader election algorithms for all kind of settings have been described. Often these solutions are stated in form of a multi-processor machine with shared memory or by means of computers in a token-ring network. Most solutions assume a perfect world in which no failures occur. Some solutions assume possible failure of the communication others possible failure of the nodes. There are over $10,000$ articles on the leader election problem and it is not easy to find a solution among them that fits the Erlang context well.

In our case, we are interested in a solution that is fault-tolerant with respect to failing and restarting processes and failing and restarting nodes. We assume Erlang nodes to have reliable communication without lost messages (basically the TCP/IP setting in which all nodes can directly communicate with all other nodes in a reliable way). In the open source Erlang community there exists an implementation of a leader election algorithm [6]. This implementation is based on an article written by Singh [4], but contains numerous adaptations to the Erlang setting. The implementation originates from the work at Ericsson with the AXD 301 telecommunication switch, but has been rewritten and turned into the OTP behavior `gen_leader`. From a user point of view, the generic leader behaves like a generic server with callback functions like *call* and *cast*. The intended use is that of having one generic leader per node and clients access only the generic leader on their node. The generic leaders communicate with each other and forward all requests to the chosen leader.

Thorough tests have shown that the above mentioned implementation, unfortunately, contains errors (see [1] for details). In some rare circumstances, two leaders can be elected at the same time. In addition, there is a possibility that the election of a new leader stands in a deadlock. The system may run for years without showing any failure, but there is always the potential danger that one day the circumstances are exactly such that those faults occur.

After failing to repair the implementation we proceeded to make a new implementation based on another algorithm. The new implementation is based on the article 'Leader Election in Distributed

Erlang'05 September 25, 2005, Tallinn, Estonia.

Systems with Crash Failures' by Stoller [5]. Compared with Singh, Stoller takes a slightly different approach to the leader election problem, which seems to fit better into the Erlang setting. However, we still had to modify the algorithm, since it was designed for a completely different situation.

We took care to supply the same interface for this new implementation as defined for the original, incorrect, implementation [6]. However, due to the differences in the implemented algorithms the interface functions that return all alive nodes and the one returning all dead nodes, could not be provided. Apart from that the behavior of the new implementation should be, when viewed from the outside, the same as the behavior of the old implementation. Except for the failures!

We have tested the implementation thoroughly, using both the test method with abstract traces that revealed the errors in the original implementation [1], and Erlang QuickCheck [3] which is a property-based random testing tool. While testing the implementation we discovered and successfully corrected a number of errors we made in the implementation. The new version of gen_leader is available at
http://www.cs.chalmers.se/~hanssv/leader_election.

In Sect. 2 we explain the algorithm we have implemented, and the adaptations that were made to make the algorithm useful in this context. In Sect. 3 we describe the implementation and the testing of the implementation. We conclude with discussion in Sect. 4.

2. Algorithm

Sometime shortly after the first implementation [6] was written *Google* was used to search for the source of Singh's algorithm [4] (on which the implementation was based). During that search for leader election algorithms another interesting algorithm on leader election in distributed systems with crash failures by Stoller [5] popped up. It was judged to be a good, if not better, alternative to Singh's algorithm, but that had then already been implemented.

Much later, when an error was detected in the first implementation, and when we failed to repair this implementation based on Singh's article, we decided to try an implementation based on Stoller's algorithm. Here it is important to notice that the failure of Singh's algorithm lies solely in the problem of adapting the algorithm to the Erlang environment, not in the algorithm itself. As an example, Singh's algorithm only deal with one election round, in the gen_leader a new election should be initiated when the elected leader fail. It is often the case that algorithms described in articles have assumptions and preconditions that are not fulfilled by the target system, such as communication behavior and specific network topologies. It is also often the case that the target system requires additional functionality that is not included in the algorithm, such as interface functions and error handling. Therefore, changes to the algorithm are necessary. When dealing with complex algorithms, such changes are dangerous, since one easily introduces an error, which was exactly what had happened in the gen_leader case.

Stoller's algorithm, is based on a pre-known set of participating processes with a globally known priority order. The algorithm also depends on the fact that there exist a mechanism for detecting *inactive* processes, for this we can use the Erlang *Monitor*. The basic algorithm is both simple and elegant. When a process is started, it first checks whether a process with higher priority is active. If such a process exists, the process simply waits for one of those processes to become the leader. If, on the other hand, the present process is the active process with highest priority, the process itself tries to become the leader. Becoming the leader is done by making sure that all processes with lower priority either are aware of its existence or are inactive. When all processes with lower priority are informed, the process announces itself as the leader. Periodically, the elected leader polls the inactive processes, if one of the inactive processes is activated, the election process is simply restarted.

There are actually two different algorithms described in Stoller's article, one with synchronous message passing and one with asynchronous message passing. What is perhaps a bit surprising, and at the same time shows how difficult it is to select a good candidate algorithm for implementation, is that we choose the synchronous algorithm, even though Erlang has asynchronous communication. A more careful reading of the article reveals however, that the difference between the synchronous and the asynchronous algorithm lies mostly in how the failure detection works (how node failures are detected and reported). The Erlang monitor works in the same way as the failure detection with synchronous message passing. This shows that it is important to have a thorough understanding of the inner workings of the implementation language.

We illustrate in detail how the algorithm works by an example with three participation processes in Fig. 1. The processes are named A,B and C, with priority $A > B > C$, i.e. A has highest priority.

```
1. A,B and C are all activated at the same time
    - C: starts monitoring A and B,
      B: starts monitoring A.
    - A: no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
    - B: receives a 'halt'-message,
         replies with an 'ack'-message
    - A: receive 'ack' from B,
         starts monitoring C,
         sends a 'halt'-message to C
    - C: receives a 'halt'-message,
         replies with an 'ack'-message
    - A: receive 'ack' from C,
         all processes notified so A is the leader,
         sends 'ldr'-message to B and C
    - B,C: receive 'ldr'-message from A,
           accepts A as the leader.

2. A and B are active and A is the elected leader,
   C is activated.
    - A: periodically sends a 'norm'-message to C
    - C: receives a 'norm'-message from A,
         replies with a 'notnorm'-message
    - A: receives a 'notnorm'-message,
         restarts the election procedure,
         no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
         ... (as in situation 1)

3. A and C are inactive, B is active.
    - B: starts monitoring A
    - B: receives a 'DOWN,A'-message from monitor,
         no higher prioritized process alive,
         starts monitoring C,
         sends 'halt'-message to C
    - B: receives a 'DOWN,C'-message from monitor,
         all processes notified so B is the leader,
         sends 'ldr'-message to A and C

4. B and C are active and B is the elected leader,
   A is activated.
    - A: no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
         ... (as in situation 1)
```

Figure 1. Examples – Original behavior

```
2. A and B are active and A is the elected leader,
   C is activated.
    - A: periodically sends a 'norm'-message to C
    - C: receives a 'norm'-message from A,
         starts monitoring A,
         replies with a 'notnorm'-message
    - A: receives a 'notnorm'-message,
         sends a 'ldr'-message to C
    - C: receives a 'ldr'-message from A,
         accepts A as the leader
```

Figure 2. Examples – Situation 2 without re-election

Unfortunately, this algorithm does not behave as is required by a leader election in this case. The requirements for the leader election implementation is that (1) it should quickly elect a leader among the active participating processes, (2) the elected process stays the leader until it fails and (3) when the leader fails, a new process should be elected automatically. The algorithm presented by Stoller fulfills (1) and (3), but fails on (2). Instead whenever an inactive process is activated, a new round of elections is started, electing the process with highest priority as the leader. This is both time consuming and inefficient from a message complexity point of view, so in order to use this algorithm we have to change its behavior.

We made this change in two steps, first we changed the algorithm such that no new election would be started if a process with lower priority than the leader was activated. This change is fairly straightforward, and just requires a small modification to the behavior when a newly activated process is polled by the elected leader. Instead of restarting the election process, the newly activated process is informed of who the leader is. If we reconsider the examples in Fig. 1 situations 1, 3 and 4 are not changed, but in situation 2 we avoid a re-election and instead proceed as in Fig. 2.

In addition we wanted to do something similar when a node with higher priority than the present leader is activated. This however turned out to be much more complicated. The reason for the complexity is the fact that a node with high priority is likely to conclude that there are no processes active with a higher priority and therefore initiates a new election. (Note however that this behavior is required, otherwise an election would never be initiated in the first place.) The basic trick here is to make sure that a process that knows who the leader is will not surrender to the newly activated process, instead it sends a reply saying who (he thinks) is the leader. In this way, also a newly activated process with high priority can be informed of who is the leader. The newly activated process finally confirms the leadership with the leader. Nevertheless, there are still many things that can go wrong, especially in situations where the present leader fails in the middle of the information phase. If we yet again reconsider the examples in Fig. 2, we see that situations 1 and 3 work as before, but as expected we do not get a re-election in situation 4. This can be seen in Fig. 3

We also made some changes that did not affect the observable functionality, but which reduced the number of messages sent by the system.

3. Implementation and Testing

We first implemented the algorithm as a gen_server behavior, in order to quickly evaluate if it was working as intended. Having corrected several minor errors, most of them related to messages that were not treated in all situations, we felt fairly sure that this algorithm would work inside the gen_leader. Replacing the old algorithm was relatively easy, the only problem was the separation into a *safe_loop* (where the process execute during elections) and a working *loop* (where the process execute when a leader is elected

```
4. B and C are active and B is the elected leader,
   A is activated.
    - A: no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
    - B: receives a 'halt'-message,
         replies with an 'hasLeader,B'-message
    - A: receive 'hasLeader,B' from B,
         starts monitoring B,
         sends an 'isLeader'-message to B
    - B: receive 'isLeader' from A,
         sends 'ldr'-message to A
    - A: receive 'ldr'-message from B,
         accepts B as the leader.
```

Figure 3. Examples – Situation 4 without re-election

and which basically is the same as the loop in a generic server). This separation made it possible to do some simplifications in the message receiving code, and introduced a couple of new errors.

Another problem is the fact that the new algorithm is fundamentally different from the old one. This leads to some problems when trying to be compatible with the existing implementation. In particular we realized that the *query*-functions alive (which returns all active participating processes) and down (which returns all inactive processes) could not be implemented. This is because the new algorithm does not keep track of this information at all times, so the information returned by these functions is not reliable. Except from this, we managed to implement the algorithm without changes to the interface.

Leader election is a well-known and clearly defined problem, which means that the requirements are also well defined: (1) Eventually, a leader should be elected, and (2) At most one of the participants is considered the leader. These properties are also stated in Stoller's article [5]. We tested the implementation with two different methodologies, first we used the method with abstracted traces, as we describe in [1] and second we used Erlang QuickCheck presented in [3].

3.1 Testing with trace recording

The built-in trace functionality in Erlang is a very useful tool when testing an implementation. However, the raw trace data has a tendency to get very verbose, containing lots of events and also a lot of data per event. Manual inspection of traces is therefore often both tedious and time consuming, and alternative approaches have been proposed. In [2], one approach is presented where *abstraction functions* are applied to state based trace data, in order to remove unnecessary data and reduce the state space. The state space is reduced since different concrete states will be reduced to the same abstract state when the abstraction function is applied. While collapsing different concrete states to the same abstract state, cyclic behaviors can be detected. The abstract state space is also visualized, something that gives a good intuition about the inner workings of an implementation.

This *abstract trace* approach is taken even further in [1], where we demonstrate the effectiveness of the method by testing the first leader election implementation [6] based on Singh's algorithm. In [1] we also introduce a small language for constructing abstraction functions, as well as checking LTL-properties for the abstract state space. To test the leader election implementation we stimulated the system by arbitrarily killing and reviving nodes, and by arbitrarily delaying messages sent between processes.

This test method initially revealed a couple of trivial implementation errors, but when those were corrected all tests where executed without errors. That is the new implementation passed all the tests, the same tests during which the previous implementation

failed in two cases. However, this test method does not change the scheduler in the runtime system, and since the Erlang scheduler is deterministic, it seemed quite possible that there exist execution paths not exercised by the trace recording testing technique.

3.2 Testing with QuickCheck

Therefore we decided to also test the implementation with Erlang QuickCheck, presented by Arts and Hughes in [3]. QuickCheck is a property-based tool for random testing. Developers write properties in a restricted logic, and then invoke QuickCheck to test the property in a large number of cases. QuickCheck tests concurrent programs by collecting a trace of events, which should have the properties the developer specifies. The events are defined by instrumenting the code under test with calls to the QuickCheck function `event`. QuickCheck delays these calls randomly, thus in effect overrides the Erlang scheduler and forces a random schedule on the system under test. This can elicit faulty behavior that would appear only very rarely with the normal scheduler, which is exactly what we want to test here. Testing the leader election implementation was done by randomly killing and reviving leader election processes.

Using QuickCheck to test the second implementation, we could not produce any trace where the properties were violated. Nevertheless, and much to our surprise, we could observe some faulty behavior, namely that a leader election process crashed unexpectedly from time to time. This did not lead to any faulty behaviour, but it indicated that something was wrong.

Closer analysis revealed a very tricky error, which would have been extremely unlikely to be found without control of the scheduling. The problematic situation occurs whenever a process A is about to contact another process B. To do this in a controlled way, process A first request a monitor on process B before sending the message. What can occur now is that process B is down when process A requests the monitor, but alive just some time later when process A send the message. In this case, process A receive both a failure-notification and a message reply. This situation was overseen in the implementation and lead to a crash. Luckily, the error could easily be corrected.

In this example we can see how important it is to have control of the scheduling, since this situation occurred frequently (like once every 150 tests) while testing with QuickCheck, but could not at all be observed when we tested the implementation with the trace recording technique.

3.3 Coverage

When working with test methods, the issue of coverage is central. Coverage should provide a measure of how exhaustively one has exercised the system, and is therefore important when evaluating the results of testing. Though it is very rare that a coverage measure can tell when we have tested enough, rather the coverage measure will warn of potential situations when we have *not* tested enough.

In [1] we discuss some coverage measures for the trace recording technique, but those measures mostly deal with quantities in the abstracted state space and are hard to compare with the QuickCheck tests. Instead we choose to look at how many nodes that where killed, and at what stage in the election process the node were killed.

In Tab. 1 we can see: the coverage result (labeled `QuickCheck`) for a QuickCheck run with 5 nodes, average numbers (labeled `QC average`), and as a comparison results for a run with the trace recording technique (labeled `Trace rec`). The first column shows the total number of killed nodes, second column the number of nodes killed during an election, third column the number of nodes killed when elected as leader, and fourth column the number of nodes killed when being surrendered to a leader. In the coverage results we can note a difference between the two techniques, since we do not influence the scheduler in the trace recording technique it is quite rare that we manage to kill a node in the middle of the election process (merely 4% of the kills) compared to the QuickCheck approach where this happens a lot more frequently (almost 25 % of the kills).

Other coverage measures that are often discussed include *code coverage* and *path coverage*. Code coverage is a very basic coverage measure, that only measures whether (or how many times) a certain line of code has been executed. This simple measure is useless here, since it is the complicated interaction of several different instances of the implementation that is studied. Path coverage is therefore more interesting, since it measure how many different paths that has been taken through the code. Unfortunately however, path coverage is hard to define in a functional language such as Erlang since paths does not exist in the same way as for an imperative language like C or Java.

4. Discussion

Implementing a new leader election algorithm was very interesting from more than one point of view. Not only is it a challenging intellectual problem, it also highlights several interesting and problematic situations that may occur in industry. For the majority of algorithmic problems that arise in practical software development today, there exist books and papers describing possible solutions. For a software engineer, it is often a non-trivial task to first find the right source of information, and then adopt the described solution to the specific setting at hand. Often software errors are made because (1) the wrong algorithms were chosen, or (2) the right algorithms were adapted in the wrong way.

Why is it such a hard problem to choose a good algorithm? Algorithm descriptions, and then especially formally verified algorithms, are often presented in a theoretical way and work only in a specific setting. It is often the case that the prerequisites stated in the article do not fit into the implementation language. It is also often the case that changes must be made to the algorithm in order to fulfill the specific requirements, such changes include error-handling and interface. Therefore it is a hard but also crucial problem to select a good algorithm. It is a task that require not only a thorough understanding of the problem, but also a good insight in the inner workings of the implementation language.

One example of this is the error found with QuickCheck, our erroneous implementation closely followed the algorithm in the paper. Nevertheless, the implementation turned out to be incorrect. Does this mean that the same error is also present in the article? No, Stoller's article [5] is not very precise about the semantic assumptions made regarding link requests between processes. Therefore, one has to assume that there is a difference in how the monitoring works, and that this is the source of the error. This clearly shows the difficulties of bridging the semantics from the article, where underlying assumptions often hide important and problematic issues, to the implementation language.

Verifying fault-tolerant distributed systems is an extremely difficult task. It is difficult and time consuming to use verification techniques such as model checking, instead testing is the commonly used method. Here, we have used two different testing techniques. In many ways these techniques are rather similar; both use random testing, and both methods use traces. The big difference between the methods are the way we control the scheduler, which in turn affects the execution paths explored in the tests. The concrete test results show that both methods are useful, we found an error with QuickCheck that was not found with the trace recording technique. On the other hand when writing the implementation it was very useful to see the visualizations from the trace recording technique, both to correct errors and to gain insight in the implementation.

	Killed nodes	in election	as leader	surrendered
QuickCheck	1601	379	102	1120
QC average	19.3	4.6	1.2	13.4
Trace rec.	101	4	11	86

Table 1. Coverage results

Our work resulted in a new implementation of the generic leader behavior. This implementation is thoroughly tested and no errors could be identified. For some very critical applications, one might want to invest in a formal verification of the presented application, but most applications would not require such thorough mathematical analysis.

5. Acknowledgments

Thanks to Ulf Wiger, co-author of the first gen_leader implementation, and John Hughes, implementer of Erlang QuickCheck. Also thanks to Koen Claessen for providing valuable comments and ideas.

References

[1] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in erlang. In *Lecture Notes in Computer Science*, volume 3395, pages 140 – 154. Springer, Feb 2005.

[2] T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 16–23. ACM Press, 2002.

[3] T. Arts and J. Hughes. Erlang/quickcheck. In *Ninth International Erlang/OTP User Conference*, Nov. 2003.

[4] G. Singh. Leader election in the presence of link failures. In *IEEE Transactions on Parallel and Distributed Systems, Vol 7*. IEEE computer society, 1996.

[5] S. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[6] U. Wiger. Fault tolerant leader election. http://www.erlang.org/.