Analyzing Eventual Leader Election Protocols for Dynamic Systems by Probabilistic Model Checking

Jiayi Gu[1], Yu Zhou[1,2](✉), Weigang Wu[3], and Taolue Chen[4]

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China
{gujiayi,zhouyu}@nuaa.edu.cn

[2] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

[3] Department of Computer Science, Sun Yat-sen University, Guangzhou 510006, China
wuweig@mail.sysu.edu.cn

[4] Department of Computer Science, Middlesex University, London, UK
t.chen@mdx.ac.uk

Abstract. Leader election protocols have been intensively studied in distributed computing, mostly in the static setting. However, it remains a challenge to design and analyze these protocols in the dynamic setting, due to its high uncertainty, where typical properties include the average steps of electing a leader eventually, the scalability etc. In this paper, we propose a novel model-based approach for analyzing leader election protocols of dynamic systems based on probabilistic model checking. In particular, we employ a leading probabilistic model checker, PRISM, to simulate representative protocol executions. We also relax the assumptions of the original model to cover unreliable channels which requires the introduction of probability to our model. The experiments confirm the feasibility of our approach.

Keywords: Dynamic systems · Verification · Leader election · Probabilistic model checking

1 Introduction

Recent years have witnessed an increasingly booming interest in dynamic systems [1–4] since they have been widely applied in many fields such as distributed systems [5, 6], neural networks [4] etc. Accordingly, problems which have been studied in the traditional distribution computing are revisited in this new setting. For example, eventual leader election protocols have gradually become a focus in the dynamic system. However, to the best of our knowledge, most existing works related to leader election protocol design are based on static systems. Indeed, it is very difficult to design such a protocol/algorithm and analyze its properties with the presence of uncertainty in the context of dynamic systems.

Model checking is a traditional approach to verifying a design artifact against certain specification. Because the verification is conducted automatically, and a counter-example can be generated if the specification is unsatisfied to reveal potential mistakes in the design, model checking has gained wide popularity and thus become the mainstream verification technique [7, 8]. To cope with the underlying uncertainty, probabilistic model checking has been proposed. Generally here the transitions are extended with probabilities. It provides a rigorous methodology for the modeling and analysis of a wide uncertainty [7, 9].

Motivated by the above considerations, we adopt probabilistic model checking to analyze some properties, such as scalability and efficiency related issues, of leader election protocols for dynamic systems featuring a high degree of uncertainty. PRISM [10], a leading open-source-probabilistic model checker, is used to construct models of various categories of systems and analyze their random or probabilistic features. It can be applied in many fields, such as communication protocols, distributed algorithms or any other systems of specific subjects like biology. PRISM provides a simulator to generate sample paths through a designed model. Via simulator, one can clearly observe the executions of the model and the exact time of achieving the target designed at the beginning of experiment. Such mechanism is very useful for our experiments which will be introduced in the subsequent part of this paper.

The main contributions of this paper are as follows. Firstly, we construct a model based on the existing protocol [2]. Through the PRISM simulator, we can analyze the average cost of steps of electing a leader. Moreover, we also give a scalability analysis for the protocol which has not been discussed in the original work. Secondly, we extend the assumptions of the original protocol and consider the unreliable message transmitting in channels. Particularly, we add the probability to verify the hypothesis that the value of this probability is related to the efficiency of leader election.

The rest of the paper is structured as follows. In Sect. 2, we briefly describe related concepts of dynamic systems and give an introduction to the processes in the system and the way of communication in such systems. In Sect. 3, we propose a modified eventual leader election algorithm for dynamic systems based on the work proposed in [2]. Section 4 presents model design and two relevant experiments. We discuss related work in Sect. 5 and finally conclude the paper in Sect. 6.

2 Dynamic System and Assumptions

The concept of the dynamic systems is described, in a nutshell, as a kind of distributed system with processes entering and leaving the systems dynamically [3]. Many systems can be regarded as a particular instance of dynamic systems, for example, peer-2-peer systems [11], wireless ad hoc networks [12] and etc. In addition, as mentioned in [2], it is obvious that there are certain similarities between traditional static systems and the dynamic ones. Hence it is not uncommon that some researchers have been attempting to adapt the protocol designed for the static system for the dynamic system rather than to design it from the scratch. In brief, a dynamic system model can be extended from its static counterpart and the work needs to be done is to provide an adaptation of the protocol designed for the static model.

2.1 Processes in a Dynamic System

As defined in [2], the dynamic system consists of infinitely many processes, but the number of processes of each run is finite. In other words, no matter what the integer value *n* is, there are more than *n* processes for all runs. In contrast, for each run, there is always a bound on the number of processes. However, despite the bound in each run, a protocol has no access to that bound since it changes over runs. Because of this, there is no existing protocol designed for the model with an upper bound on the amount of processes which can be demonstrated in [5, 13].

Each process has its own unique identity. For example, a process denoted p_i means a process with its identity *i*. However, a process will be assigned with a new identity as a new process when it re-enters the system. Below we use some notations as follows:

up(t): at time $t \in N$, the set of processes existing in the system. These processes have joined the system before time *t* and they have not crashed or left before time *t*.

$$STABLE = \left\{ i | \exists t : \forall t^{'} \geq t, p_i \in up(t') \right\}.$$

STABLE is the set of processes which will never crash or leave since enter the system. We note that this set is similar to its counterpart in the static model, i.e., the set of *correct* processes. Recall that in the static model, if a process does not crash during a run, it is considered to be a correct one; otherwise, it is considered as faulty. Moreover, in the static model, there is an integer *n-f* which denotes the number of correct processes and guarantees these processes not to be blocked forever. Similarly, in the dynamic model, a value α is proposed to define the number of static process and prevent processes from blocking forever. Therefore, we have the following correspondence between the static model and the dynamic model, i.e., *n- f* corresponds to α, and p_i corresponds to $i \in STABLE$.

2.2 Communication in the Dynamic System

All the processes in the dynamic system communicate with each other via the query-response primitive. Based on the existing work on the leader election in dynamic systems [2, 3], we describe such query-response primitive as follows:

(1) Once a process broadcasts a query message, then all the live processes in the system (including the sending process itself) can receive it.
(2) The process which has sent a query message will keep waiting until it has received "enough" messages from the other processes. It should be underscored that "enough" means a specific number α mentioned before.

More specifically, each process broadcasts a query message within the system. To avoid the condition that process blocks forever, only α responses are waited for the process which has sent the query message. (The number of α is defined previously.) During this period, we define a response that arrives among α responses p_i is waiting for as winning response, and similarly, the set of processes from which p_i has received a winning process to its last query terminated before or at time t is considered as $winning_i(t)$.

In light of the above considerations, the assumption regarding the query response pattern can be formulated $MP_{ds\psi}$:

In the dynamic system, there are time t, process p_i that $i \in STABLE$, and set Q of processes such that $\forall t^{'} \geq t$, we have:

(1) $Q \subseteq up(t^{'})$;
(2) $i \in \bigcap_{j \in Q} winning_j(t^{'})$; and
(3) $\forall x \in up(t^{'}) : Q \bigcap winning_x(t^{'}) \neq null$.

Intuitively, after time t, there exist a subset Q of a universal set $up(t^{'})$ of the dynamic system, and a stable process denoted by p_i. For every process in Q, p_i always belongs to the intersection of their $winning_j(t^{'})$. And simultaneously, for every process existing in $up(t')$, the intersection of Q and $winning_x(t^{'})$ is always not empty.

3 The Eventual Leader Protocol and PRISM Model Design

The leader election protocol is based on the existing work [2]. In our paper, we borrow the idea from the software engineering community and re-examine the protocol model through probabilistic model checking. Particularly, we utilize PRISM model checker to conduct the analysis which is lacking in [2]. Such an analysis allows us to investigate the protocol from a different perspective. More importantly, we extend the model by relaxing crucial assumptions previously made on the environment (i.e., the communication is reliable), and perform a considerably more detailed quantitative analysis.

3.1 Algorithm Introduction

Generally, the original protocol consists of four tasks. For details, we refer interested readers to [2]. In this subsection, we give a very brief introduction to these four tasks.

The key function of task1 is to diminish the size of the set of candidate leader denoted by *trust*. In task2, it reveals that when the process has received a *QUERY* message from other processes (besides itself), it will send a *RESPONSE* message immediately. In the original protocol [2], the paper posed a very strong assumption that the underlying channel is reliable, i.e., no message loss is allowed. However, in many cases, this is an unrealistic assumption as the underlying network could usually only provide best-effort delivery. Whether or not the *RESPONSE* message can reach its destination (the process which sends a *QUERY* message) cannot be determined in advance. Here we use the concept of probability to address this issue. Namely, we introduce a parameter named *ratio_suc* to evaluate the probability of sending a *RESPONSE* message and making it successfully reach to its destination. In addition, the responsibility of task3 is to update the *trust* set by comparing the *log_date* when a certain process receives a *TRUST* message from other processes. And it also supervises its *trust* set to ensure it is not empty. On top of it, the duty of task4 is to modify the value of leader according to its *trust* set.

To be more specific, a process p_i is always keeping such working cycle as follows.

```
while(true)
    broadcast $QUERY(i)$ to the whole system;
    wait until $RESPONSE(j,rec\ from_j)$ received from $\alpha$ processes;
    $RECFROM_i$ = the union of $rec\_from$ of those senders;
    $trust_i = trust_i \cap RECFROM_i$;
    $rec\_from_i$ = the set of the senders which send $RESPONSE$;
    if $trust_i$ is modified
        broadcast $TRUST(trust_i,log\_date_i)$ to the whole system;
    endif
endwhile
```

Algorithm 1. task1

```
upon $QUERY(i)$ is received from $p_i$
    $ratio\_suc$ : send $RESPONSE(j,rec\_from_j)$ to $p_i$;
```

Algorithm 2. task2

Firstly, it sends a *QUERY* message and then it keeps waiting until α *RESPONSE* messages have been received. Next it updates $RECFROM_i$ by computing the union of the rec_from_i of all the processes sending *RESPONSE*. Afterwards the $trust_i$ should be modified by set intersection. Moreover, it updates its rec_from_i set based on the ids of those processes which have sent *RESPONSE* messages. If this value has been changed, then process p_i should broadcast another message named *TRUST* to the system so as that all processes in the system can adjust their leader accordingly as shown in Algorithm 3.

The operation of every process is the same as the way of p_i introduced above. After some time, a unique leader will be eventually elected once all trust sets keep stable (unchanged) and the identity of leader of all processes point to the same value.

In addition, each component of data structure of a process is introduced as below and its value should be initialized. We use the data structure of process p_i as an example.

Among all variables, rec_from_i is the set of processes where p_i receives *RESPONSE* messages. $RECFROM_i$ is the union set of rec_from. And $trust_i$ is the set of candidate-leaders. Besides, log_date_i is a logical time defining the age of $trust_i$. Moreover, $leader_i$ is the leader of p_i.

All variables should be initialized as follows.

```
$log\ date_i = 0$ ;
$trust_i = \pi$;
$leader_i = i$ ;
$RECFROM_i = \pi$ ;
```

```
upon TRUST($trust_i$,$log\_date_i$) is received from $p_i$
    if $log\_date_i == log\_date_j$
        $trust_j = trust_j \cap trust_i$;
    else if $log\_date_i > log\_date_j$
        $trust_j = trust_i$;
        $log\_date_j = log\_date_i$;
    endif
    if $trust_j$ == null
        $trust_j = \pi$;
        $log\_date_j = log\_date_i$;
    endif
```

Algorithm 3. task3

```
if $trust_i$ == null or $trust_i == \pi$
    $leader_i = i$;
else
    $leader_i$=min($trust_i$);
endif
```

Algorithm 4. task4

3.2 PRISM Model

Following the above descriptions, we model the protocol in PRISM. In order to demonstrate the procedure clearly, we assume that the system consists of four processes and the value of α is set to be three.

Data Structure The data structure in the model consists of global variables and local variables. For the former, we use four reply counters and a signal counter. Every reply counter corresponds to a process and its target is to count the number of received *RESPONSE* messages. And for the whole system, there is a signal counter aiming to monitor whether or not the leader has been elected. All variables should be initialized to be 0.

$$\text{global } reply\_i : [0..\alpha] \text{ init } 0;$$

$$\text{global } signal : [0..1] \text{ init } 0;$$

For the latter, the local variables of every process are classified into three categories. In the first category (shown in Listing 1.1), each variable ranges over from 0 to *N* and is initialized to be *N*. It should be noticed that *N* here is equal to ($2^n - 1$) where n is the number of the processes in the system. Actually in our example, its value is set to 15 ($2^4 - 1$). The reason will be made clear later. In the second category (shown in Listing 1.2), each variable is of Boolean value and the role is to classify whether or not the required messages have been received. In the third category (shown in Listing 1.3), every variable is a counter which is used to increase the values of execution steps or any other similar information. Therefore, $process_i$ in the system has its data structure as below.

Listing 1.1. First category

```
rec_from_i     :  [0..15] init 15;
RECFROM_i  :  [0..15] init 15;
trust_i          :  [0..15] init 15;
new_trust_i    :  [0..15] init 15;
num_i            :  [0..15] init  15;
a_i               :  [0..15];
b_i               :  [0..15];
c_i               :  [0..15];
```

Listing 1.2. Second category

```
query_i          :  [0..1]  init  0;
response_i1    :  [0..1]  init  0;
response_i2    :  [0..1]  init  0;
response_i3    :  [0..1]  init  0;
response_i4    :  [0..1]  init  0;
```

Listing 1.3. Third category

```
sign_i1           :   [0..1]  init  0;
sign_i1           :   [0..1]  init  0;
sign_i1           :   [0..1]  init  0;
sign_i1           :   [0..1]  init  0;
s_i                :   [0..6]  init 0;
log_date_i       :   [0..m] init 0;
leader_i          :   [1..4]  init i;
```

In the first category (shown in Listing 1.1), *rec_from_i* is a decimal number which in fact should be converted to a binary string. This string represents a set of processes sending *RESPONSE* messages. And similarly, *RECFROM_i* is the union of *rec_from* of the processes in *rec_from_i*, *trust_i* is a candidate leader set and *new_trust_i* is a temporary *trust_i*. Moreover, *a_i*, *b_i* and *c_i* are also temporary variables which store the variables participating in the AND and OR operations.

In the second category (shown in Listing 1.2), *query_i* denotes whether or not *process*$_i$ has sent *QUERY* message. And *response_ij* represents whether *process*$_i$ has sent the *RESPONSE* message to *process*$_j$.

In the third category (shown in Listing 1.3), *sign_ij* is used to record the steps, despite the fact that it has only two choices. *s_i* is the same as *sign_ij* which records the execution steps of every process. And *log_date_i* is a log variable used to note the logical time of *trust_i*. And obviously *leader_i* is the leader belonging to *process*$_i$.

Key Procedures. According to the given algorithm, we design PRISM modules for every process. The algorithm is shown in Sect. 3.1. It should be noticed that we assume that α here is equal to three for illustration.

We note that since there are no APIs or relevant methods supporting the set operations in PRISM, we translate two sets to binary strings and then use these two strings to complete the AND and OR operations. Evidently they are equivalent to the intersection and union operations of sets respectively.

Listing 1.4. calculating *rec_from_i*

```
[ ] (signal=0)&(reply_i=3)&(s_i=1)
−>(rec_from_i'=(response_1i=1?1:0)+...+(response_4i =1?8:0))
&(a_i'=(response_1i=1? rec_from_1 : rec_from_2))
&(b_i'=(response_1i=1&response_2i=1)? rec_from_2 : rec_from_3)
&(c_i'=(response_1i=1&response_2i=1&response_3i=1)? rec_from_3 : rec_from_4 )&(s_i'=2);
```

In brief, Listing 1.4 demonstrates the procedure of calculating *rec_from_i*. Once the guard is satisfied, *rec_from_i* will change its value by means of given formula. The guard consists of three conditions: (1) whether or not the leader has been elected (denoted by *signal* = 0); (2) whether or not *reply_i* equals to three (denoted by *reply_i* = 3); (3) *process*$_i$ keeps state1 (denoted by *s_i* = 1), and then *rec_from_i* calculates its value. For example, if *process*$_i$ has received *RESPONSE* messages from *process*$_1$, *process*$_3$ and *process*$_4$, then *rec_from_i* will change its value to 13 since *response_1i* is 1, *response_3i* is 4 and *response_4i* is 8 and consequently the sum of them is 13. Simultaneously, *a_i* will change to *rec_from_1*, *b_i* will adjust to *rec_from_3* and *c_i* will be modified by *rec_from_4*.

Listing 1.5. calculating *RECFROM_i*

```
[ ] (signal=0)&( s_i =2)
−>(RECFROM_i'=max(max(mod( a_i ,2) ,mod( b_i ,2)) ,mod( c_i ,2))
+max(max(mod( floor (a_i /2) ,2) ,mod( floor (b_i/2) ,2)) ,mod( floor (c_i/2) ,2))*2
+max(max(mod( floor (a_i /4) ,2) ,mod( floor (b_i/4) ,2)) ,mod( floor (c_i/2) ,2))*4
+max(max( floor (a_i/8) , floor (b_i/8)) , floor (c_i/ 8))*8)&( s_i'=3);
```

The information modification regarding to *RECFROM_i* is demonstrated in Listing 1.5. It should be pointed that *RECFROM_i* compares with every digit of *a_i*, *b_i* and *c_i*, and then select the maximum of them as the digit. After that, it multiplies the weight of every digit and sums them. The variable *trust_i* has the similar solution.

Listing 1.6. broadcasting *TRUST*

```
[ syn ] (signal =0)&( s_i =4)&( new_trust_i != trust_i) &( log_date_j=log_date_i)–> solution1 ;
[ syn ] (signal =0)&( s_i =4)&( new_trust_i != trust_i ) &(log_date_j>log_date_i)–> solution2 ;
[ syn ] (signal =0)&( s_i =4)&( new_trust_i != trust_i ) &(log_date_j<log_date_i)–> solution3 ;
```

The algorithm regarding to *TRUST* has three branches (shown in Listing 1.6). Each time, one of them must be executed. Considering this trait, we use a synchronization sign to tackle this issue. The above demonstration can successfully fulfill the mission.

Listing 1.7. getting *leader_i*

```
[ ] (signal=0)&((CH1=CH2)&(CH2=CH3)&(CH3=CH4))–>(signal '=1);
[done] (s_i !=6)&((CH1=CH2)&(CH2=CH3)&(CH3=CH4) ) | (signal =1) –> (s_i'=6);
```

The unique leader of the system will be elected once all leader_i points to the same objects (shown in Listing 1.7). And the time after the unique leader has been elected, it always keeps the status that "elected". Therefore, we design the solution above.

4 Model Design and Experiments

Usually, users are interested in whether or not the protocol can elect a unique leader in the dynamic system as fast as possible, so the efficiency is a crucial concern. Considering the characteristic of the system and analyzing several possible factors, we make assumptions that the time of electing a unique leader of the system is related to the number of processes in the system and also the channel reliability of processes among the whole system communicating with each other via sending query-response primitive messages.

4.1 Scalability of the System

Various factors may influence the efficiency of election one of which is the number of processes in the system. To verify this hypothesis, we attempt to design an experiment of three systems each of which with different number of processes. Besides, for each system, any other variables should be fixed, such as the probability of successfully sending messages.

In our experiments, we consider three systems with three, four and five processes respectively. For each of them, we repeat the simulation 100 times and then compute

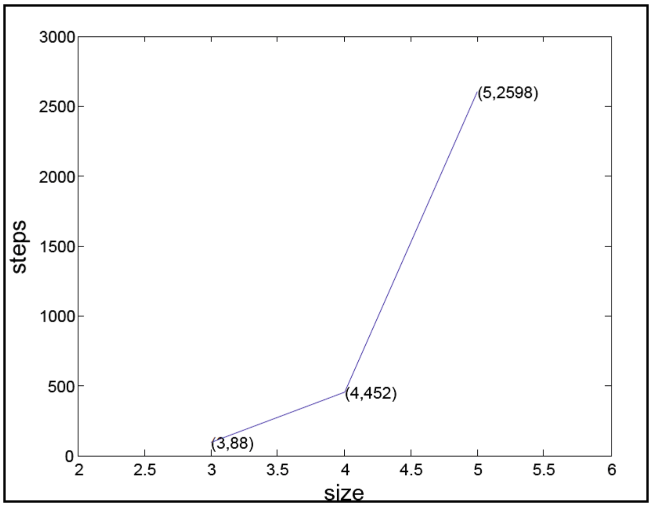

Fig. 1. Average steps of execution

the average steps of execution they spent on electing a leader. We set that the probability of sending *RESPONSE* messages is 1.0 because this time we only concentrate on the influence of the size of system rather than other factors. Based on the above setup, we record the results of each system as follows:

In Fig. 1, apparently with the system size increasing, the average number of execution steps that system spends on electing a leader grows rapidly. From the first experiment regarding the system consisting of three processes, the number of steps is around 88 on average. While adding a new process to the system, the average number grows to 452. When we continue to add another process, the result reaches 2598, approximately 5 times of the former one. In Fig. 2, we use a box plot to illustrate that with the increasing size of the system, the variance of the number of execution steps grows rapidly as well. Therefore we conclude that the size of the system must be controlled well. Otherwise, leader election in the dynamic system will be too costly to be affordable.

4.2 Unreliable Channels

As mentioned earlier, processes communicate with each other by query-response messages, and the probabilities of successfully sending messages, especially sending *RESPONSE* messages, are related to the efficiency of electing a leader in the dynamic system. In order to verify this assumption, we vary our model with three different probabilities capturing the channel reliability, i.e., the ratio of successfully sending *RESPONSE* messages among different processes.

Recall that the probabilities are introduced in task2. And the parameter denoted by *ratio_suc* means the ratio of successfully sending messages. We adopt three different values of *ratio_suc* - 0.5, 0.7, 0.9 - respectively to demonstrate the fact that the efficiency

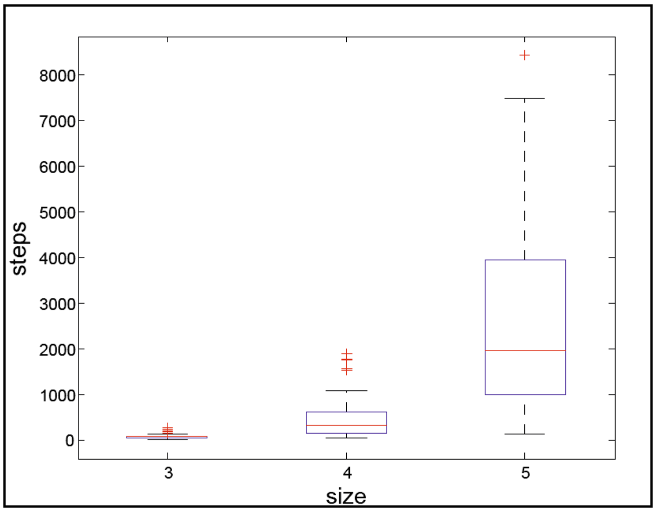

Fig. 2. Distribution of execution steps

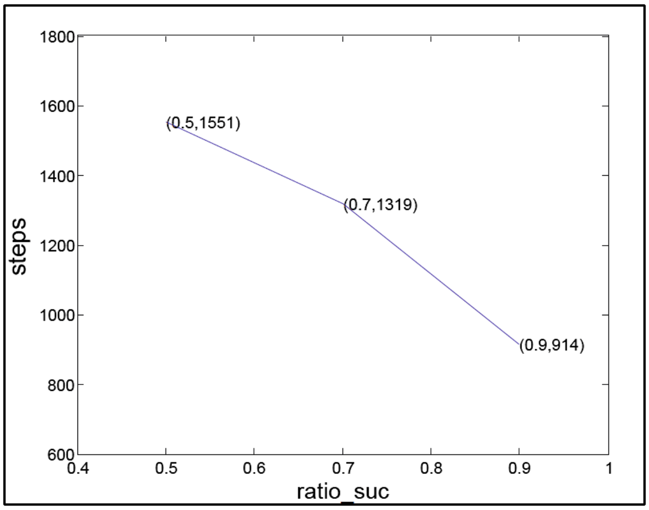

Fig. 3. Average steps of execution

of electing a leader is influenced by the probability of sending messages. This fact can be revealed by the number of execution steps, and the result is illustrated by Figs. 3 and 4.

In Fig. 3, it is obvious that the average steps of execution decrease smoothly with the increasing of probability of successfully sending *RESPONSE* messages. When the probability is 0.5, the cost of average execution steps equals to 1551 steps. And once the probability increases to 0.9, the cost diminishes to 914 steps.

In Fig. 4, we can clearly observe that the variance of execution steps decreases with the rising probability of successfully sending messages. Simultaneously the number of outliers also reduces with this rising. Because of the conditions demonstrated above, we can conclude that the ratio of sending *RESPONSE* messages successfully plays a vital role in the efficiency of leader election in the dynamic system and thus we should guarantee the stability of the communication channels in the system in order to improve the efficiency.

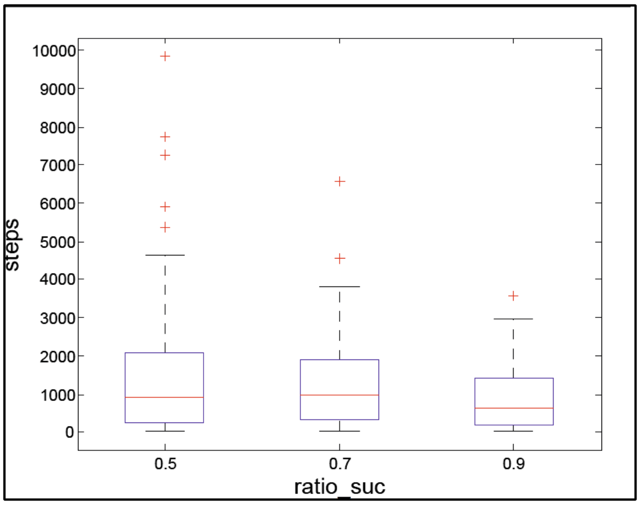

Fig. 4. Distribution of execution steps

5 Related Work

Most existing leader election algorithms are based on static systems [14–16], while in contrast, their counterparts relied on dynamic systems have attracted less attention. However, currently various applications are based on the dynamic system and thus the status of the dynamic system cannot be neglected any longer. The importance of leader election, which is one the fundamental building blocks of distributed computing, should be highlighted in the dynamic system.

Mostefaoui et al. [2] adapted an existing eventual leader election protocol designed for the static model and then translated it to a similar model suiting in the dynamic systems by means of comparing those two models' traits and adapting some specific features. In the paper, it was also theoretically proved that the resulting protocol was correct within the proposed dynamic model.

In [3], we also proposed a hierarchy-based eventual leader election model for dynamic systems. The proposed model was divided into two layers. The main idea in the lower part was to elect cluster-heads of every cluster while the target in the upper one was to elect global leader of the whole system from these existing cluster-heads. The concept of the model was to distinguish the important processes from all processes and then paid more attention to those selected ones in order to diminish the size of concerns and then improve the efficiency.

In addition, Larrea et al. [17] has pointed out the details and the keys to elect an eventual leader. In other words, to achieve the goal of electing an eventual leader, there are some significant conditions which must be satisfied, such as stability and synchrony condition-a leader should be elected under the circumstance of no more processes joining in or leaving out the system. The proposed algorithm relies on entering time stamp comparing.

Meanwhile, there is another line of work with regard to applying formal methods to protocol verification, for example [18, 19]. In [18], Havelund et al. employed the real-time verification tool UPPAAL [20] to perform a formal and automatic verification of a protocol existing in reality in order to demonstrate how model checking had an influence on practical software development. Although it is unrelated with eventual leader election, it demonstrates the feasibility of applying such technique to real world protocols. In [19], Yue et al. used PRISM to present an analysis of a randomized leader election where some quantitative properties had been checked and verified by PRISM. However, the work did not cover the issues for dynamic systems, which is the main focus of the current paper.

6 Conclusions and Future Work

In this paper, we have investigated and analyzed properties of eventual leader election protocols for dynamic systems from a formal perspective. Particularly, we employ PRISM to model an existing protocol, and illustrate the average election round and its scalability via simulation. Moreover, we relax the assumptions made by the original protocol and utilize probability to model the reliability of message channel. We also illustrate relationships between the reliability and the efficiency of election rounds taken by the revised protocol based on probabilistic model checking. In the future, we plan to extend our model and cover more performance measure such as the energy assumption to give a more comprehensive analysis framework.

Acknowledgements. The work was partially funded by the NSF of China under grant No.61202002, No.61379157 and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

1. Yang, Z.W., Wu, W.G., Chen, Y.S., Zhang, J.: Efficient information dissemination in dynamic networks. In: 2013 42nd International Conference on Parallel Processing, pp. 603–610 (2013)
2. Mostefaoui, A., Raynal, M., Travers, C., Patterson, S., Agrawal, D., Abbadi, A.E.: From static distributed systems to dynamic systems. In: Proceedings of the 24th Symposium on Reliable Distributed Systems (SRDS05), IEEE Computer, pp. 109–118 (2005)

3. Li, H., Wu, W., Zhou, Yu.: Hierarchical eventual leader election for dynamic systems. In: Sun, X.-h., Qu, W., et al. (eds.) ICA3PP 2014, Part I. LNCS, vol. 8630, pp. 338–351. Springer, Heidelberg (2014)
4. Chen, S., Billings, S.A.: Neural networks for nonlinear dynamic system modelling and identification. Int. J. Control **56**(2), 319–346 (1992)
5. Merritt, M., Taubenfeld, G.: Computing with infinitely many processes. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 164–178. Springer, Heidelberg (2000)
6. Guerraoui, R., Hurfin, M., Mostéfaoui, A., Oliveira, R., Raynal, M., Schiper, A.: Consensus in asynchronous distributed systems: a concise guided tour. In: Krakowiak, S., Shrivastava, S.K. (eds.) BROADCAST 1999. LNCS, vol. 1752, pp. 33–47. Springer, Heidelberg (2000)
7. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A bayesian approach to model checking biological systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009)
8. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
9. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
10. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
11. Zhou, R.F., Hwang, K.: Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. IEEE Trans. Parallel Distrib. Syst. **18**(4), 460–473 (2007)
12. Vaze, R., Heath, R.W.: Transmission capacity of ad-hoc networks with multiple antennas using transmit stream adaptation and interference cancellation. IEEE Trans. Inf. Theory **58**(2), 780–792 (2012)
13. Aguilera, M.K.: A pleasant stroll through the land of infinitely many creatures. ACM Sigact News **2**, 36–59 (2004)
14. Gupta, I., van Renesse, R., Birman, K.P.: A probabilistically correct leader election protocol for large groups. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 89–103. Springer, Heidelberg (2000)
15. Mostefaoui, A., Raynal, M., Travers, C.: Crash-resilient time-free eventual leadership. In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004, pp.208–217. IEEE (2004)
16. Bordim, J.L., Ito, Y., Nakano, K.: Randomized leader election protocols in noisy radio networks with a single transceiver. In: Guo, M., Yang, L.T., Di Martino, B., Zima, H.P., Dongarra, J., Tang, F. (eds.) ISPA 2006. LNCS, vol. 4330, pp. 246–256. Springer, Heidelberg (2006)
17. Larrea, M., Raynal, M.: Specifying and implementing an eventual leader service for dynamic systems. In: 2011 14th International Conference on Network-Based Information Systems (NBiS), pp. 243–249 (2011)
18. Havelund, K., Skou, A., Larsen, K.G., Lund, K.: Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In: IEEE 18th Real-Time Systems Symposium, 2p (1997)
19. Yue, H., Katoen, J.-P.: Leader election in anonymous radio networks: model checking energy consumption. In: Al-Begain, K., Fiems, D., Knottenbelt, W.J. (eds.) ASMTA 2010. LNCS, vol. 6148, pp. 247–261. Springer, Heidelberg (2010)
20. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)