

Intelligent Distributed System: Solving the Byzantine Generals Problem when Communication is Noisy



HØGSKOLEN I AGDER
Agder University College
Faculty of Engineering and Science

IKT-404
Spring 2007

Author : Trabelsi Walid	Supervisor : Ole Christoffer Granmo
Version : 1.0 Status : FINAL	Pages: 31 (including this page) Modified date: 2007-05-12
Keywords: Intelligent distributed System, Fault Tolerance, Byzantine General Problem, Learning Automata, Linear Reward inaction, Simulated environment, Java. Abstract : <p>Several forms of networks in distributed system (long-haul networks, modems, cell phones), and of medium to long term storage (DRAM, hard drives, backup tapes) are susceptible to data corruption by noise in the environment. Such corruption often manifests as random, independent bit errors, which can be protected against by solving Byzantine General Problem. The Byzantine Generals Problem requires processes to reach agreement upon a value even though some of them may fail. It is weakened by allowing them to agree upon an "incorrect" value if a failure occurs.</p> <p>Our claim here is that to implement and test a novel and previously unpublished scheme that is able to solve the Byzantine Generals Problem when communication is noisy. Then, we suggest handling malfunctioning components in such reliable computer system that give conflicting information to different parts of the system, basing on the L_{RI} - Automaton approach. Our solution is tested through a Java implementation and an evaluation by developing an algorithm for a simulated environment.</p>	

Table of Contents

1.	Introduction.....	3
2.	Problem Description	4
3.	Project's Goal.....	5
4.	Background.....	5
4.1	Intelligent Distributed Systems.....	6
4.2	Fault tolerance.....	6
4.3	Byzantine general problems.....	7
4.3.1	OM and SM, Two Solutions to the Byzantine Generals Problem	8
4.3.1.1.	Oral Message Model.....	8
4.3.1.2	Signed Message Model.....	9
4.4	A Game of Learning Automata.....	9
4.4.1	Linear Reward-Inaction Scheme (L_{RI}).....	11
5.	Solution.....	12
6.	Requirements	13
7.	Design Specification	13
8.	Implementation	18
9.	Evaluation and testing.....	22
10.	Discussion and Further work	29
11.	Conclusion	30
	Appendix.....	31
A1	Glossary & abbreviations.....	31
A2	References.....	31

1. Introduction

Malicious attacks and software errors are increasingly common. The growing reliance of industry and government on online information services makes malicious attacks more attractive and makes the consequences of successful attacks more serious. In addition, the number of software errors is increasing due to the growth in size and complexity of software leading to noise environment. Since malicious attacks and software errors in such noise environment can cause faulty nodes to exhibit Byzantine (i.e., arbitrary) behavior, Byzantine-fault-tolerant algorithms are increasingly important.

This can be seen as a Byzantine General Problem, which involves obtaining agreement among a collection of processes, some of which may be faulty. Then by given a collection of processes which communicate by sending messages to one another, the aim of this project is to find an algorithm by which Processes can reach an agreement in presence of faulty process in such noise environment.

[1,2] presented the Learning reward Inaction (L_{RI} – Automaton) as one of theoretical stochastic learning Automaton solution whenever an unfavorable response results from the environment. It demonstrated that Learning represents one of the most important psychological processes and it is essential in the behavior of self-adjusting organisms.

In this project, we present and evaluate a novel intelligent distributed system as a solution to Byzantine General Problem in noise environment using the Learning reward Inaction Automaton (L_{RI} – Automaton). The L_{RI} – Automaton seems particularly promising for such problem where information given by faulty process in Byzantine General Problem can be considered as an unfavorable response results from the noise environment.

Our solution is tested through a Java implementation and an evaluation by developing an algorithm for a simulated environment.

The rest of this report is organized as follows: In section 2 we describe the problem of our project and in section 3 we present the goal of the project. We present a review of relevant literature and some related work in section 4. After that it comes in section 5, the description of our novel scheme as solution for Byzantine General Problem. We detail the requirements for the testing and evaluation of this problem in section 6. The proposed design of a variant of our novel as a solution to the problem presented in section 2 is then presented in section 7. In section 8, the implementation process is described based on our proposed design in section 7. The development of the algorithm is then tested and evaluated in section 9. The discussion is found in section 10. We conclude the paper with Section 11 and offer prospects for further work.

2. Problem Description

Given a noise environment consisting of large numbers of failure devices, many challenges exist in designing reliable systems, including programming large networks of energy constrained, error devices, re-programming devices after deployment and exploiting redundancy to improve system lifetime and fault-tolerance.

Applications can utilize hundreds of computational nodes, sensors, actuators and communication links. Numerous nodes imply that most applications will utilize multiple processing units, and these applications must be partitioned to make effective use of available hardware resources. Although applications can run on a single node, nodes may still be coordinated in a larger scope.

The lifetime of a system is defined as the amount of time that the system remains functioning. We believe that lifetime of a reliable system will be increasingly important to be durable as long as possible. However in such noise environment, failure can happen anytime and the number of software errors is increasing due to the growth in size and complexity of software.

That is why reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked namely, sending conflicting information to different parts of the system.

This can be seen as a Byzantine General Problem in distributed system, as we imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that all loyal generals decide upon the same plan of action. In other word, the loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee this condition regardless of what the traitors do. The loyal generals should not only reach agreement, but should agree upon a reasonable plan. We therefore also want to insure that a small number of traitors cannot cause the loyal generals to adopt a bad plan.

Byzantine General Problem can be assumed in real world as malfunctioning components and errors software in reliable computer system, where general can be considered as several process and traitor is seeing as a faulty process leading to fault tolerant problem.

In this project, we suggest to implement and test a novel scheme that is able to solve the Byzantine Generals problem when communication is noisy. We were to test if Linear Reward-Inaction Automaton (L_{RI} -Automaton) could be used for such kind of problem. It is highly desirable that reliable computer system reach an agreement in presence of any faulty process as quick as possible.

3. Project's Goal

The main objective of our project is to implement a novel scheme and to develop an algorithm solving Byzantine General Problem in noise environment. With such intelligent distributed system, we should create a simulated environment to verify our proposed solution solving Byzantine failure. This environment is designed to represent the behavior of the real world of intelligent distributed system, both with regards of the behavior of processes and occurrence of fault tolerant computing.

For a system to exhibit a Byzantine failure there must be a system-level requirement for consensus. If there is no consensus requirement, a Byzantine fault will not result in a Byzantine failure.

The heart of our scheme suggested especially that Byzantine Generals have been observed attacking systems. They are not mythical. On the contrary, Byzantine faults in safety-critical systems are real and can occur with failure rates far more frequently than the 10^{-6} , 10^{-9} , or even 10^{-12} faults per operational hour requirement. We believe that in real world, we have to conclude that Byzantine failures are a significant problem. As alluded to earlier, being aware of Byzantine behavior enabled these observations. Had this not been the case, observed misbehavior may have been dismissed as an unknown or an extremely rare "anomaly". It is hoped that discussion of the Byzantine observations will show that the problem is real and it is important to take care about the characteristics of these faults. That's why we believe that our aim to solve Byzantine General Problem is desirable important and interesting in real safety-critical systems.

In the following, we address some related background to Byzantine General Problem and suggested methods of solving the problem.

4. Background

To accomplish the task at hand, we have been reading about previous work done in the field. In this section we look at different technologies relevant to intelligent distributed systems, Byzantine general problems therefore their solutions and some versions of pursuit of Learning Automata.

4.1 Intelligent Distributed Systems

According to [3], most information and processing systems are based on centralized technologies and design principles in which the information and knowledge are centralized at strategic sites, with access, command, and control organized in client-server architectures. Centralized systems have many disadvantages that make them unsuitable for large-scale integration, including high reliance on centralized communication, high complexity, lack of scalability, and high cost of integration.

The use of distributed intelligence system technologies avoids these weaknesses. By distributing implementation details of the logistical and integration requirements, it is possible to achieve greatly improved reliability, scalability, and security.

Distributed intelligence systems are based on the use of cooperative agents, organized in hardware or software components, that independently handle specialized tasks and cooperate to achieve system-level goals and achieve a high degree of flexibility. By distributing the logistic and strategic requirements of a system, it is possible to achieve greatly improved robustness, reliability, scalability, and security. Key to achieving these benefits is the use of these system technologies that establish a peer-to-peer environment to enable coordination, collaboration, and cooperation within the network. Such systems require both hardware and software components.

4.2 Fault tolerance

Fault management is the monitoring of error indications in a computer system in order to log the occurrences and send alerts to system administrators and field service. Fault management software keeps track of hardware faults such as memory parity errors and software crashes.

As in [4,6], Fault tolerance is one of the important types of fault control. It is the ability to continue non-stop when a hardware failure occurs. A fault-tolerant system is designed from the ground up for reliability by building multiples of all critical components, such as CPUs, memories, disks and power supplies into the same computer. In the event one component fails, another takes over without skipping a beat.

Fault tolerance is usually achieved by duplicating key components of the system. Many systems are designed to recover from a failure by detecting the failed component and switching to another computer system. These systems, although sometimes called fault tolerant, are more widely known as "high availability" systems, requiring that the software resubmits the job when the second system is available.

True fault tolerant systems with redundant hardware are the most costly because the additional components add to the overall system cost. However, fault tolerant systems provide the same processing capacity after a failure as before, whereas high availability systems often provide reduced capacity.

4.3 Byzantine general problems

To more easily identify the problem, concise practical definitions of Byzantine fault and Byzantine failure are presented here:

Byzantine fault: a fault presenting different symptoms to different observers.

Byzantine failure: the loss of a system service due to a Byzantine fault in systems that require consensus.

For a system to exhibit a Byzantine failure there must be a system-level requirement for consensus. If there is no consensus requirement, a Byzantine fault will not result in a Byzantine failure. Many distributed systems have an implied system-level consensus requirement such as a mutual clock synchronization service. Failure of this service will bring the complete system down. Asynchronous approaches do not remove these problems. It is yet another myth that asynchronous systems are immune to Byzantine problems. Any coordinated system actions will still require consensus agreement. In the real-time control systems that make up the majority of systems that have high dependability requirements, it is nearly impossible to create a redundant system in which there is absolutely no coordination required among the system's redundancies and/or outputs. In addition, adequately validating asynchronous systems to the typically required level of assurance is usually much more difficult than for synchronous systems.

According to Lamport et's paper [5], The Byzantine Generals Problem involves obtaining agreement among a collection of processes, some of which may be faulty. It can be stated precisely as follows.

- There are n armies camped outside of an enemy city.
- Each army is commanded by a general.
- Some generals are loyal, while some generals are traitors.

The Byzantine Generals Problem is solved by an algorithm when it achieves the following conditions:

1. All loyal generals will agree to either attack or retreat.
2. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Because of the difficulty in formalizing the second of the above conditions, the Byzantine Generals Problem is reformulated by Lamport into the following modified form.

- There are n armies camped outside of an enemy city.
- One army is led by a general while the rest are led by lieutenants.
- Some number of the lieutenants and the general are loyal, while others are traitors.

This modified problem statement is solved when the following conditions hold.

1. All loyal lieutenants decide upon the same course of action.
2. If the general is loyal, all loyal lieutenants obey the order he sends.

4.3.1 OM and SM, Two Solutions to the Byzantine Generals Problem

4.3.1.1. Oral Message Model

The first solution considers the Byzantine Generals Problem without utilizing digital signatures. In the context of the problem, it considers such messages “oral messages,” and they are governed by the following restrictions.

1. Every sent message is correctly delivered.
2. The receiver of a message knows who sent it.
3. The absence of a message can be detected.

These assumptions assure a few conditions. First, traitors cannot interfere with communication between two other generals, by the first two conditions. The third condition is used to ensure that a traitor cannot prevent a decision by not sending messages. We use this model of message passing in the first Lamport et. al. Byzantine Generals solution, $OM(m)$.

The $OM(m)$ algorithm solves the Byzantine Generals Problem for m traitors where there are at least $3m+1$ total generals. $OM(m)$ is defined inductively, and proceeds from the base case $OM(0)$.

Algorithm $OM(0)$

1. The general sends his value to every lieutenant.
2. Each lieutenant uses the value received from his general, or uses a preset default value if he receives no message (assumption 3 of our message passing model allows this).

Algorithm OM(m)

1. The general sends his value to every lieutenant.
2. For each lieutenant i , let v_i be the value he receives from the general, or else a preset default if he receives no message. Lieutenant i performs the algorithm OM(m-1) among the other $n - 2$ lieutenants using the value v_i as its value.
3. Let v_j be the value lieutenant i received from lieutenant j in the above execution of the OM(m-1) algorithm, or else a preset default if no message was received. Lieutenant i uses the value $\text{majority}(v_1, \dots, v_{n-1})$.

4.3.1.2 Signed Message Model

The signed message model adds the following constraints to the message transmission model.

1. A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
2. Anyone can verify the authenticity of a general's signature.

No assumption is made about the validity of the signature of a traitorous general. This specifically permits the possibility of collusion among traitors.

Using the signed message model, the SM(m) algorithm solves the Byzantine Generals Problem for $2m + 1$ processors, not total generals, in the presence of m traitors.

1. The general signs and sends his value to all lieutenants.
2. upon receipt of a message with value v , lieutenant i :
 - (a) Adds the value v to a list V_i .
 - (b) If the message has fewer than $m + 1$ signatures (including that of the general) sign the message and send a copy to all lieutenants who have not signed the message.
 - (c) After no more messages will be received, lieutenant i takes his value to be the majority of the values in V_i .

4.4 A Game of Learning Automata

A "learning" automaton is an automaton that interacts with a random environment, having as goal to improve its behavior. It is connected to the environment in a feedback loop, such that the input of the automaton is the output of the environment and the output of the automaton is the input for the environment, as shown in Figure.

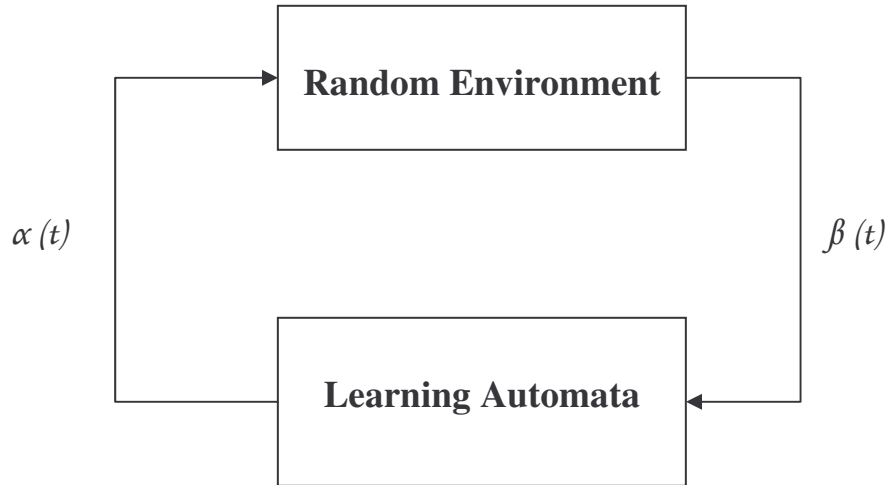


Figure 1: Learning Automaton

The goal of an automaton is to determine the optimal action from a set of possible actions. The automaton performs these actions in a random environment that generates a response for each action.

Learning automata can be classified with respect to their transition states as being either deterministic or stochastic. For a deterministic automaton, given an initial state and input, the next state and action are uniquely specified. For a stochastic automaton, given an initial state and input sequence, there is no certainty regarding the next states and actions of the automaton. These automata can be classified with respect to their transition functions in two categories: Fixed Structure Stochastic Automata (FSSA) and Variable Structure Stochastic Automata (VSSA).

In the case of FSSA, the state transition and output functions are independent of time, and are thus considered to be of a fixed structure". Tsetlin, Krylov, and Krinsky presented notable examples of these automata type.

The VSSA are designed with more flexibility, allowing the state transition and output function to Vary in time. The VSSA can be classified according to the updating probability functional form. If $\mathcal{P}(t+1)$ is a linear function of $\mathcal{P}(t)$, the automaton is said to be linear, otherwise it is considered nonlinear. Occasionally, two or more automata are combined to form a hybrid automaton. Independent of the form of the updating scheme, a VSSA follows some basic learning principles. If an action α_i has been rewarded, the automaton increases the probability for this action, decreasing the probability for all other actions. If an action α_i has been penalized, the automaton decreases the probability for this action, increasing the probability for all other actions. Depending on the learning

principle of its VSSA, different combinations of updating schemes can be enumerated as:

RP: Reward-Penalty (L_{RP}): the probabilities are updated when the automaton is rewarded and penalized.

RI: Reward-Inaction (L_{RI}): the probabilities are updated when the automaton is rewarded and are left unchanged when the automaton is penalized.

IP: Inaction-Penalty (L_{IP}): the probabilities are updated when the automaton is penalized and are left unchanged when the automaton is rewarded.

4.4.1 Linear Reward-Inaction Scheme (L_{RI})

We are interested to describe only LRI-Automaton scheme as we will use in our solution. The basic idea of the reward-inaction scheme (L_{RI}) is to keep the probabilities unchanged whenever the environment replies with an unfavorable response. Following a favorable response, however, the probability of the action is increased as in (L_{RP}) Scheme. The updating equations for this scheme can be presented as follow:

$$\begin{array}{lll}
 \mathcal{P}(t+1) = \mathcal{P}(t) + \lambda(1 - \mathcal{P}(t)) & \alpha(t) = \alpha_1 & \beta(t) = 0 \\
 \mathcal{P}(t+1) = \mathcal{P}(t) & \alpha(t) = \alpha_1 & \beta(t) = 1 \\
 \mathcal{P}(t+1) = (1 - \lambda)\mathcal{P}(t) & \alpha(t) = \alpha_2 & \beta(t) = 0 \\
 \mathcal{P}(t+1) = \mathcal{P}(t) & \alpha(t) = \alpha_2 & \beta(t) = 1
 \end{array}$$

These equations indicate that the probability $\mathcal{P}(t)$ is increased if action α_1 is performed and results in a favorable response, is unchanged if an unfavorable response results when α_1 or α_2 is performed and is decreased only when the other action α_2 is performed and results in a favorable response.

These equations indicate that this scheme has two absorbing States: $[0, 1]^T$ and $[0, 1]^T$. For example, if $\mathcal{P}(t)$ becomes unity and action α_1 is rewarded, the probability becomes

$$\mathcal{P}(t+1) = 1 + \lambda(1 - 1) = 1$$

If the automaton is penalized at this time, then the probability remains unchanged since the automaton does not react to penalties, which implies that the state $[0, 1]^T$ is an absorbing state. In an analogous manner, it can be shown that $[0, 1]^T$ is an absorbing state. This makes the L_{RI} scheme inappropriate for non-stationary environments.

5. Solution

In order to solve the Byzantine General Problem we have used linear reward-inaction Automaton (L_{RI} -Automaton) that is described in the section 4.4.1.

[1,2] presents linear reward-inaction Automaton (LRI-Automaton) as one of theoretical stochastic learning Automaton solution whenever an unfavorable response results from the environment. It demonstrated that Learning represents one of the most important psychological processes and it is essential in the behavior of self-adjusting organisms.

The objective of this project is to present and evaluate a novel scheme as solution to Byzantine General Problem using Linear Reward-Inaction Automaton (LRI-Automaton). The LRI-Automaton seems particularly promising for Byzantine failure because it learns incrementally/on-line, has low computational complexity and has an excellent behavior whenever an unfavorable response results from the environment. We were to test if Linear Reward-Inaction Automaton (LRI-Automaton) could be used for such kind of problem. It is highly desirable that reliable computer system reach an agreement in presence of any faulty process as quick as possible.

Applications such as adaptive control systems and reliable computer system in distributed environment, needed to incorporate learning characteristics in their functionality in order to mode systems with a substantial amount of uncertainty. Many of these systems are required to choose the correct action (decisions) without a priori knowledge of the consequences of performing these actions. To perform well under these conditions of uncertainty, the systems needed to acquire some knowledge about the consequences of performing the various actions. This can be seen as the goal of Learning Automata when the functionality is to determine the optimal action out of a set of allowable actions.

The number of software errors into reliable computing systems is increasing due the growth in size and complexity of software leading to noise environment. Since malicious attacks and software attacks in such noise environment can cause faulty nodes or faulty processes to exhibit Byzantine-fault-tolerant problem. Our solution using the LRI-Automaton seems particularly promising for this kind of failure because it learns incrementally/on-line and has an excellent behavior whenever an unfavorable response results from the environment.

6. Requirements

The solution under development needs some requirements to work satisfactory, we have predefined a few requirements for the Automaton's functionality.

- We need to create a simulated environment to test and evaluate our solution basing on the L_{RI} - Automaton to solve Byzantine General Problem. This environment is designed to represent the behavior of the real world as best as possible, handling malfunctioning components and errors software in reliable computer system knowing by Byzantine failure.
- In order to evaluate our algorithm, we need to realize a several tests to fix some parameters to be used. In this way, we choose to merge the state of the environment when communication is noisy. We should vary the number of faulty processes running in parallel with a different collection of processes. We calculate also the time needed to solve Byzantine fault-tolerant problem in each test phase.
- The Automaton should be able to organize the messages given by right processes and faulty processes basing on its actions: reaching the same decision of all processes in final step.
- In order to solve Byzantine failure when communication is noisy, our solution should be able to reach the same decision and resulting to an agreement between all processes.
- If we finish the development of the algorithm as soon as possible, we develop also a graphical user interface GUI of the automation in action, to observe the Automaton's decisions. This GUI should be able to give us an overview of the actions taken around the Automaton.

7. Design Specification

Our novel scheme is based on linear reward-inaction Automaton (L_{RI} -Automaton) described in section 4.4 in order to solve Byzantine General Problem. Before we describe our assumption of the choice of actions, we present the mapping of practical Byzantine fault tolerance between the developed algorithm for Byzantine General Problem and a real world in reliable computing system in the table 1.

Simulated Environment	Reliable computing system
Generals	Processes
Loyalist/Traitor	Right/faulty process
Messages	bits
Attack/Retreat	1/0

Table 1: Mapping between a simulated environment and a real reliable computing system.

We start by describing the general rules of Byzantine General Algorithm. First of all each division of Byzantine army are directed its own general. Generals, some of which are traitors, communicate each other by messengers. Traitors try to confuse the loyalists by given misleading answers. Into this presence of traitors, all generals decide upon the same plan of action reaching to an agreement. The aim is to reach a consensus when small number of traitors cannot cause the loyal generals to adopt a bad plan. It exist two types of messages (Attack / Retreat) to handle the behaviour of both of Loyalists and traitors. Figure 2 presents the general behavior of Byzantine General Problem

Unlike the basic rules and solutions given by Lamport in [5] and described at section 4.3, our scheme involves that:

- **Traitor:** selects Attack/Retreat always randomly, each with probability 0.5. This is to confuse the loyalists.
- **Loyalist:** Use LRI-Automaton to select between Attack/Retreat.

The Automaton runs in one specific loop until the Automaton reach an agreement between all generals, with an iteration defined as one test and the following action taken based upon this test. A general iteration is outlined below, in figure 3.

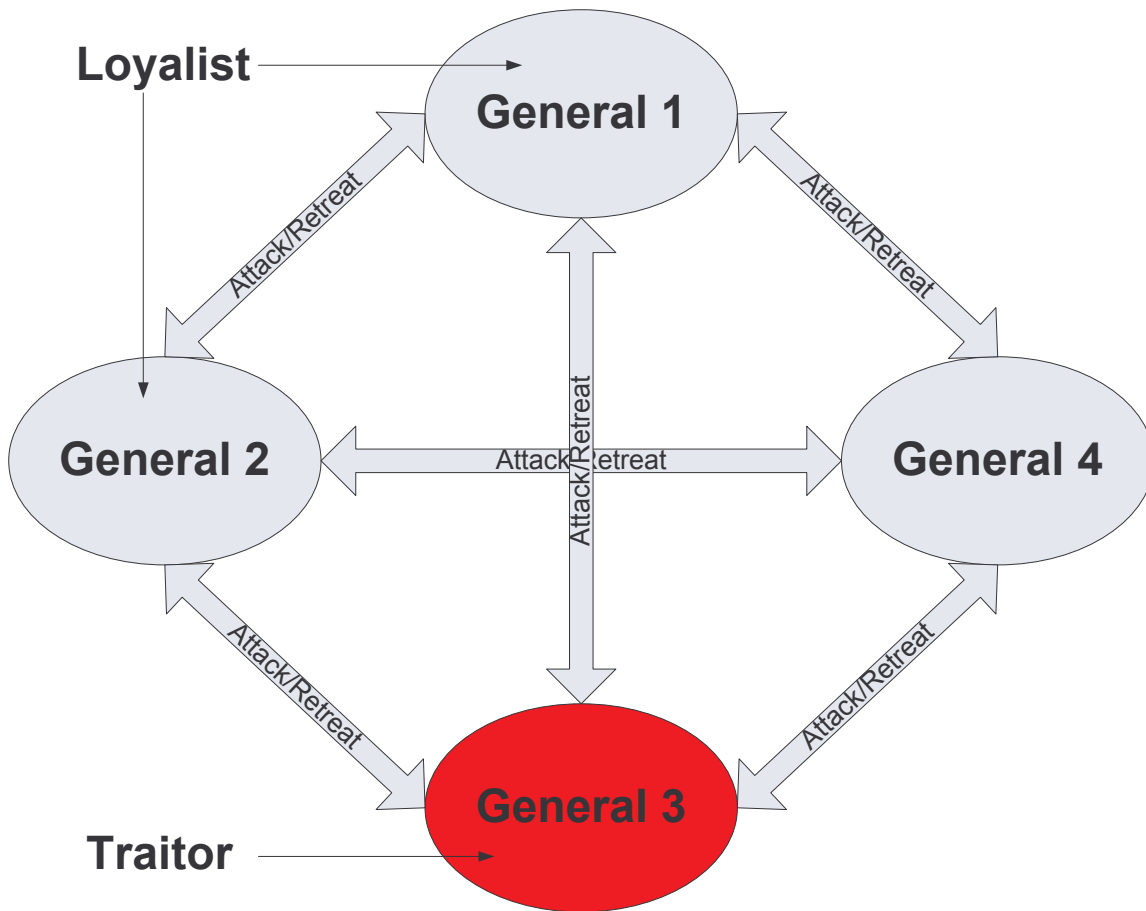


Figure 2: Behavior of Byzantine General Problem

It is noticeable that in our solution, the Automaton runs into specific loop depending on probability ' \mathcal{P} ' calling the probability of choosing "Attack" to distinguish between selecting Attack and Retreat. It depends also on reward probability R defining by the number of generals that agree and choose the same decision as the others over the total number of generals. The reward probability ' \mathcal{R} ' is used to update the probability of choosing "Attack" according to the LRI-Automaton. Still, we base our actions on the two probabilities ' \mathcal{P} ' and ' \mathcal{R} '. The general assumptions of the choice of Automaton's actions are described below in figure 3.

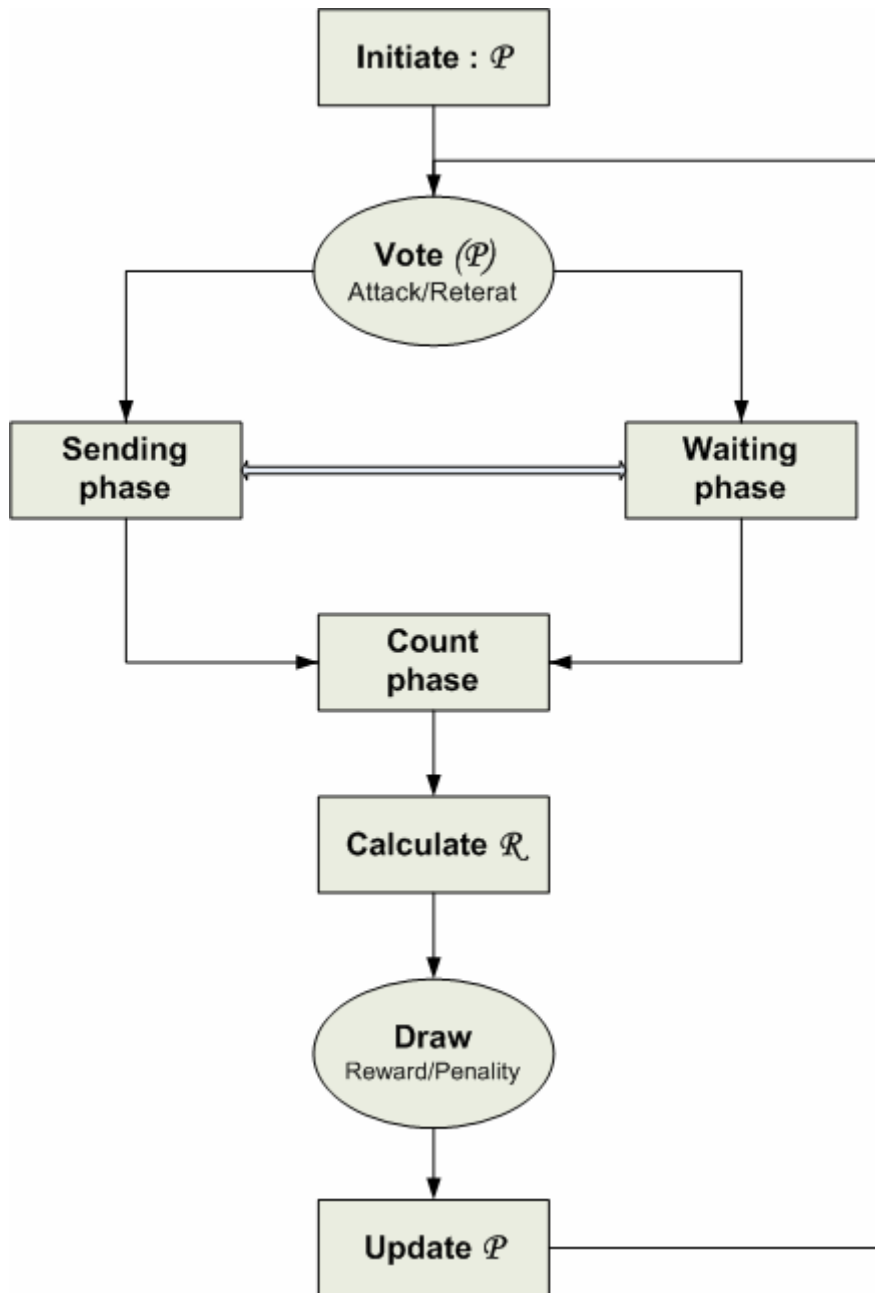


Figure 3: The L_{RI} -Automaton process

According to Automaton process describing in figure 3, all the generals follow this same procedure concurrently:

1. Initiate the probability of choosing "Attack" to 0.5. $P = 0.5$.
2. Decide to vote for either 'Attack' or 'Retreat' randomly using the action probability ' \mathcal{P} '.
3. Assume that corresponding general have decided to go for ' \mathcal{X} ' (where ' \mathcal{X} ' could be either 'Attack' or 'Retreat'). Also assume that you know the total number of generals ' \mathcal{N} '.
4. Decision ' \mathcal{X} ' is sending to the other generals.
5. Waiting till the general received the decisions of the other generals.
6. Count the number of other generals that also choose to do ' \mathcal{X} '.
7. Calculate the reward probability ' \mathcal{R} ' as follows:

$$\mathcal{R} = \text{"The number of other generals that also chose } \mathcal{X} \text{"} / \mathcal{N}$$

8. Draw a reward or penalty randomly based on ' \mathcal{R} ', and update ' \mathcal{P} ' accordingly to standard L_{RI}-Automaton as follow.

$\mathcal{P}(t+1) = \mathcal{P}(t) + \lambda(1 - \mathcal{P}(t))$	$\alpha(t) = \text{Attack}$	$\beta(t) = \text{reward}$
$\mathcal{P}(t+1) = \mathcal{P}(t)$	$\alpha(t) = \text{Attack}$	$\beta(t) = \text{inaction}$
$\mathcal{P}(t+1) = (1 - \lambda)\mathcal{P}(t)$	$\alpha(t) = \text{Retreat}$	$\beta(t) = \text{reward}$
$\mathcal{P}(t+1) = \mathcal{P}(t)$	$\alpha(t) = \text{Retreat}$	$\beta(t) = \text{inaction}$

These equations indicates that the probability $\mathcal{P}(t)$ of choosing "Attack" is increased if action $\alpha(t) = \text{Attack}$ is performed and results in a favorable response $\beta(t) = \text{reward}$, is unchanged if an unfavorable response results when $\alpha(t) = \text{Attack}$ or $\alpha(t) = \text{Retreat}$ is performed and is decreased only when the other action $\alpha(t) = \text{Retreat}$ is performed and results in a favorable response.

9. Go to 2.

Note we are working with an unknown environment. This means that the algorithm can only know the number of iterations by actually performing a time passing to solve problem and reaching an agreement. In other word, in order to represent the environment presented above, each process in our environment is seen as a general with a random stochastic variable. This means that the parameters used in the algorithm are represented as a random variable that can

only be retrieved after the selecting both of probability of choosing Attack ' \mathcal{P} ' and reward probability ' \mathcal{R} '.

8. Implementation

First of all, in order to make our solution into intelligent distributed system, we need an application server, we use Jakarta Tomcat 4.1.x version. We use Java as programming language and web services (XML RPC) as technique of distributed system. Assuming that we have a web server up and running on the localhost at port 8080 after installing tomcat 4.1. We use Axis 1.4 and we copy then the folder called axis into C:\Program Files\Apache Group\Tomcat 4.1\webapps. Axis needs to be able to find an XML parser. To add an XML parser, we use Xerces 2.5 jars and we copy the Xerces's jar files (xercesImpl.jar, xercesSamples.jar, xml-apis.jar and xmlParserAPIs.jar) within C:\Program Files\Apache Group\Tomcat 4.1\webapps\axis\WEB-INF\lib.

After predefining the work environment in distributes system, we present the parameters and the methods used as an implementation of our design described in section 7. The system starts by reading the input stream to be used into the Automaton. This can be done by having some predefined parameters such that the collection of processes as total number of generals, the number of faulty processes as Traitors in Byzantine General Problem. Many others parameters will be used such that the probability of choosing "Attack" ' \mathcal{P} ', the small value close to zero λ used in update phase of ' \mathcal{P} ' and the reward probability ' \mathcal{R} ' into the L_{RI} -Automaton loop. After predefining the parameters, we predefine the necessary methods that can be used as an implementation of our design.

To implement this as web service we divide the work into two parts: the client side and the server side. In the server side we implement different classes of processes (generals) containing the different methods that user can find such us general_sender, reward_probability, update_probability... These classes depend on the parameters defining above.

In the client side we start by acquiring the arguments given by the user and we verify also that the called method exist on the server side. After making the test and being sure that the user is requesting the service in the right way, we define the necessary data. Then we open a connection from the client side in order to access the web service by giving the URL of the jws file that contains the required service. Then the client or the user can enter in the graphical user interface (figure 5) the parameters (choosing the number of faulty processes "Traitor") to be sent to the server in order to get the result. Before calling invoke() method you need to call addParameter for each parameter and setReturnType for the return and we need to precise for these two methods the xml type of the parameters and also if it is an input or an output. After calling invoke the

algorithm is running giving the final result and we can show it on the screen into the GUI. General_sender method is an example of using this procedure that is showing in the list below:

```

.....
try{
    String endpoint = "http://localhost:8080/axis/Gen8"+id+".jws";

    Service service = new Service();
    Call call = (Call) service.createCall();
    call.setTargetEndpointAddress( new java.net.URL(endpoint) );
    call.setOperationName( "GeneralSender" );
    call.addParameter( "message", XMLType.XSD_STRING,
    ParameterMode.IN );
    String m =new String (message);
    call.setReturnType( XMLType.XSD_STRING );
    String Res = (String) call.invoke( new Object [] {m});

        return Res;}
        catch(Exception e){
            return null;}
    }
    .....

```

Figure 4: Pseudo code of implementing a web service technique

After that, we are interesting on the core of algorithm solving the Byzantine General Problem, we start by making Traitor choose or selects randomly between Attack and Retreat.

```

{
.....
Random generator = new Random();
int c = generator.nextInt(2);
String choose []={"Attack","Retreat"};
    return choose [c];
.....
}

```

Figure 5: Pseudo code of Traitor's choose

The next step is to define the vote method that it depends on probability of choosing "Attack". We calculate also the reward probability \mathcal{R} to expand the reward_prob method. The reward probability is calculated by counting the number of generals that agree with others generals dividing by the total number of generals. Our assumption to calculate the reward probability with collection of four generals is outlined pseudo code in figure 6.

```

.....
{
String Order [] = {"", "", ""};

int count=0;
for (int i=0;i<3;i++){
    if (Order[i].equals(my_decision))
        count=count+1;
    }
return ((double)count)/4;
}
.....

```

Figure 6: Pseudo code of computing a reward probability

Our solution is able to handle and solve Byzantine General Problem when communication is noisy. To make in evidence this approach, we assume that environment can disturb the arriving of right messages. If one general sends an “Attack” message to the others generals, noise environment can give a misleading answer, “Retreat” or “Attack” message choosing randomly, with probability equals to 10%. This is described in figure 7

```

/.....
Random generator = new Random();
double noise=0.1;
double n=generator.nextDouble();

    if (n>=noise)
        Order[i]= order;
    else{
        int c= generator.nextInt(2);
        String choose []={"Attack", "Retreat"};
        Order[i]= choose[c];
    }
/.....

```

Figure 7: Computing noise communication

```

/.....
double d = generator.nextDouble();
if (d<=r){
    if (Voting.equals("Attack"))
        p = p*(1-a)+a;
    else
        p = p*(1-a);
    }
return p ;
/

```

After that, it comes the update phase of probability of choosing “Attack”. Sure it depends on reward probability to decide. This by drawing a reward or penalty randomly based on \mathcal{R} , and update \mathcal{P} accordingly to standard L_{RI} -Automaton as showing in figure 8.

Figure 8: Update the probability of choosing “Attack”.

Now it remains only to implement the LRI-Automaton loop described in design section. Note that running the specific loop stops and our solution converges when all generals reach to the same decision after several iterations.

We also implemented a graphical user interface for the simulated environment to show the process of the Automaton, and to easily get the information attached to each iteration and the general process of the Automaton. Figure 9 shows a screenshot of this user interface.

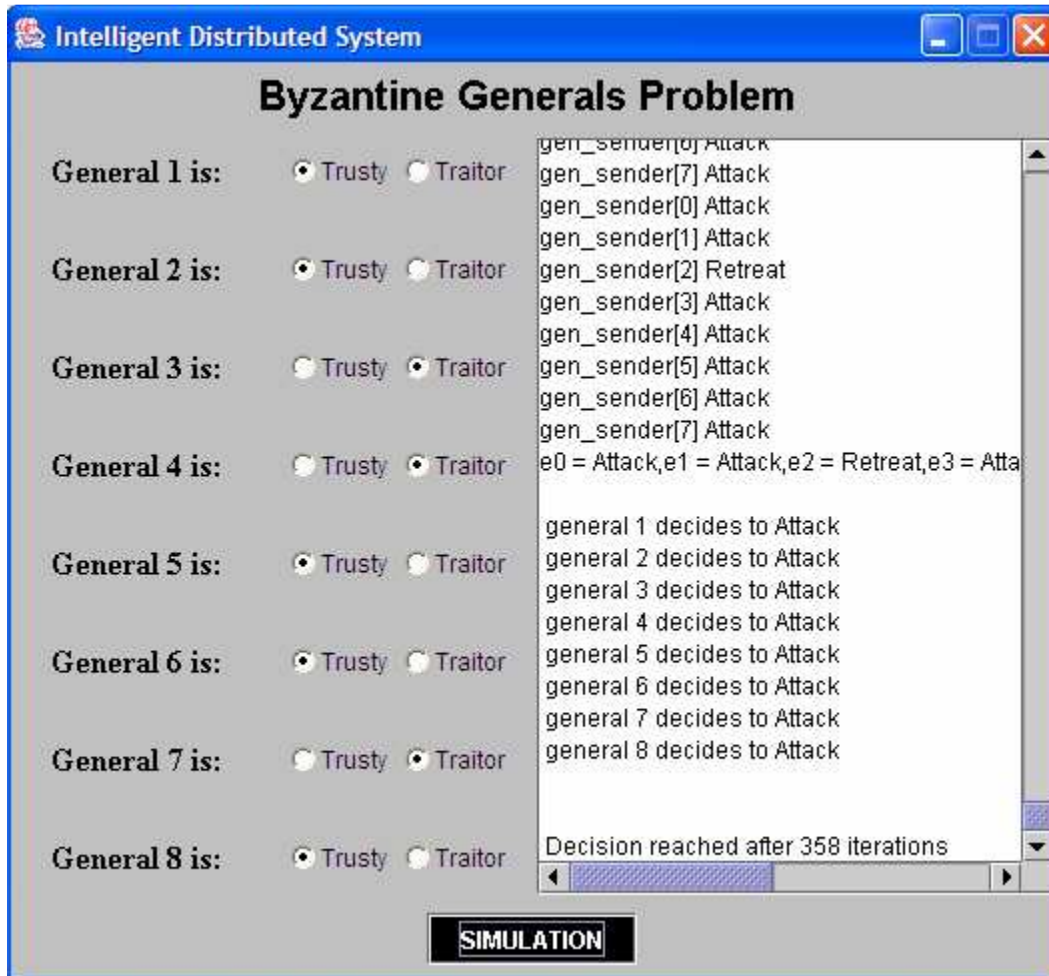


Figure 9: Graphical User Interface for the simulated environment

9. Evaluation and testing

In this section, we evaluate our solution and the presented algorithm. In all the test results, the iterations where solution converges and it can be solved to reach the same decision of all process are the mean value of 10 test runs per test. Then all results presented are the outcome of averaging 10 runs.

Note also that in all the test results, the amount per cent of noise is taken 10 %, in other word the probability of occurrence of noise is equal to 0.1.

Phase 1: Solving Byzantine General Problem with one faulty process and Noise = 10 %.

In the first phase we try to solve Byzantine General Problem in presence of only one faulty process when communication is noisy with probability equals to 10 %. We start by testing collection of two processes; one of them is a faulty process. As result, the algorithm can not reach agreement. That's why we try to increase the total number of processes to three. Now our solution converges and the three processes can reach an agreement with only one faulty process.

We present in table 2 the result of the first phase. Note that we calculate the number of iterations as time spending by the Automaton to get the same decision for all process.

Number of process	2	3
Agreement	No	Yes
Iterations to agreement	---	870

Table 2: Agreement with one faulty process and Noise = 10 %

Phase 2: Solving Byzantine General Problem with two faulty processes and Noise = 10 %.

The next test phase included an extra faulty process. We try now to find a solution for Byzantine General Problem when it exist two faulty processes. Experimental Testing shows collection of three and four processes can not be able to solve problem of 2 two faulty processes. Our solution converges using a collection of five processes. All processes can reach the same decision in presence of two faulty processes when communication in noisy with probability of 10%. Table 3 presents the result of second phase test.

Number of process	3	4	5
Agreement	No	No	Yes
Iterations to agreement	---	---	698

Table 3: Agreement with two faulty processes and Noise = 10 %

Phase 3: Solving Byzantine General Problem with three faulty processes and Noise = 10 %.

The aim of third test phase was to find how many processes are needed to cope with two faulty processes. Starting by a collection of five processes till seven processes, experimental testing confirms that this number of processes could not solve Byzantine General Problem with two faulty processes. Our algorithm converges when we use a collection of eight processes. In such case, all processes can reach an agreement in presence of three faulty processes when communication in noisy with probability of 10%. Table 4 presents the result of third phase test.

Number of process	5	6	7	8
Agreement	No	No	No	Yes
Iterations to agreement	---	---	---	530

Table 4: Agreement with three faulty processes and Noise = 10 %

Phase 4: Solving Byzantine General Problem with four faulty processes and Noise = 10 %.

This is the final phase of testing to be able to evaluate the different results. We try to solve Byzantine General Problem in presence of four faulty processes. Basing on experimental testing, our solution converges using a collection of 10 processes. The ten processes can reach the same decision in presence of four faulty processes when communication in noisy with probability of 10%. Table 5 presents the result of third phase test.

Number of process	8	9	10
Agreement	No	No	Yes
Iterations to agreement	---	---	601

Table 5: Agreement with three faulty processes and Noise = 10 %

We believe that our solution depends on the total number of processes, especially the number of iterations as time spending to converge the algorithm. For example, with a collection of eight processes, time spending to reach an agreement is totally different of time spending with a collection of four processes, in the case of presence one faulty process. For this reason, we calculated the number of iterations for all cases during testing phase.

The results of the testing process are presented and summarized below in table 6. We can conclude that solving a certain number of faulty processes depends on a total number of processes to reach an agreement between them. Time spending to solve Byzantine General Problem in such noise environment depends also of total number of processes.

According the experimental result in table 6, we could conclude a general guideline for how many generals \mathcal{N} are needed to cope with $\mathcal{N}_{\mathcal{G}_T}$ traitors (faulty processes) to solve Byzantine General Problem. As first impression, we could suggest and provide one formula as follow:

$$\mathcal{N} = 3 * \mathcal{N}_{\mathcal{G}_T} - \lceil \mathcal{N}_{\mathcal{G}_T} / 2 \rceil$$

Where $\mathcal{N}_{\mathcal{G}_T}$ is the number of traitors, \mathcal{N} is the total number of generals, and $\lceil \]$ is the round number as nearest integer closer to minus infinity. Verification of this formula is presented in the table 7:

Number of process ----- Number of faulty process	1		2		3		4	
	Agr	Iter	Agr	iter	Agr	iter	Agr	iter
3	Yes	870	No	-----	No	-----	-----	-----
4	Yes	315	No	-----	No	-----	No	-----
5	Yes	201	Yes	698	No	-----	No	-----
6	Yes	164	Yes	400	No	-----	No	-----
7	Yes	120	Yes	341	No	-----	No	-----
8	Yes	112	Yes	190	Yes	530	No	-----
9	Yes	117	Yes	175	Yes	334	No	-----
10	Yes	112	Yes	133	Yes	261	Yes	601

Table 6: Testing results

Note: Ag = Agreement, Iter = number of iterations

Number of traitors	Total number of generals needed to solve BGP	
	Experimental result	Formula
1	3	$3^1 - \lfloor 1/2 \rfloor = 3$
2	5	$3^2 - \lfloor 2/2 \rfloor = 5$
3	8	$3^3 - \lfloor 3/2 \rfloor = 8$
4	10	$3^4 - \lfloor 4/2 \rfloor = 10$

Table 7: Mapping between experimental result and our proposed Formula.

In order to confirm our proposed Formula, we tried to demonstrate the result informally. We start by defining the general parameters used to explain the approach. Byzantine General Problem involves obtaining agreement among a collection of generals, some of which may be traitors, the others are loyalist. Then the total number of generals N is equals to the sum of them.

$$\mathcal{N} = \mathcal{N}_L + \mathcal{N}_{Tr} \quad (1)$$

Traitor selects Attack/ Retreat randomly with probability $1/2$. Then decision of traitor can be considered as $1/2 * O_{Att} + 1/2 * O_{Ret}$, where O_{Att} and O_{Ret} are respectively the occurrence of Attack/Retreat messages verifying that . In general we have \mathcal{N}_{Tr} traitors then traitors decision is formulated as follow:

$$\mathcal{D}_{Tr} = \mathcal{N}_{Tr} * (1/2 * O_{Att} + 1/2 * O_{Ret}) \quad (2)$$

Loyalist selects Attack/Retreat according to $L_{RI_Automaton}$ and the probability of choosing Attack \mathcal{P} . then decision of Loyalist can be considered as $\mathcal{P} * O_{Att} + (1-\mathcal{P}) * O_{Ret}$. In general we have \mathcal{N}_L Loyalists then Loyalists decision is formulated as follow:

$$\mathcal{D}_L = \mathcal{N}_L * (\mathcal{P} * O_{Att} + (1-\mathcal{P}) * O_{Ret}) \quad (3)$$

The aim is to obtain an agreement among all generals and especially between traitors and loyalist to get the same decision. Then we need equalize the two equations given by (2) and (3).

$$\mathcal{D}_{Tr} = \mathcal{D}_L \iff \mathcal{N}_{Tr} * (1/2 * O_{Att} + 1/2 * O_{Ret}) = \mathcal{N}_L * (\mathcal{P} * O_{Att} + (1-\mathcal{P}) * O_{Ret}) \quad (4)$$

Our algorithm converges according to the probability \mathcal{P} and exactly when \mathcal{P} is closer to 1 or closer to 0. In first case, we suppose that \mathcal{P} is closer to 1 then \mathcal{P} is approximately equals to 1. In accordance with (1) and (4), we have a system with two equations as follow:

$$\begin{cases} \mathcal{N}_{Tr} * (1/2 * O_{Att} + 1/2 * O_{Ret}) = \mathcal{N}_L * (\mathcal{P} * O_{Att} + (1-\mathcal{P}) * O_{Ret}) \\ \mathcal{N} = \mathcal{N}_L + \mathcal{N}_{Tr} \end{cases}$$

From these two equations, we can extract the result below:

$$\mathcal{N} = (\mathcal{N}_{Tr} / O_{Att}) + \mathcal{N}_{Tr} * (3/2 * O_{Att} + 1/2 * (O_{Ret} / O_{Att}))$$

Or Probability of choosing "Attack" \mathcal{P} is closer to 1 then $O_{Ret} = 0$ and $O_{Att} = 2$. as consequence

$$\mathcal{N} = (\mathcal{N}_{Gr} / 2) + 3 * \mathcal{N}_{Gr}$$

As it is logic that \mathcal{N} is an integer then we can consider the term $(\mathcal{N}_{Gr} / 2)$ as a round number to the nearest integer closer to the minus infinity.

We get the same result if we suppose P is closer to 0. In this case, solution converges to Retreat decision. We just exchange the occurrence O_{Ret} and O_{Att} .

The figure 10 shows the collection of processes plays an important role to reach the agreement. Thus, the Automaton needs more iteration to solve Byzantine General Problem when the number of faulty processes increases. However, the Automaton needs less iteration to reach agreement when the number of total process increases. That's why we think to increase also the collection of processes during testing and evaluation phase.

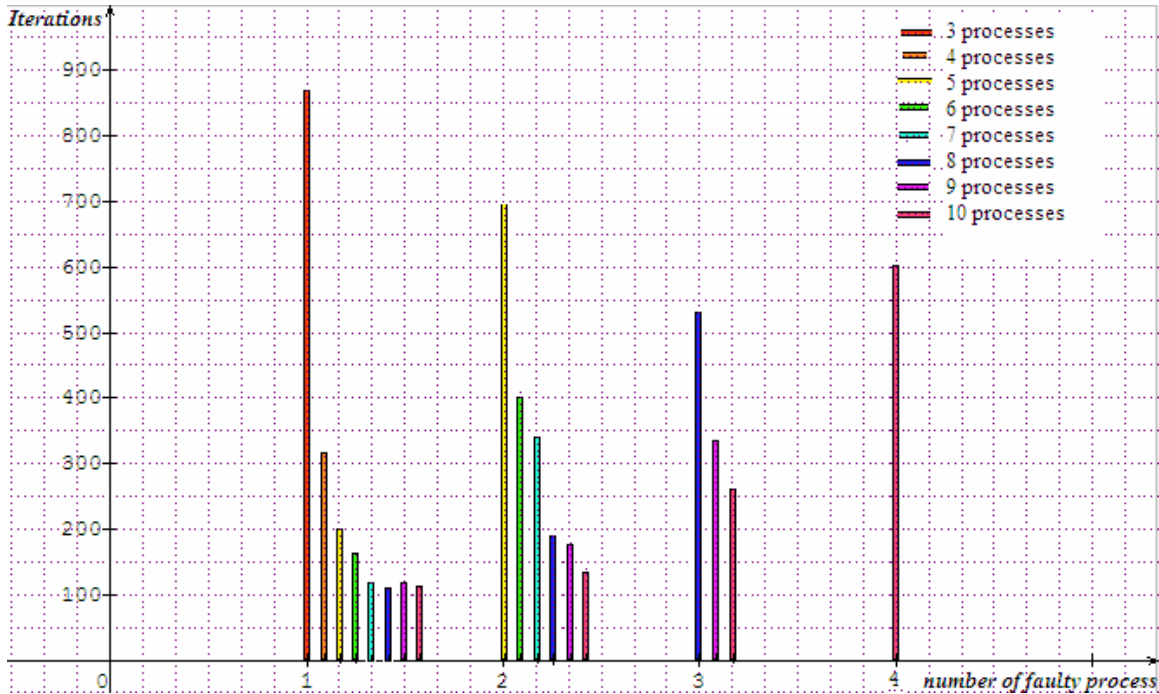


Figure 10

Figure 11 presents the percentage of iterations taken to solve Byzantine general Problem in the case of one faulty process varying the total number of processes. It confirms the result given by figure 12 that the Automaton needs less iteration to reach agreement when the number of total process increases.

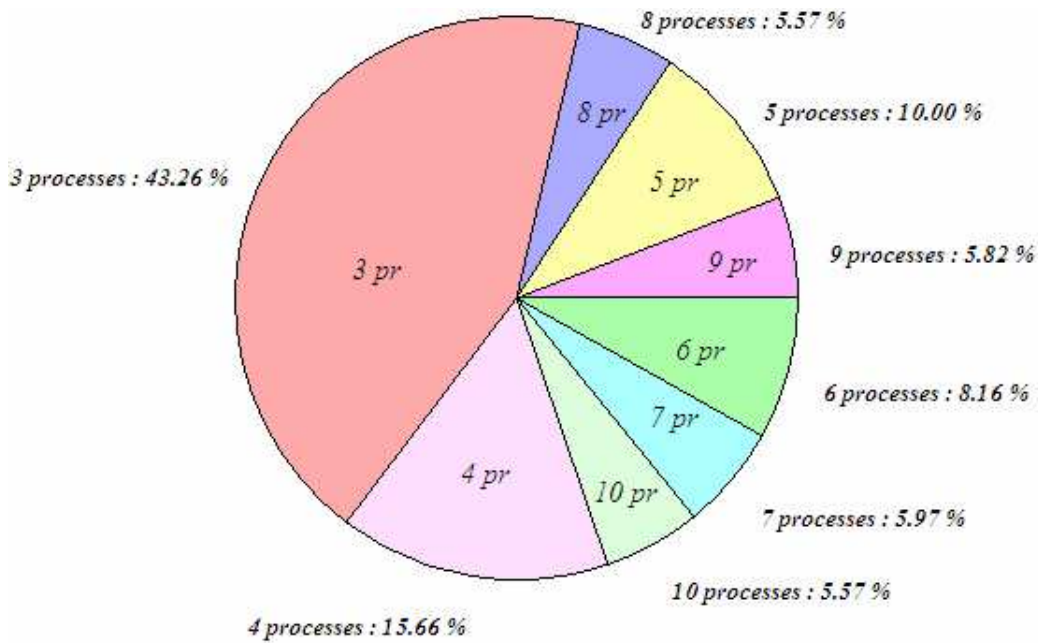


Figure 11

Figure 12 present also the percentage of iterations taken to solve Byzantine general Problem in the case of a collection of eight processes varying the number of faulty process. It confirms the result given by figure 12 that the Automaton needs more iteration to solve Byzantine General Problem when the number of faulty processes increases.

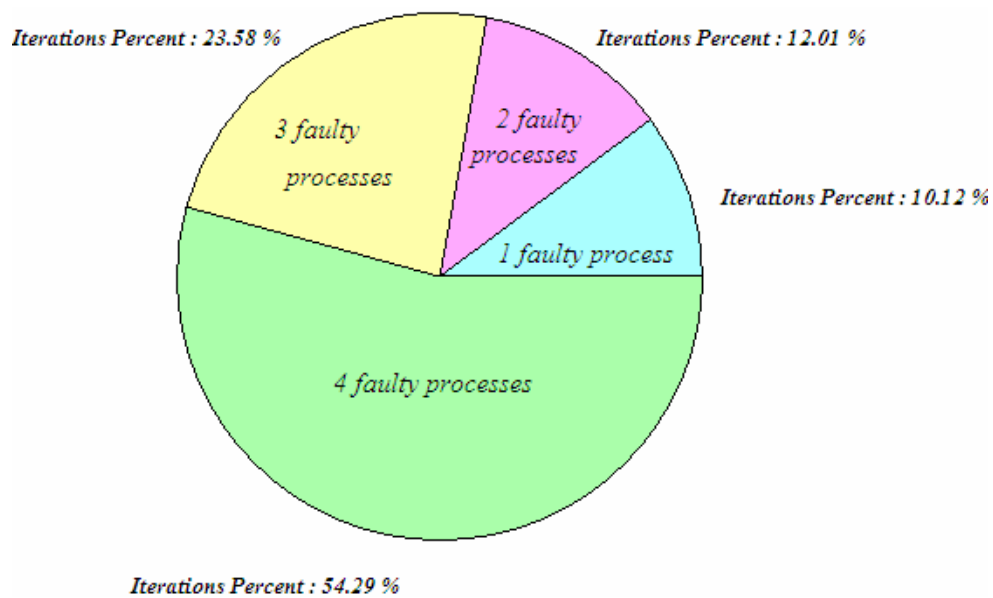


Figure 12

10. Discussion and Further work

Our initial task was to see whether the Automaton approach was a good alternative to use in intelligent distributed system solving Byzantine General Problem when communication is noisy. We have found that it works very well into reliable computer systems to handle malfunctioning components that give conflicting information to different parts of the system. The Automaton needs a different number of iterations to reach an agreement among a collection of process which of them can be a faulty process.

We believe that our novel scheme has a good result comparing to related work. It can be explained that previously work suggest that Byzantine General Problem can be solved only if fewer than one-third of the processes may fail, in other word only if fewer than one-third of the generals are traitors. With presence of k traitors, it should be a collection of N generals such that $N = 3*k + 1$. That means the number of traitors k should be equals to $(N-1) / 3$ which is fewer that one – third of process.

Our solution has an improved result to solve Byzantine General Problem. First we take in consideration of environment disturbance when communication could be noisy. Secondly, we suggest a new guideline for how many generals are needed to cope with k traitors. We find a general Formula for that basing both on experimental testing result and an informal demonstration. In fact, for k traitors, our algorithm need a collection of $N = 3*k - [k/2]$ generals to reach an agreement.

We have found that LRI-Automaton is a good solution to improve Byzantine failure if the loss of a system service due to a Byzantine fault in systems that require consensus. We believe that the system would always converge if the learning speed was slow enough and we waited sufficiently long. Then the parameter λ use by the Automaton to update the probability of choosing “Attack” has an influence on the time spending to reach an agreement. The less value of λ to be closer to zero, the more iterations are used to make a solution converge. For further work, we try to find the relation of this parameter on solving Byzantine General Problem. We take also on consideration the variation of noise factor because we have fixed his value in our work to 10 %

11. Conclusion

In this report, we have presented a working solution for Byzantine General Problem when communication is noisy using the linear reward-inaction Automaton (LRI-Automaton). The Automaton is able to handle fault-tolerance and especially Byzantine failure in reliable computing system.

We have proposed a new scheme and we developed a new algorithm during this project. Evaluating and experimental results have demonstrated that our solution can be used for solving Byzantine General Problem when communication is noisy to handle specialized tasks into intelligent distributed system and cooperate to achieve system-level goals and achieve a high degree of flexibility.

We also learned in the development process that for this type of solution depends of the total number of process to cope with a small number of faulty processes. Collection of processes plays a significant role to reach an agreement with malfunctioning of some of them comparing to the number of iterations.

The solution described to handle Byzantine General Problem has shown a good result comparing to related and previously work. Our conclusion is that the game of Learning Automaton and especially linear reward-inaction Automaton (L_{RI} -Automaton) is very suitable for Byzantine General Problem when communication is noisy. As it described in the problem description, as intelligent distributed system, L_{RI} -Automaton is a good solution to handle the Byzantine failure. It rectifies malfunctioning components that give conflicting information to different parts of the system in such reliable computing system. Our successful experiments have shown that processes were able to reach an agreement and the same decision when some of them are faulty process.

Appendix

A1 ***Glossary & abbreviations***

Short definitions of terms used and references to further relevant glossary sources

A2 ***References***

[1] Kumpati S.Narendra and M.A.L Thathachar. Learning Automaton – A survey

[2]<http://scholar.lib.vt.edu/theses/available/etd-5414132139711101/unrestricted/ch3.pdf>. Chapter 3: Stochastic Learning Automata

[3] D. GaYti. Intelligent Distributed Systems: New Trends

[4] Chris Laas. Fault Tolerant Computing 6.911 Architectures Anonymous

[5] Leslie Lamport, Robert Shostack, and Marshall Pease. The Byzantine Generals Problem

[6] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance