Simulating the Allocation and Reallocation of Critical Tasks in a Highly Dependable Distributed System

Rajanikanth R. Mitta

Computer Science Department

Arizona State University

Tempe, AZ

Raj.Mitta@asu.edu

Abstract

The aim of the project is to demonstrate a technique that supports the development of highly dependable applications in a distributed system environment. The technique involves task reallocation, load balancing, error detection and fault injection. The firewall application has been chosen for this demonstration, which will be simulated and the task reallocation algorithm is demonstrated by injecting fault into the system and then the fault is detected. If a computer node is giving erroneous results frequently, then it is considered to be permanently faulty, and the tasks associated with it are moved to other computer nodes. This is done in a way that will make sure there is a load balance among the different computer nodes. The load balancing will prevent communication bottlenecks.

The system is implemented by simulation in the framework of this project. The Java multithreading feature provided by Java has been used to simulate the various parts of the system like the packet generator, computer nodes, tasks, etc.

1 Introduction and motivation

Dependability is one of the key features of any distributed system. It guarantees that a task will be completed even if there are sub system crashes or faults in communication. In today's world when internet is growing at a fast pace, more and more systems are becoming distributed. Reference [2] defined dependability as the property of a computer system such that reliance can justifiably be placed on the service it delivers.

But the design and maintenance of distributed systems is very difficult for various reasons. One of them is the delay caused due to communications coordination and synchronization. To this if we add fault tolerance, then there will be more difficulties in designing such a system. Firewall is one of the applications that can benefit from the use of a dependable distributed system. As we know firewalls could become performance bottlenecks, as all information to and from an internal network has to pass through it. Most firewalls have rulebases with thousands of rules. More about firewall rules can be found in [3] and [4]. And every packet has to be checked against these rules before being allowed to pass through it. By introducing the distributed dimension to a firewall we can run multiple copies of the firewall in parallel. We can also improve the dependability or reliability of a firewall by checking every packet more than once in parallel and then comparing the results to make a decision on allowing the packet through.

In this report I simulate a technique given in [1] to demonstrate how effective and dependable a firewall could become if it is distributed and redundancy is introduced into it.

The paper is organized in the following way. Section 1 gives the introduction and motivation for the project. Section 2 will give the system overview (the big picture) of the system and how it has been simulated. Section 3 explains in general what a firewall is and the improvements being proposed. Section 4 explains in detail what each part of the tool is designed to do and its need. Section 5 explains all the algorithms used in the tool. Section 6 will give a brief explanation of the metrics being collected from the simulation. Section 7 explains the Graphical User Interface (GUI) used for the tool. In section 8, I will give the conclusion and talk about the future work. Finally at the end are the references I cited in this paper.

2 System Overview

The figure 1 shows the overall picture of the tool that will simulate a distributed firewall system. This figure just gives an example scenario where there are four computer nodes and five tasks to run on those computers. There are two tasks that run on single mode, two in double redundant mode and one in triple redundant mode. There is a fault-injector that will inject faults into the communication lines between the computer nodes and the tasks. The fault detection is done at the Tasks. The process starts when a packet generator creates a packet. It will be assigned to a task based on one of three algorithms: Random, Round-Robin and Queue-Length Based. The task forwards the packet to the computer node(s) and gets the result(s) of the processing that has been done. Depending on the redundancy, it will decide whether to accept the packet or not. If the packet is accepted, then it is placed in the message queue of the packet collector.

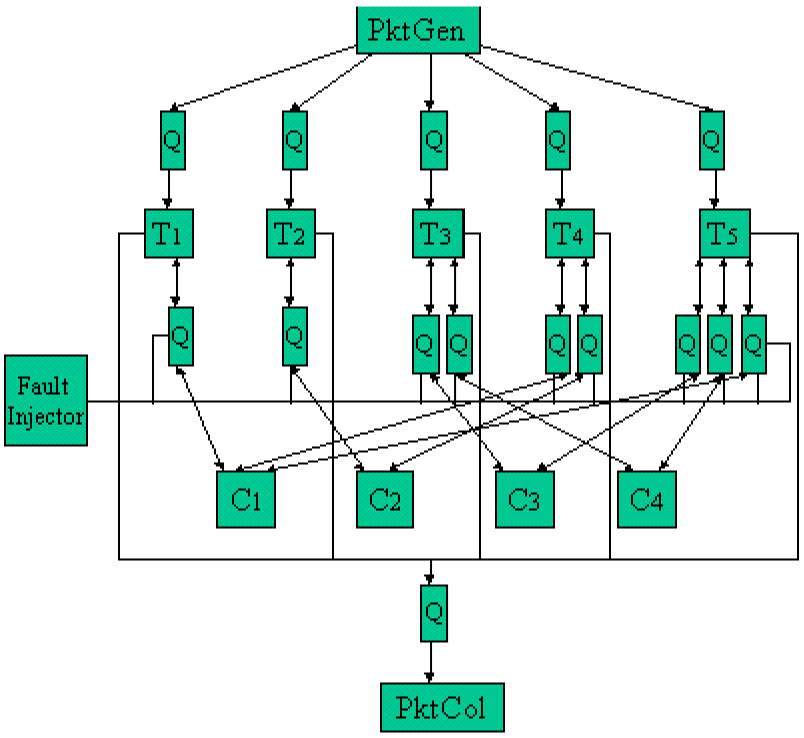


Figure 1. System Overview

In the next few paragraphs I will explain the separate system components in detail. Let's start with the packet collector. This module generates packets and decides whether it should be processed on a single, double or triple redundant task. In the current implementation, the decision is made in random for simplicity. The module can be easily modified to base the decision on user's assignment of criticality of tasks. The packet is then assigned to a task based on one of the three algorithms mentioned above. Just before placing in the queue the packet is time stamped to measure the amount of time the packet spends in the queue and in processing before reaching the packet collector.

The queues have been implemented based on the producer-consumer problem. When the tasks get packets from their respective queues, they send the packet and its copies, if any, to the computer node(s) associated with that task. The communication line between the task and the node is where the fault injector comes into the picture. Here faults are injected randomly or the user can specify the level of fault injection. The computer node parses the packet and gets the needed information and does the processing. It decides whether the packet should be accepted or not and sends the result back to the task.

The task depending on its redundancy takes different actions. If the task is executing in single mode, then that result returned by the computer node is taken as it is and that packet is accepted or rejected. If the task is running in double redundant mode, then if both the computers agree that packet is accepted or rejected according to their decision. In case they both disagree, the packet is rejected and the nodes are given a penalty of one each. This penalty is entered into a fault table. The fault table monitors the number of penalties that each node has received and when the penalty reaches a certain pre-determined level, the node is shut down (it is decided that there is a permanent communication fault with that node) and the tasks running on it are migrated to other nodes. If the task is running in the triple redundant mode, then the majority decision is accepted. In case one of the nodes disagrees with the other two, then that node gets a penalty of two.

The nodes accepted by the tasks are time stamped and sent to the queue of the packet collector. The packet collector measures all the statistics for the simulation. Now, to talk more about the task reallocation. When a node is shut down, the tasks are reallocated to other nodes based on an algorithm that is derived from the one explained in [1]. More about the algorithms used can be found in section 5.

To simulate the distributed system, I used the feature of multi-threading provided by the Java programming language. Reference [5] provides examples and code for using this feature of Java. The different parts of the system that are run in parallel using the threads are packet generator, load balancer, tasks, computer nodes, task reallocator and the packet collector.

A thread is very helpful in simulations as we do not have the distributed system to test our techniques. The distributed system has to be simulated using threads or by creating child processes. A thread by itself is not a program as it cannot run on its own. It runs as part of a program. And a program can run any number of threads. All the threads run in parallel and we can communicate from one thread to another just like in a distributed system. But, we also need to introduce random delays in communication to simulate the real world applications.

3 Firewalls

The application chosen for the simulation is the firewall. Detailed description of various types of firewalls and issues related to them can be found in [6]. Here I will give a brief description of a firewall and some issues related to it. As shown in figure 2, firewalls are basically designed to keep unauthorized traffic away from the intranet or private networks. Firewalls filter packets based on the source address, destination address and criteria defined in the firewall rules by the firewall administrator. The main purpose of a firewall is to protect computers and devices on the intranet side of the firewall from those on the internet side.

Not only that, but different sections of an organization could have firewalls between them to protect them from each other. There are two ways in which a firewall can be configured. In one method we can block traffic from some sites or packets that contain some keywords. This has a drawback that the system administrator has to make sure that the firewall rules are upto date. In the second type of configuration only traffic from specific sites or users is allowed to pass while the rest of the traffic is blocked. The advantage of this is that the firewall rules will be less and the time and computational power needed decreases. In this tool the firewall has been configured to run in the second method.

Reference [7] gives a more detailed description of the load balancing firewalls. We need load balancing firewalls to increase the throughput (performance) of the system and to avoid bottlenecks. If we have only one firewall (that is the firewall running on only a single node), then if there is a failure of that node, the whole connection between the internal private network and the external internet is lost till the system is up and running again. To avoid this scenario, we can use more than one computer node to work as the firewall. There should be a load balancer on both sides of the firewall to balance the amount of packets coming into and going out of the intranet among the available nodes.

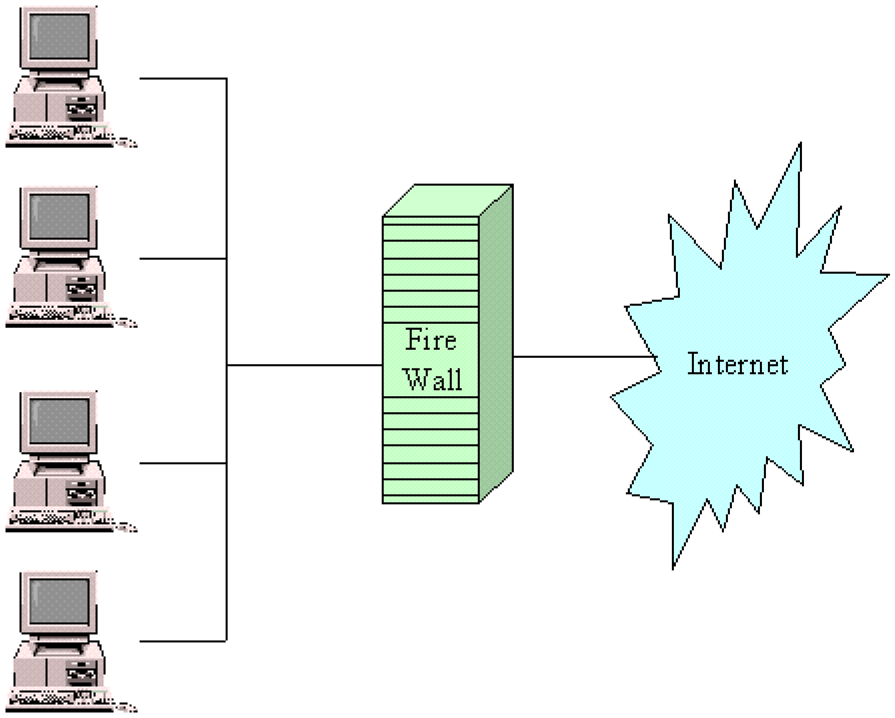


Figure 2. Intranet separated from the internet using a firewall

Even with all these features some times a computational or communication error might cause the firewall to let malicious traffic through while blocking legitimate traffic. The legitimate traffic when blocked will be retransmitted till it is accepted if TCP is used, but this causes a delay which could be fatal for some critical systems like transaction processing. To avoid these kinds of mishaps that could be very costly, we introduce redundancy into the checking of network traffic. The various services and systems inside the internal network can register with the firewall and tell it how much redundancy they want for the verification of packets destined for them. When a packet arrives, the source destination is checked and the packet will be sent to a task that will verify the packet the number of times the source wants it to and depending on the majority result sends the packet through or blocks it.

4 The Tool

In this section, I will explain in detail what has been implemented in this project and what each part of the simulator does and its need. The major parts of the simulator are:

1. Packets
2. Packet Generator
3. Load Balancer
4. Queues
5. Tasks
6. Computer Nodes
7. Packet Collector
8. Fault Injector
9. Fault Detection
10. Fault Table
11. Task Reallocation
12. Statistics

- Packets:

  The packets used in the simulation are structured and of variable length. They allow time stamping and can carry any type of messages.

- Packet Generator:

  The packet generator gets the input from the user on how many packets to generate. Depending on the user's choice of having tasks that are single, double or triple redundant tasks the packet generator will randomly choose which type the packet should be. The packets are then sent to a program that handles the load balancing. At the end of generating all packets, the packet generator sends an "End of Experiment" message to the load balancer.

- Load balancer:

The load balancing part of the tool looks at the user choice of algorithm and distributes the packets accordingly. It is very critical to the system that the load balancing algorithm chosen is fair and does not swamp one task with many packets and leave some tasks no packets at all. When the load balancer receives a packet, it parses it and finds the redundancy of the packet. And using the user specified algorithm sends the packet to the queue of one of the tasks of the same redundancy. Just before placing the packet in a queue it is time stamped for statistical purposes. When it receives the "End of Experiment" message, it sends the message to all the tasks.

- Queues:

  All communication in the system is done using message queues. These queues have been based on the unbounded buffer producer-consumer problem, i.e., the consumer will block if the queue is empty, but there is no restriction on the producer as there is no upper bound on the buffer.

- Tasks:

  Three different types of tasks are used. Ones that execute only one copy of the packet (here by execution I mean, it sends the packet to a computer node to verify whether that packet is valid or not), two copies of the packet or three copies of the packet. Depending upon the user input the number of each type of these tasks is determined. The task sends the packet to the computer node(s) and waits for the result(s) from the node(s). If it is executing only a single copy of the packet, and then if the node tells the packet is valid the task will accept the packet and let it through the firewall to the packet collector, otherwise, it rejects the packet. If it is executing two copies, then it will accept the packet only if both the nodes agree that it is a valid packet. For all other cases the packet is rejected. If the task is executing three copies, then depending on the majority of the results it either accepts or rejects the packet. For a packet to be accepted or rejected at least two nodes must either accept it or reject it respectively. When it receives the "End of Experiment" message, that task will pass the message on to the packet collector.

- Computer Nodes:

  All the processing of packets is done right here. They decide if a packet needs to be accepted or rejected. A computer node reads a packet from the queue assigned to it and processes it. And sends the result back to the task that sent the packet. Here it should be noted that the computer node has no knowledge whether it is executing a duplicate copy or the original packet and it does not even need to know.

- Packet Collector:

  This part of the tool is used to collect all the packets that are allowed by the firewall to pass through it. When a packet reaches the collector, it is time stamped again for statistical purposes. When the packet collector receives the "End of Experiment" message, it increments a counter. When the counter equals the number of tasks running in the system, the collector stops reading from its queue as all tasks have ceased to execute.

- Fault Injector:

  To study the behavior of the system under stress, we can introduce fault into the communication lines. This tool enters a 5% error into the communication link between the task and the computer node. The percentage of fault injection can be changed for different simulations. The fault is entered when the tasks are writing the packet into the queues assigned for communication with the computer nodes.

- Fault Detection:

  Fault detection is done at the tasks. Fault detection can be done only if the tasks are running in double or triple redundant modes. A fault is detected when two computers return different results for the same packet. If it is running in double redundant mode, the task does not know which computer is faulty. So, it just rejects the packet. In triple redundant mode, if two computers return the same result and one disagrees, then the task knows that the fault is with the communication link between it and that computer node.

- Fault Table:

  A fault table is maintained in the system to keep track of faults that have been detected. This helps us in isolating the nodes whose communication links are faulty and which need to be shutdown. In the double redundant mode, if a fault is detected, both nodes are given a penalty of one. In triple redundant mode, if a fault is detected, the node that is disagreeing with the other two nodes is given a penalty of two. When a computer's penalties cross ten in the fault table, then it is time to shut it down and reallocate the tasks assigned to it to the other computer nodes.

- Task Reallocation:

  Whenever a computer node is shutdown, all the tasks assigned to it need to be reassigned to other computer nodes. This process has to be done in such a way that the two conditions mentioned in section 5 of this report are satisfied. For each task on the node that has been shutdown, we find the node that has the least number of tasks assigned to it and we assign this task to that node. If there is violation of condition I, i.e., if a node has more than one copy of the same node, then it is calculated to find a task on another node with which this task can be swapped and the condition be met. By choosing the node with the least number of tasks at each iteration we satisfy the condition II that states that the task allocation should be balanced. In some circumstances if there are not enough nodes to satisfy condition I, an error message is generated and the experiment is halted with the display of statistics generated so far.

- Statistics:

  When the packet collector receives the "End of Experiment" message or if the task reallocation algorithm stops the experiment as mentioned above, the final statistics are calculated and displayed to the user. Right now five metrics are being calculated. They are number of packets collected by the packet collector, number of seconds the simulation took, number of packets per second that went through the firewall, total response time and the average response time. More on the statistics generated could be found in section 6 of this report. The time stamping done at the packet generator and at the packet collector helps in calculating most of these metrics.

The simulation tool has been completely implemented and is running. The tool is installed and runnable at the following website: http://www.public.asu.edu/~rmitta/FireWall.html

5 Algorithms

There are five algorithms used in this simulation. Three are for achieving load balance during packet distribution, one is used for initial task allocation and another for the dynamic task reallocation.

The three algorithms used for achieving load balance are Random, Round-Robin and Queue-Length based. Here is a brief description of each of these algorithms:

- Random:

  Whenever a packet is generated, we randomly choose the task to which it will be allocated. It can only choose from tasks that are of the same redundancy as the packet is intended to be.

- Round-Robin:

In this allocation, the packets are allocated on a round-robin basis, i.e., all tasks will get equal number of packets.

- Queue-Length Based:

  Packets are allocated to the task whose queue has the shortest length, i.e.; the task whose queue has the least number of packets waiting in it will be assigned the next packet.

The user has the choice of selecting which algorithm to use for the simulation. My experience with the tool has shown me that Queue-Length based algorithm is the best, as different packets need different processing times. And if we allocate packets in the other two ways, there might be a packet that needs more processing time and all the packets that are waiting after it in the queue will be delayed.

The initial task allocation algorithm is used to allocate the task in a balanced way. Before I explain about the algorithm let me introduce some terminology. We assume there are m tasks that have to be allocated on n nodes in the system. Each task has one, two or three redundant copies. The total number of tasks that have only one copy are represented by t1, that have two copies by t2 and that have three copies by t3. Now the total number of copies of all tasks in the system is K = t1 + (t2 * 2) + (t3 * 3). To achieve a balanced task allocation the total number of tasks running on a node C must satisfy the following inequalities:

$P_{min} \leq |C| \leq P_{max}$, where

$P_{min} = \lfloor K/n \rfloor$

$P_{max} = \lceil K/n \rceil \leq P_{min} + 1$

That is, the overall difference between the number of tasks running on any node cannot be more than one. Now let me explain the algorithm for initial allocation of tasks.

```
For j = 1 to K
    i = (j – 1) mod n + 1
    Allocate task copy j to node $C_i$
```

This algorithm will allocate copies of the same task to different nodes if $n \geq P_{max}$. The complexity of the algorithm is O(K), where K is the total number of copies of all tasks in the system, including the redundant copies. The following example shows how the tasks are initially allocated. Let us suppose the user has entered the following data: 5 computer nodes, 2 tasks that run in single mode, 3 tasks that run in double redundant mode and 2 tasks that run triple redundant mode. Following the algorithm explained above the tasks are allocated in the following way:

Node 0	**Node 1**	**Node 2**	**Node 3**	**Node 4**
T[0:1]	**T[1:1]**	**T[2:1]**	**T[2:2]**	**T[3:1]**
T[3:2]	**T[4:1]**	**T[4:2]**	**T[5:1]**	**T[5:2]**
T[5:3]	**T[6:1]**	**T[6:2]**	**T[6:3]**	

The notation followed here is that T[4:2] means the second copy of the fourth task.

The final algorithm is the dynamic task reallocation. If any one of the nodes is shut down due to some fault, then the copies of tasks assigned to it must be reassigned among other nodes. The task reallocation must satisfy the following two conditions:

- Condition I:
  The copies of the same task should be allocated to different nodes.
- Condition II:
  The task allocation should remain balanced, i.e., the difference between the number of tasks running on any two nodes should not be greater than one.

The algorithm used for dynamic task reallocation is based on the algorithm presented in [1] with some minor changes for accomplishing balanced task allocation among the nodes. The pseudo code for the algorithm is given below. For each copy of a task on the node C that has been shut down, find the node that has the least number of tasks running on it. This node should not be C. Then allocate the task to that node. If there is a violation of condition I mentioned above, then for each node find whether swapping this task with one of its tasks will remove the violation. If it does, then swap the tasks.

```
For each task ti on C
    Find node Cj with the least # of tasks such that Cj ≠ C
    Allocate ti to Cj

    If there is a violation of condition I by this task ti
    Then
        For k = 1 to n
            For each task t on Ck
                 If (Ck ≠ Cj) and (Ck ≠ C)
                 Then
                     If Swap (ti, t) will remove violation
                     Then
                        Swap (ti, t)
                     End If
                  End If
            End For
        End For
    End If
End For
```

According to [1], the complexity of this algorithm is O(K).

Now let me give an example, consider the initial task allocation in the previous example. Now let us suppose that node 2 has become error prone and needs to be shut down. The following tasks T[2:1], T[4:2] and T[6:2] need to be reallocated in such a way that both the conditions are met. T[2:1] is assigned to the computer node with the least number of tasks. In this example it is node 4. Next T[4:2] should be assigned to the node with the least number of tasks. But, all the nodes have equal number of tasks (3 each). So, we select the first node, i.e., node 0. After that T[6:2] is assigned to node 1. The tasks after reallocation look like

Node 0	**Node 1**	**Node 2**	**Node 3**	**Node 4**
T[0:1]	**T[1:1]**		**T[2:2]**	**T[3:1]**
T[3:2]	**T[4:1]**		**T[5:1]**	**T[5:2]**
T[5:3]	**T[6:1]**		**T[6:3]**	**T[2:1]**
T[4:2]	**T[6:2]**			

As we can see, this violates condition I. T[6:1] and T[6:2] are assigned to the same node. Now the algorithm tries to swap T[6:2] with tasks on other nodes so that condition I is satisfied. It check to see if swapping of T[0:1] on node 0 with T[6:2] on node 1 will remove the violation. Since it does, the swapping is done. In case it did not, then it would have checked with the next task on node 0 and soon with all tasks on all nodes till it finds a task with which task T[6:2] can be swapped. Now T[0:1] and T[6:2] are swapped. The resulting task reallocation satisfies both the conditions.

Node 0	**Node 1**	**Node 2**	**Node 3**	**Node 4**
T[6:2]	**T[1:1]**		**T[2:2]**	**T[3:1]**
T[3:2]	**T[4:1]**		**T[5:1]**	**T[5:2]**
T[5:3]	**T[6:1]**		**T[6:3]**	**T[2:1]**
T[4:2]	**T[0:1]**			

6 Statistics

The following metrics are being generated at the end of the simulation. More could be added if necessary.

1. # of packets collected
2. # of seconds the simulation took
3. # of packets per second
4. Total response time
5. Average response time

- # of packets collected:

  This metric measures the number of packets that went through the firewall, i.e., accepted by the firewall as valid packets.

- # of seconds the simulation took:

  This is the total number of seconds the simulation took. The starting time is noted when the first packet is generated and ending time is noted when the last "End of Experiment" message is received by the packet collector. The difference between the ending time and starting time gives the total time of the simulation in seconds.

- # of packets per second (Throughput):

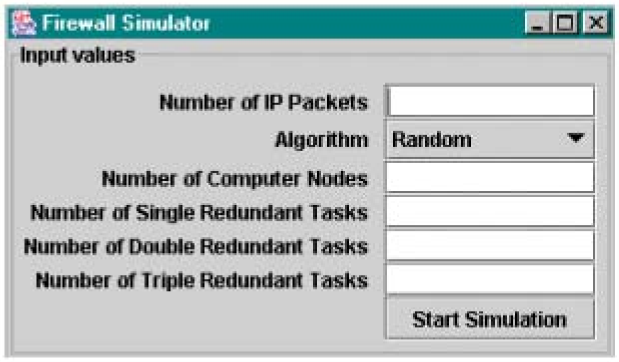
  The total number of packets collected divided by the total number of seconds the simulation took gives the number of packets that went through the firewall per second. This is the throughput of the system.

- Total response time:
  The difference between the two timestamps put on a packet gives the response time for that packet. The response time is defined as the time a packet spent in the queue and in the processing before being allowed through the firewall. The total response time is calculated by adding together the response times of each packet that arrived at the packet collector.

- Average response time:
  The average response time is defined as the time a packet spent on the average in the queue and in processing before being allowed through the firewall. This is calculated by dividing the total response time by the total number of packets collected.

7 Graphical User Interface

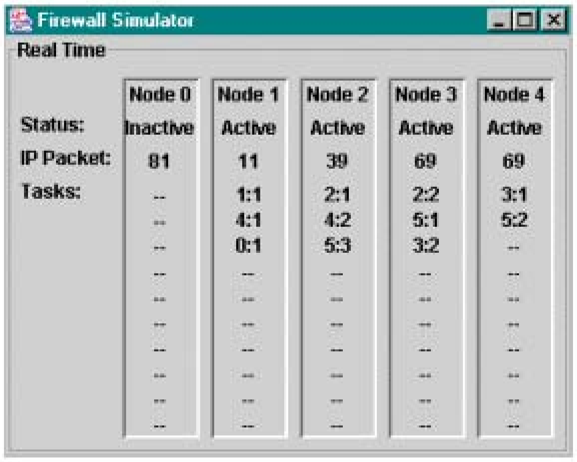
Showing the results being generated by the simulation in a GUI (Graphical User Interface) instead of plain text on the command prompt will help us to see all the data as it is changing instead of scrolling through the output text after the experiment has been concluded. The graphical interface was developed under an Honor's project, as a sub project of this one. He was assigned the task of creating a window that takes input, a window that shows the



Number of IP Packets
Algorithm: Random
Number of Computer Nodes
Number of Single Redundant Tasks
Number of Double Redundant Tasks
Number of Triple Redundant Tasks

Figure 3.1

simulation as it is running and a window to show the final statistics. The screen shots of the three windows are shown in figures 3.1 to 3.3. Figure 3.1 is the input window, figure 3.2 is the simulation window and figure 3.3 is the final statistics window. The GUI was created using the JSwing API of Java. I will explain more about JSwing in the next paragraph.

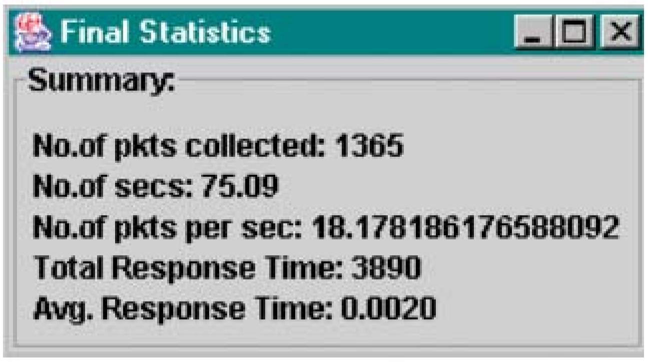


Firewall Simulator

Real Time

	Node 0	Node 1	Node 2	Node 3	Node 4
Status:	Inactive	Active	Active	Active	Active
IP Packet:	81	11	39	69	69
Tasks:	--	1:1	2:1	2:2	3:1
	--	4:1	4:2	5:1	5:2
	--	0:1	5:3	3:2	--
	--	--	--	--	--
	--	--	--	--	--
	--	--	--	--	--
	--	--	--	--	--
	--	--	--	--	--
	--	--	--	--	--
	--	--	--	--	--

Figure 3.2



Final Statistics

Summary:

No.of pkts collected: 1365
No.of secs: 75.09
No.of pkts per sec: 18.178186176588092
Total Response Time: 3890
Avg. Response Time: 0.0020

Figure 3.3

The JSwing feature of Java allows the development of a wide variety of graphical interfaces. In the input window, the user can enter the values for the simulation. Then we check if the input values are valid or not. If they are not, then a warning message will be shown and the user will be given another chance to enter the inputs. Once valid inputs are given, they are passed on to the simulator. The simulator then starts the simulation of the firewall. Now another window is opened that shows the nodes that are running, if they are active or inactive, the IP packet that node is executing at that moment and the tasks assigned to the nodes. Once the simulation is complete another window is opened and the statistics calculated for the simulation are shown. To display the windows we have used JPanel's and JoptionPane's are used for displaying warning messages.

8 Conclusion and future work

In this report I first briefly introduced dependable distributed systems and the technique used to make them dependable. And then I explained the overall system structure and the different algorithms that are used for achieving the high dependability. I had given some examples of initial task allocation and task reallocation. All the requirements given to me have been completed. But, still some future work needs to be done to improve the system. They are explained below.

More control and measurement parameters can be introduced in the tool, for example, the fault injection rate, task criticality, communication delay factor, throughput, individual task delay and reliability.

The tool will be simulated for different N, where N is the number of computer nodes. Test cases will be developed to run the tool for different combination of tasks. For example, one test case could say that the tool needs to be run using (2-3-1) combination of tasks. Here the first number corresponds to the number of tasks running in single mode. The next number corresponds to the number of tasks running in the duplicate mode and the final number corresponds to the total number of tasks running in the triple mode. Here are a few examples of test cases, (0-0-1), (10-3-0), etc. For each different value of N, we need to run all the test cases to find N, for which the system throughput is the most optimal.

The current version of the tool does not support reintegration. Reintegration of the tasks is done once a computer node that went down comes back online. During reintegration the tasks should be taken from nodes that are running the maximum number of tasks and allocate them to the new node. Care should be taken that both conditions I and II discussed in section 5 are satisfied after the reintegration.

Due to the time limit, I only implemented the tool that supports experiments on dependable distributed systems. More experiments can be conducted to measure and analyze, for example, the relationship between the throughput and

- the number of available nodes
- the number of tasks and redundant tasks
- the individual communication delays
- the fault injection rates

I am creating a web page for this tool. Right now only the executable file and this report are available at http://www.public.asu.edu/~rmitta/FireWall.html. In the future it will have the results of experiments done and users guide. I will also try to improve the GUI for the tool. In this version of the tool, it displays about four different windows. I will try and fit all that information into one window so that the user can see the whole simulation in one window.

A References

[1] Yinong Chen, "Developing Highly Dependable Application in a Distributed System Environment". CSE Department, Arizona State University.

[2] J.C. Laprie, "Dependable Computing and Fault Tolerance: Concept and Terminology", *IEEE 15th Annual International Symposium on Fault-Tolerant Computing* (FTCS-15), Michigan, June 1985, pp. 1 – 11.

[3] Yinong Chen, R. Mateer, "Performance Simulation of a Dependable Distributed System, Simulation, Special Issue on Modeling and Simulation Applications in Scheduling Multiprocessor Systems", Simulation Councils Inc., ISSN 0037-5497/01, Vol.77, No. 5-6, November/December 2001, pp. 230 – 237.

[4] S. Hazelhurst, A. Attar, R. Sinnappan, "Algorithms for Improving the Dependability of Firewall and Filter Rule Lists", *The International Conference on Dependable Systems and Networks*, New York, June 2000, pp. 576 – 585.

[5] Java Tutorial, http://developer.java.sun.com/developer/onlineTraining/

[6] Firewalls: A Complete Guide, Marcus Goncalves, 2000, The McGraw-Hill Companies Inc.

[7] Load Balancing Servers, Firewalls and Caches, Chandra Kopparapu, 2002, John Wiley & Sons, Inc.