Simulating Highly Dependable Applications in a Distributed Computing Environment

Yinong Chen

Computer Science & Engineering Department, Arizona State University,
Tempe, AZ 85287-5406, USA, yinong@asu.edu

Abstract

The aims of the research are to investigate techniques that support the development of highly dependable applications in a distributed system environment. Techniques we developed include redundant task allocation, load balancing, fault-tolerant computing and communication. The application we have implemented in the system is a firewall application. The firewall runs in redundant mode. Each incoming or outgoing packet is checked by two or more copies of the firewall application. Only when the majority of the firewall copies decide to accept the packet, the packet can go through the firewall. Different decisions from the different firewall copies signify a possible hardware fault or a software error in the underlying system. This paper reports a recent implementation of a simulation system, in which computing nodes, redundant copies of tasks and packet queues are implemented as independent threads, as well as experiment result we obtained based on queuing models.

Keywords: *Distributed system, task allocation, load balancing, firewall, queuing*

1. Introduction

Dependability has been defined as the property of a computer system such that reliance can justifiably be placed on the service it delivers [1]. Different kinds of software and hardware dependable techniques have been developed to produce various kinds of highly dependable systems with different dependability attributes, including reliability, availability, fail-safe, etc. Highly dependable systems are traditionally used only in safety-critical control and monitoring systems like nuclear reactors, flight control and traffic scheduling, etc. The recent developments in pervasive computing, embedded systems, wireless communication and using commercial off-the-shelf components to build dependable systems have greatly encouraged the use of dependable computing techniques in cost-sensitive commercial systems [2]. For example, 20% to 30% of the value of a top range car today comes from its on-board embedded system, including a dozen of networked high performance processors, several megabytes of program code and relevant circuits that interface on-board computers to sensors and actuators. The system controls various parts of the vehicle. Some of the operations are safety-critical, for example, engine control, anti-block system, and airbag deployment, and dependable system techniques have to be employed [3].

The design of distributed systems is known to be very difficult for a number of reasons. Maintaining the integrity of global state information, reducing latency and performance bottle-neck caused by communication, coordinating and synchronizing concurrent behaviors, and the need for a higher degree of fault-tolerance pose significant scientific and engineering challenges that are far from being met. Implementing a dependable distributed system in an embedded system environment adds further dimensions of complexity and challenges. For example, vastly diverse requirements on performance, reliability and cost-effectiveness, limitation on the size of components, power consumption and field environment make the techniques required for the development of such a system of great scientific interest with exciting application potentials.

Firewall application is another example where a dependable distributed system can be used to improve its performance and dependability. Like any security checkpoint, the firewall between networks is known to be a performance and dependability bottleneck. First, all packets leaving and arriving have to go through the firewall. A typical firewall has a rulebase of several thousands of rules and each packet going through the firewall has to be checked again the rules [6-8]. A distributed system can improve the performance by running multiple copies of the firewall task in parallel. Cisco's DistributedDirector [9] and Checkpoint Software Technologies' FireWall-1 [10] used parallel copies of tasks to improve the performance. A dependable distributed system can also improve the dependability by checking each packet by two or more independent copies of the firewall task and then comparing the decisions of the redundant copies [4-8]. To avoid common-mode failures, the redundant copies and the comparison units must run on different computer nodes or have a higher level of dependability.

In the past a few years, we have developed and implemented a prototype of a dependable distributed system on a local area network [4-8]. The components used were diskless Intel Pentium computers connected by Ethernet network cards. The overall system design was reported in [4]. The reliability modeling was proposed in [5]. The prototype implementation and the performance measured on the prototype were presented in [6]. The implementations of the firewall rulebase based on the system were investigated in [7-8]. Although the prototype gave us realistic data on the dependability and performance of the system, it is complex to use and difficult to add new experiments on the system. On the hardware prototype, we only collect data to evaluate the throughput and reliability of connections between directly connected pairs of computing nodes [6].

We have recently implemented a more sophisticated simulation system of the dependable distributed system. The system allows us to experiment new algorithms and techniques flexibly and quickly. Currently we are investigating load balancing algorithms and their performances under the redundant and parallel task allocation. These are the topics of this paper. We first briefly outline the simulation system we developed and the experiments and simulation we are conducting on the system.

In the next section, we outline the structure and the main components of the system. Balanced task allocation algorithms are studied in section 3, including an initial task allocation algorithm and a dynamic task reallocation algorithm. The load balancing algorithms for packet distribution among the firewall nodes, three queuing models and simulation results based on the three queuing models are presented in section 4. Section 5 concludes the paper.

2. System Description

The simulation system we are developing is shown in figure 1. The system is implemented in a Java multi-thread and synchronized programming environment. Each module in the diagram is a thread or a group of threads. At the bottom layer the system consists of a set of nodes. A graphic interface is used to configure the system by assigning the number of nodes, the number of tasks and the number of replicas of each task. It also displays the states of the system including the working nodes, failed nodes, the packets in each queue, the replicas on each node, and the experiment data measured. In the current system, we only implemented a firewall application and thus all tasks are parallel and redundant copies of the firewall application. Nodes running firewall applications are FW nodes in the system. The incoming and outgoing packets are generated by two packet-generation nodes (TG) that simulate the two sides of the firewall, e.g., the Internet and the Intranet, respectively. The results from redundant copies of firewalls will be checked by a fault-tolerant protocol.

The function modules in the system are explained as follows.

(a) Task allocation and load balancing

There are two factors that affect the response time of the system: Task allocation and the load balance. Although we only have one user task, the firewall application, we have a number of service tasks supporting fault-tolerant computing. They are fault-tolerant protocols, error logging, evaluation, reporting and reconfiguration. According the criticality of tasks, a task can be executed at single, double or triple redundant modes. Redundant copies of the same task must be allocated on different nodes so that the task can survive when a fault occurs in a node. The number of tasks allocated on each node should be dynamically balanced so that one node is not much heavily loaded than another. The second factor is the distribution of work to be handled by each task. The workload of a firewall task is the incoming and outgoing packets to be checked. If packets arrive faster than the firewall task can handle, a queue will be formed for each copy of the firewall task. The purpose of the workload balancing is to keep the waiting time in each queue approximately the same.

We will focus on the workload balancing in this paper and use a queuing model to evaluate the queuing time and remain the balance of the queuing times on each queue.

(b) Fault-tolerant protocols

A comparison protocol and a voting protocol are built on in the underlying communication system. It exchanges and compares (votes) the output of redundant copies of tasks in double or triple redundant mode. A disagreement in comparison indicates a transient error in one of the computing nodes or communication links involved. A disagreement with the majority in voting indicates a transient error in the computing nodes or communication links involved. The accumulation of transient errors indicates possible permanent fault and reconfiguration requirement.

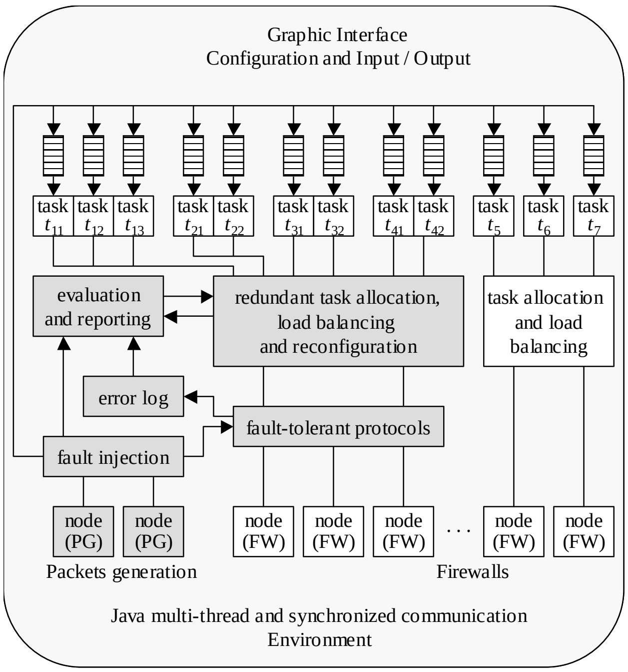

Figure 1. Overview of the simulation system

(c) Error reporting, evaluation, reconfiguration and task reallocation

Errors recorded by the comparison protocol are used by the evaluation system to assess the reliability, to diagnose and to report transient and permanent faults in the system, and to trigger the reconfiguration (removal or replacement) of computing or communication components. The reconfiguration is implemented by a task reallocation that exclude the fault node from participating in executing the tasks. Workloads need to be rebalanced among surviving nodes. Repaired or replaced nodes will be reintegrated into the system and reconfiguration is again need to reallocate the task to include new nodes.

(d) Testing environment

In the test stages, test cases will be generated to test the application in a systematic way. A fault injector will be used to create possible scenarios of faults and their manifestations (errors). To test and compare our highly dependable distributed system environment, we will have some firewall copies running in single mode and others running at the double and triple redundant modes. The copies in single mode will be tested by ordinary test cases, while those that are in redundant modes will be tested under fault injection.

3. Task Allocation

Figure 2 shows a possible scenario of task allocation and load distribution. A replica t_{11} of task t_1 and a replica t_{21} of task t_2 are running on node c_1; Replicas of tasks t_1, t_2 and t_3 are running on node c_2; Replicas of tasks t_3 and t_4 are running on node c_3; Replicas of tasks t_4 and t_5 are running on node c_4; Replicas of tasks t_6 and t_7 are running on node c_5. As we can see that in this example each node is running either two or three replicas of tasks and thus we cannot make a better task allocation. That is, the task allocation in figure 2 is balanced.

Now let's examine the tasks and their replicas. Task t_1 has three replicas (in triple redundant mode) running on nodes c_1, c_2 and c_3. An incoming/outgoing packet sent to t_1 will be replicated into three queues and checked by the three replicas of the task, respectively. The results (decisions) will be voted by a voting protocol. The voting protocol will synchronize the replicas and choose the majority. Tasks t_2, t_3 and t_4 are in the duplicate mode, respectively. That is, an incoming/outgoing packet will be checked by two replicas of a task and the decisions are synchronized and compared by the comparison protocol. Tasks t_5, t_6 and t_7 have only one copy each and they check the incoming/outgoing packets independently. The highly dependable system environment is in fact not used by these three simplex tasks.

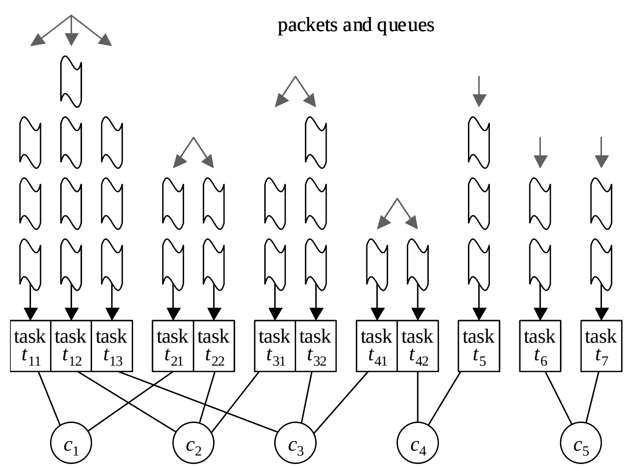

Figure 2. Task allocation and load distribution

There is a queue of packets associated to each replica of each task. We can classify these queues into three different classes according to the number of copies (or criticality) of the tasks. Class 1 consists of queues for the tasks that have only one copy each. In the example above tasks t_5, t_6 and t_7 belong to this class. Class 2 consists of queues for the tasks that are running in duplicate mode. In the example above tasks t_2, t_3 and t_4 belong to this class. Class 3 consists of queues for the tasks that are running in triple redundant mode. In the example above task t_1 is the only task that belongs to this class. We also replicate the packet queues so that each replica has a separate queue. Creating a separate queue for each replica allows that some replicas to be executed faster than their peers (loosely synchronized). When a node becomes faulty, the replicas together with their queues on the faulty node can be re-allocated to another node.

The purpose of the load balance is to balance the number of packets in the queues in the same class. In the example above, the load among tasks t_5, t_6 and t_7 are balanced: Task t_5 has three packets in the queue and t_6 and t_7 have two packets each in their queue. We cannot do better because if we move one packet to another queue, we still have the situation of each queue has two or three packets. The load among tasks t_5, t_6 and t_7 are not balanced: Task t_3 has three packets in its queue while t_4 has only one packet in its queue. We can improve the load balance by moving one packet from the queue of t_3 to the queue of t_4.

3.1 Balanced Task Allocation

We generally assume that there are m tasks $t_1, t_2, ..., t_m$ in the system and the tasks will be allocated onto the n nodes. Each task t_i has k_i redundant copies, $t_{i1}, t_{i2}, ..., t_{ik_i}$, running on different nodes, where $i = 1, 2, ..., m$. In the example in figure 2 we have

$m = 7$, $n = 5$, $k_1 = 3$, $k_2 = 2$, $k_3 = 2$, $k_4 = 2$,

$k_5 = 1$, $k_6 = 1$, and $k_7 = 1$.

And the sequence of task copies is

$T = t_{11}, t_{12}, t_{13}, t_{21}, t_{22}, t_{31}, t_{32}, t_{41}, t_{42}, t_{51}, t_{61}, t_{71}$

Then the total number of copies of all tasks is then

$$K = \sum_{i=1}^{m} k_i \tag{1}$$

The aim of a balanced task allocation among all nodes is to maintain the number of tasks or replicas of tasks, p, running on each node to satisfy the inequalities:

$$p_{\min} \le p \le p_{\max} \tag{2}$$

where

$$p_{\min} = \lfloor K / n \rfloor \tag{3}$$

$$p_{\max} = \lceil K / n \rceil \le p_{\min} + 1 \tag{4}$$

In words, each node should run the average number of tasks. If the total number of tasks is not divisible by the number of nodes n, any nodes can only have one more task than any other nodes.

3.2 Initial Task Allocation Algorithm

Based on the formulas (1-4), we can use the following algorithm to achieve a balanced task allocation. We assume that the task copies are stored in the array T[1..K].

```
Initial-Allocation Algorithm(T[1..K])
  For j = 1 to K
     For i = 1 to n do
        For r = 0 to ⌊K/n⌋
           If j == i+r*n Then
              Allocate T[j] to cᵢ
```

In the algorithm, T is the task sequence in the form

$t_{11}, ..., t_{1k_1}, t_{21}, ..., t_{2k_2}, t_{31}, ..., t_{3k_3}, t_{m1}, ..., t_{mk_m}$

K is the total number of copies of all tasks, and n is the number of computing nodes.

The variable j is the index to the array T. The variable i is the index to the task, and variable r is the index to the copy of the task i. The three loops allocate the task in $T[i]$ to c_i is if $j = i + r*n$. Using this relation, we can reduce the three loops into two loops:

```
     For j = 1 to K
           i = (j – 1)modn + 1
           Allocate T[j] to cᵢ
```

Thus we reduce the complexity of the algorithm from $K*n*\lfloor K/n \rfloor = O(K^2)$ to $O(K)$.

The Initial-Allocation Algorithm guarantees that the task allocation is balanced and the condition in formulas (2-4) is satisfied; allocates the copies of the same task to different nodes if $n \ge p_{\max}$; can be done in $O(K)$ time.

In other words, the algorithm balance the number of task copies among the nodes and if the number of the nodes is greater than or equal to the maximum number of copies of the same task, then, copies of the same task will be allocated on different nodes.

Consider the example in figure 2, the input sequence is

$T = t_{11}, t_{12}, t_{13}, t_{21}, t_{22}, t_{31}, t_{32}, t_{41}, t_{42}, t_{51}, t_{61}, t_{71}$

The balanced task allocation is shown in the table 1 below, where r = 0, 1, and 2 = $\lfloor K/n \rfloor = \lfloor 12/5 \rfloor$.

task	$T[1]=t_{11}$ $T[6]=t_{31}$ $T[11]=t_{61}$	$T[2]=t_{12}$ $T[7]=t_{32}$ $T[12]=t_{71}$	$T[3]=t_{13}$ $T[8]=t_{41}$	$T[4]=t_{21}$ $T[9]=t_{42}$	$T[5]=t_{22}$ $T[10]=t_{51}$
node	c_1	c_2	c_3	c_4	c_5

Table 1. A balanced task allocation

Please note that this allocation is different from the allocation in figure 2. However, both allocations are balanced according to the definition of load balance.

3.3 Dynamic Task Reallocation Algorithm

During the operation a node may fail and the task copies on the failed node will be reallocated to other nodes. The aims of dynamic task reallocation are to: allocate the copies of the same task to different nodes; reallocate tasks on the failed node only; remain a balanced task allocation.

Assume node c is faulty and tasks on node c must be reallocated. The following Dynamic Allocation Algorithm (DAA) tries to reallocate tasks on the faulty node to other nodes while retaining the three conditions.

```
DAA1(T, c)  // T is the set of all tasks
and c is the faulty node
  For each task t on c {
     For i = 1 to n {
        If (c ≠ ci) and
           (ci has no duplicate of t) and
           ((|ci|= pmax - 1) or (pmin = pmax))
        Then
              Allocate t to ci
     }
  }
```

The complexity of the algorithm is O(n).

Let's consider the example in table 1 and assume node c_2 becomes faulty. Then tasks t_{12}, t_{23}, t_{71} need to be reallocated. Following the algorithm DAA1, t_{12} will be reallocated to c_4, t_{32} will be reallocated to c_3, and t_{71} will be reallocated to c_5, as shown in table 2.

task	t_{11} t_{31} t_{61}		t_{13} t_{41} t_{71}	t_{21} t_{42} t_{12}	t_{22} t_{51} t_{32}
node	c_1	$\mathbf{c_2}$	c_3	c_4	c_5

Table 2. Tasks reallocated after c_2 be

In this example, the three conditions (aims) of the task reallocation are met. Unfortunately, this algorithm cannot satisfy all three conditions in all cases. Following is an example where the three conditions cannot be met at the same time.

The initial task allocation is shown in table 3. Assume now node c_6 becomes faulty. According to the algorithm DAA1, tasks t_{23} and t_{52} will be allocated to nodes c_4 and c_5 that have the lowest load. Since a copy of task t_2 is running on both c_4 and c_5, allocating task t_{23} will violate the condition 1).

Generally, we can prove that there is no algorithm that can meet all the three conditions. However, we can develop algorithms to meet any two of the three conditions.

By removing the condition test

```
((|ci|= pmax - 1) or (pmin = pmax))
```

in algorithm DAA1, the algorithm, called DAA1 will guarantee conditions 1) and 2), but not 3).

To guarantee conditions 2) and 3), but not 1), we can use the following DAA2 algorithm:

```
DAA2(T, c)  // T is the set of all tasks
If (pmin = pmax) Then
  Allocate |c| task t on c to the first
  |c| fault-free nodes   // c is faulty
Else
  For each task t on c {
     For i = 1 to n {
        If (c ≠ ci) and (|ci|= pmax - 1)
        Then
           Allocate t to ci
           Exit the inner for-loop
        Else
           If i = n then
              pmax = pmax+1
              reallocate current task t
     }
  }
```

The third possibility is to guarantee conditions 1) and 3), but not 2). The following algorithm implements this compromise.

```
DAA3(T, c)
  call DAA2
  If there is a violation of
     condition 1) by task td
  Then
     For i = 1 to n
        For each task t copy on ci
           If swap(td, t) will remove
              violation
           Then swap(td, t)
```

Although there are two nested loops in the algorithm, the total number of swap operation won't exceed the total number of task replicas, Thus, the complexity of the algorithm is O(K).

task	t_{11} t_{31} t_{61}	t_{12} t_{32} t_{71}	t_{13} t_{41} t_{81}	t_{21} t_{42}	t_{22} t_{51}	t_{23} t_{52}
node	c_1	c_2	c_3	c_4	c_5	c_6

Table 3. Example where condition 1) cannot be met

4. Load Balancing and queuing models

The number of tasks running on a computing node will affect the user's response time from the node because the tasks will share the time slices of the node. Another factor that affects the response time is the jobs (packets to be checked). Having studied the task allocation that balances the number of tasks on each computing nodes, we now turn to discuss the load balancing that manages the packet queue to each task.

As shown in figure 2 and discussed at the beginning in section 3, we have different classes of queues and we want to balance the lengths of queues in the same class.

In this section, we outline the idea of using queuing theory to model the length of the queue and the time spent in the queue and the response time. Assume we have p queues in a given class, q_1, q_2, ..., q_p, the relationship among the components that constitute the response time can be devised as follow.

$\text{Time}_{\text{response}} = \text{Time}_{\text{service}} + \text{Time}_{\text{queue}}$

$\text{Time}_{\text{service}}$ = Average time to service a task

Service rate $\mu = 1 / \text{Time}_{\text{service}}$

$\text{Time}_{\text{queue}}$ = Average time per task in the queue

$\text{Length}_{\text{response}} = \text{Length}_{\text{service}} + \text{Length}_{\text{queue}}$

$\text{Length}_{\text{service}} = \rho_i$, $\text{Length}_{\text{queue}} = (\rho_i)^2 / (1 - \rho_i)$,

$\text{Length}_{\text{response}} = \rho_i / (1 - \rho_i)$.

where $\rho_i = \lambda_i / \mu$.

Assume the packet arrival rate is λ, then according to Little's Law we have

$\text{Length}_{\text{system}} = \lambda \times \text{Time}_{\text{system}}$

We can define the service utilization $\rho = \lambda / \mu$. Then, suppose that the packet arrival rate λ = 10 packets per second and the average time to check a packet is 50ms, the service rate μ = 1 / 50 ms = 20 packets per second. Then

$\rho = \lambda / \mu = 10 / 20 = 0.5$

We assume that

λt_i is the packet arrival rate for one task t_i, $i = 1, \ldots, m$, and

μ is the service rate for one service node c_i, so entire system service rate is $n\mu$.

Following the DAA1 Algorithm, we can consider the system as n different M/M/1 queuing models:

Queue 1 for node c_1: sum(λt_j), where t_j is allocated to c_1, and λ_1 = sum(λt_j). Parameters for this queue is (λ_1, μ).

…

Queue i for node c_i: sum(λt_j), where t_j is allocated to c_i, and λ_i = sum(λt_j). Parameters for this queue is (λ_i, μ).

...

for $i = 1, 2, ..., m$.

The aim of load balancing among the queues in the same class is to maintain the same response time for all jobs (packets) on each queue, i.e. for all q_i and q_j in the same class:

$\text{Time}_{\text{response}} (q_i) = \text{Time}_{\text{response}} (q_j)$

The response time $\text{Time}_{\text{response}}$ can be calculated as follows:

For each queuing model M/M/1 with parameters (λ, μ)

$\text{Time}_{\text{service}} = 1/\mu$

$\text{Time}_{\text{queue}} = \lambda/(\mu(\mu-\lambda))$

Thus, we have

$\text{Time}_{\text{response}} = 1/\mu + \lambda/(\mu(\mu-\lambda))$

We have then

$\text{Time}_{\text{response}} (q_i) = 1/\mu + \lambda_i /(\mu(\mu-\lambda_i))$

We have performed simulation on the response time of packets going through the distributed firewall using three queuing models: (1) basic M/M/1 queuing model, (2) the practical M/M/n queuing model with static balanced task allocation, and (3) the practical M/M/n queuing model with dynamic balanced task allocation. In the rest of the section, we present the three queuing models and simulation results that obtained. In all three models, we used a sample number of K = 100, the total arrival number N = 600, and 10 sets of simulation results to build the average.

Model 1: Basic M/M/1 queuing model

We first use the basic M/M/1 queuing model to evaluate the $\text{Time}_{\text{response}} (q) = 1/\mu + \lambda /(\mu(\mu-\lambda))$ assuming that we only have one computer node to handle the packets. We assumed that the arrival rate λ = 10 packets per second and the service rate μ = 1 / 50 ms = 20 packets per second. The simulation results for $\text{Time}_{\text{response}} (q)$ are given in table 4.

Note: since λ = 10 and μ = 20, the accurate value of $\text{Time}_{\text{response}}$ is 0.1 according to the formula $\text{Time}_{\text{response}} (q) = 1/\mu + \lambda /(\mu(\mu-\lambda))$. As can be seen from the table, the simulation results are very accurate.

index	1	2	3	4	5	6	7	8	9	10	average
time resp	0.0997	0.1000	0.1014	0.0992	0.0992	0.0977	0.1005	0.1002	0.1010	0.1001	0.0999
Related error(%)	0.3	0	1.4	0.8	0.8	2.3	0.5	0.2	1.0	0.1	0.1

Table 4. Simulation value for Time$_{response}$(q) under M/M/1 Model

Model 2: Practical M/M/n queuing model with static balanced task allocation

In this model, we assume that we have n independent M/M/1 models and we use the practical M/M/n model to evaluate the packet response time:

$Time_{response}(q_i) = 1/\mu + \lambda_i /(\mu(\mu-\lambda_i))$

We assume that each single task's arrival rate λ = 10 packets per second, and the service rate μ = 1 / 10 ms = 100 packets per second. The tasks allocation that we use is given in the table 1 in section 3.2. The simulation results of Time$_{response}$ (q_i) for five nodes c_1, c_2, c_3, c_4 and c_5 are summarized in table 5 and the response time for each task t_i is given in table 6, respectively.

node	1	2	3	4	5	6	7	8	9	10	average
c_1	0.0143	0.0142	0.0142	0.0143	0.0145	0.0140	0.0143	0.0143	0.0142	0.0143	0.0143
c_2	0.0144	0.0143	0.0141	0.0144	0.0146	0.0143	0.0141	0.0141	0.0141	0.0145	0.0143
c_3	0.0126	0.0126	0.0126	0.0124	0.0125	0.0126	0.0124	0.0125	0.0124	0.0126	0.0125
c_4	0.0125	0.0126	0.0126	0.0124	0.0125	0.0126	0.0125	0.0124	0.0124	0.0124	0.0125
c_5	0.0125	0.0125	0.0124	0.0125	0.0124	0.0124	0.0125	0.0125	0.0124	0.0123	0.0124
System	0.0133	0.0132	0.0132	0.0132	0.0133	0.0132	0.0132	0.0132	0.0131	0.0132	0.0132
Related error(%)	0.7576	0	0	0	0.7576	0	0	0	0.7576	0	0

Table 5. Simulation results for Time$_{response}$ (q_i) of the five computing nodes under M/M/5 Model

How accurate are these simulation results? Let's see an example.

For λ = 10 and μ = 100, the accurate value of Time$_{response}$ (q_i) = $1/\mu + \lambda_i /(\mu(\mu-\lambda_i))$ is 0.0143 for c_1 and c_2, and 0.0125 for c_3, c_4 and c_5. The difference is 0.0018, or 0.18%, which is accurate.

task	1	2	3	4	5	6	7	8	9	10	average
T1	0.0144	0.0143	0.0142	0.0144	0.0146	0.0143	0.0143	0.0143	0.0142	0.0145	0.0143
T2	0.0125	0.0126	0.0126	0.0125	0.0125	0.0126	0.0125	0.0125	0.0124	0.0124	0.0125
T3	0.0144	0.0143	0.0142	0.0144	0.0146	0.0143	0.0143	0.0143	0.0142	0.0145	0.0143
T4	0.0126	0.0126	0.0126	0.0124	0.0125	0.0126	0.0125	0.0125	0.0124	0.0126	0.0125
T5	0.0125	0.0125	0.0124	0.0125	0.0124	0.0124	0.0125	0.0125	0.0124	0.0123	0.0124
T6	0.0143	0.0142	0.0142	0.0143	0.0145	0.0140	0.0143	0.0143	0.0142	0.0143	0.0143
T7	0.0144	0.0143	0.0141	0.0144	0.0146	0.0143	0.0141	0.0141	0.0141	0.0145	0.0143

Table 6. Simulation results for Time$_{response}$ (t_i) of the seven tasks under M/M/5 Model

Model 3: Practical M/M/n queuing model with dynamic balanced task allocation

We assume that each single task's arrival rate is λ = 10 packets per second and the service rate is μ = 1 / 20 ms = 50 packets per second. The dynamic balanced task allocation that we use is given in the table 2 in section 3.2. The simulation results of Time$_{response}$ (q_i) for four nodes c_1, c_3, c_4 and c_5, as well as the response times for each task t_i are summarized in table 7.

5. Conclusion

In this paper we first briefly introduced the simulation of dependable distributed system based on a prototype we developed in the past a few years and a more sophisticated simulation version of the prototype that we recently developed. The simulation version allows us to development applications with larger number of tasks and computing nodes. We discussed redundant task allocation algorithms and load balancing strategies. We devised an initial task allocation algorithm and three dynamic task allocation algorithms that can balance the task allocation efficiently. We used queuing theory to model the queuing time and queuing length of packets to be processed by redundant firewall tasks. Simulation results based on there queuing models have been obtained to show the relationship between the arrival rates, service time, and the number of parallel queues in the dependable distributed system environment. These experiment results are still preliminary and further experiments are being conducted on the simulation version of the system.

node	1	2	3	4	5	6	7	8	9	10	average
c_1	0.0507	0.0505	0.0501	0.0485	0.0502	0.0492	0.0503	0.0504	0.0508	0.0488	0.0500
c_3	0.0489	0.0492	0.0489	0.0496	0.0485	0.0493	0.0494	0.0500	0.0500	0.0491	0.0493
c_4	0.0504	0.0492	0.0494	0.0490	0.0504	0.0497	0.0504	0.0487	0.0483	0.0498	0.0495
c_5	0.0507	0.0505	0.0501	0.0485	0.0502	0.0492	0.0503	0.0504	0.0508	0.0488	0.0500
System	0.0502	0.0499	0.0496	0.0489	0.0498	0.0493	0.0501	0.0499	0.0500	0.0491	0.0497
Related error(%)	0.4000	0.2000	0.8000	2.2000	0.4000	1.4000	0.2000	0.2000	0	1.8000	0.6000

Table 7. Simulation results for $Time_{response}$ (q_i) of the four computing nodes under M/M/4 Model

References

[1] J. -C. Laprie, "Dependable computing and fault tolerance: Concept and terminology", IEEE 15th Annual int'l symposium on fault-tolerant computing (FTCS-15), Michigan, June 1985, pp. 1 - 11.

[2] N. Bowen, D. Sturman, T. Liu, Towards continuous availability of Internet services through availability domains, the International conference on dependable systems and networks, New York, June 2000, pp. 559 - 566.

[3] Y. Chen, Dependable computing a necessity in the automotive industry, Elektron: Journal of South African IEE, Feb 1999, p.55.

[4] Y. Chen, V. Galpin, S. Hazelhurst, R. Mateer, C. Mueller, Modeling software development of a decentralized virtual service redirector for Internet applications, in Proc. the 7th IEEE Workshop on Future Trends of Distributed Computing Systems, Cape Town, December 1999, pp.235 - 241.

[5] Y. Chen, Z. He, Y. Tian, Efficient Reliability Modeling of the Heterogeneous Autonomous Decentralized Systems, IEICE/IEEE Joint Special Issue on Autonomous Decentralized Systems and Systems' Assurance, IEICE Transactions on Information & Systems, Vol. E84-D, No. 10, October 2001, pp.1360-1367.

[6] Y. 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.

[7] 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.

[8] R. Sinnappan and S. Hazelhurst, A reconfigurable approach to packet filtering, In Proceedings of FPL 2001: 11th International Conference on Field Programmable Logic and Applications, Belfast, United Kingdom, August 2001. pp. 638--642.

[9] Cisco, Overview on the Cisco DistributedDirector, http://www.cisco.com/

[10] Checkpoint Software Technologies, FireWall-1, http:// www.checkpoint.com/