

# SIMULATION

<http://sim.sagepub.com>

---

## The Simulation of a Highly Dependable Distributed Computing Environment

Yinong Chen and Zhongshi He  
*SIMULATION* 2003; 79; 316  
DOI: 10.1177/0037549703037150

The online version of this article can be found at:  
<http://sim.sagepub.com/cgi/content/abstract/79/5-6/316>

---

Published by:

 SAGE Publications

<http://www.sagepublications.com>

On behalf of:



Society for Modeling and Simulation International (SCS)

**Additional services and information for *SIMULATION* can be found at:**

**Email Alerts:** <http://sim.sagepub.com/cgi/alerts>

**Subscriptions:** <http://sim.sagepub.com/subscriptions>

**Reprints:** <http://www.sagepub.com/journalsReprints.nav>

**Permissions:** <http://www.sagepub.com/journalsPermissions.nav>

**Citations** (this article cites 2 articles hosted on the SAGE Journals Online and HighWire Press platforms):  
<http://sim.sagepub.com/cgi/content/refs/79/5-6/316>

# The Simulation of a Highly Dependable Distributed Computing Environment

**Yinong Chen**

Computer Science & Engineering Department  
Arizona State University  
Tempe, AZ 85287-5406, USA  
[yinong@asu.edu](mailto:yinong@asu.edu)

**Zhongshi He**

College of Computer Science  
Chongqing University  
Chongqing 400044, People's Republic of China  
[zshe@cqu.edu.cn](mailto:zshe@cqu.edu.cn)

The aim of the research is to investigate techniques that support the development of highly dependable applications in a distributed system environment. Techniques developed include redundant task allocation, load balancing, fault-tolerant computing, and communication. The application the authors 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 can the packet go through the firewall. Disagreed decision from the different firewall copies signify a possible hardware fault or a software error in the underlying system. This article 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 results 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, and so on. Highly dependable systems are traditionally used only in safety-critical control and monitoring systems such as nuclear reactors, flight control and traffic scheduling, and so forth. 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 (e.g., engine control, antiblock 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 bottlenecks caused by communication, coordinating and synchronizing concurrent behaviors, and needing 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 and limitations 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 in which 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 thousand rules, and each packet going through the firewall has to be checked against the rules [4-6]. A distributed system can improve the performance by running multiple copies of the firewall task in parallel. Cisco's Distributed-Director ([www.cisco.com](http://www.cisco.com)) and Checkpoint Software Technologies' FireWall-1 ([www.checkpoint.com](http://www.checkpoint.com)) use parallel copies of tasks to improve the performance. A dependable distributed system can also improve 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 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 Chen et al. [7]. The reliability modeling was proposed in Chen, He, and Tian [8]. The prototype implementation and the performance measured on the prototype were presented in Chen and Mateer [4]. The implementations of the firewall rulebase based on the system were investigated in Hazelhurst, Attar, and Sinnappan [5] and Sinnappan and Hazelhurst [6]. 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 [4].

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 article. We first briefly outline the simulation system we developed and the experiments and simulation we are conducting on the system.

This article is an extended version of Chen and He [9], with the implementation of the distributed system described in section 2 and new experiments presented in sections 4 and 5.

In the next section, we outline the structure and the main components of the system. Then, we present the simulated distributed system, including its graphic interface, and the status reporting system in section 3. Balanced task allocation algorithms are studied in section 4, including an initial task allocation algorithm and a dynamic task reallocation algorithm. The load-balancing algorithms for packet distribution among the firewall nodes, two queuing models, and simulation results based on the two queuing models are presented in section 5. Section 6 concludes the article.

## 2. System Description

The simulation system we are developing is depicted in Figure 1. The system is written in a Java multithread 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 intranet, respectively). The results from redundant copies of firewalls will be checked by a fault-tolerant protocol.

The function modules in the system are briefly explained as follows. More specific explanation related to the current implementation is given in the following section.

### 2.1 Packet Generation and Rulebase

To simulate Internet applications, transmission control protocol/Internet protocol (TCP/IP) packets with the required formats are generated. The packets are distributed to the firewall tasks. In the current implementation, three groups of firewall tasks are implemented: single mode, double-redundant mode, and triple-redundant mode. The packets are randomly distributed to the three groups. In general, the tasks do not have to be the firewalls; they can be any kind of distributed application. If they are different applications, we have to send different data to different applications. Within each group, multiple (parallel) tasks can be running. For example, we can run two single-mode firewalls, three double-mode firewalls, and one triple-mode firewall (see Fig. 2 in the next section).

Within each group, the tasks are the parallel copies of the same application. We implemented three algorithms to distribute the packets (workload) in each group: random, round-robin, and shortest queue algorithms.

Upon receiving a packet, the firewall will check the packet using its rulebase. The rulebase is a set of complex conditions that define whether a packet should be accepted or rejected. For example, where does a packet come from? Where does the packet go? What is the required operation? Is it an independent packet or a response to a previous operation? A typical rulebase consists of several thousand conditions and is the most time-consuming part of the firewall operation.

### 2.2 Task Allocation and Load Balancing

Two factors affect the response time of the system: task allocation and the load balance. Although we only have

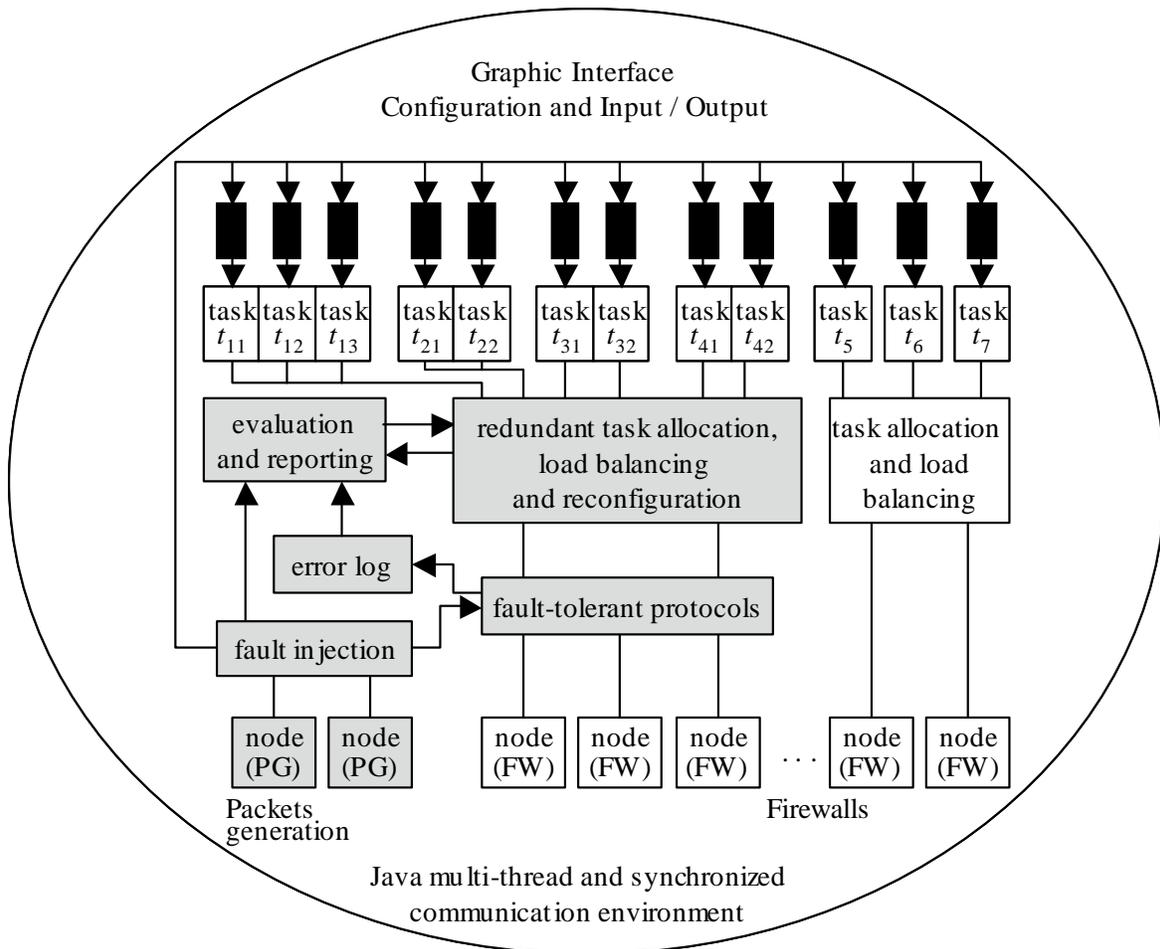


Figure 1. Overview of the simulation system

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 to 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 workload balancing is to keep the waiting time in each queue approximately the same.

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

### 2.3 Fault-Tolerant Protocols and Error Handling

A comparison protocol and a voting protocol are built on 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. We will mark each node with one error tick. A disagreement with the majority in voting indicates a transient error in the node or in its communication links involved. This node will be marked with two error ticks. The accumulation of transient errors indicates possible permanent fault and

reconfiguration requirement. When the number of ticks associated with a node exceeds the given threshold (e.g., 10), the node will be considered faulty.

After a node fault is detected, a reconfiguration will be performed. The reconfiguration is implemented by a task reallocation that excludes the faulty 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 needed to reallocate the task to include new nodes.

Errors recorded by the comparison protocol and voting protocol are also used by the evaluation system to assess the reliability of individual nodes and of the overall system.

#### 2.4 Fault Injection

In a normal operational environment, a fault can only be observed in a relatively long period of time. To accelerate the testing and evaluation process of the fault-tolerant (redundant) execution, we have implemented a random fault injector in the system. The fault injector will reverse the firewall decision (accept or reject) of a replica in the double- or triple-redundant task. No fault will be injected into the single-mode tasks because these tasks cannot detect or tolerate any faults.

Injecting faults at the firewall decision level is a very high-level approach. A more realistic approach is to inject faults in the packets. However, the lower level fault injection will have an unpredictable effect at the firewall decision level. In our current implementation, a high-level fault injection is more appropriate for our testing and evaluation process. The fault injection rate is implemented as a parameter in the graphic interface.

### 3. The Simulated Distributed System

The distributed system described in section 2 is simulated by a program written in Java. This section presents a few snapshots of the system.

Figure 2 shows the input-setting dialog window. Users can define the number of IP packets to be generated and choose one of the three algorithms in each experiment (random, round-robin, and packet queue length-based algorithms), the number of computer nodes, and the number of tasks running in single mode, duplicate mode, and triple-redundant mode. The next text box allows the user to select the fault injection rate. The last text box specifies the name of the firewall rulebase. The user can define multiple rulebases and choose different rulebases in different experiments.

Figure 3 gives two scenarios of the dynamic task allocation on the computer nodes. In the left snapshot, node 0 is faulty, and the tasks are allocated on working nodes 1, 2, 3, and 4. The right snapshot shows a later scenario, in which node 0 is repaired and reintegrated into the system while nodes 2 and 4 become faulty. Now the tasks are reallocated to nodes 0, 1, and 3.

Figure 2. Graphic input interface

Figure 4 shows the dynamic status table and the fault table of the nodes. A node can be in the active (working) status or in the inactive (faulty) status.

If a node is in the active status, replicas of firewall tasks will be allocated to the node according to the task allocation algorithms explained in the next section. The status table also shows the number of packets processed by each node. The fault table on the right-hand side keeps track of the number of faults that occurred in each node (faults injected to the nodes by the fault injector). The threshold is set to 10; that is, if more than 10 faults occurred in a particular node, the node will be considered faulty and set to inactive status. In the table, nodes 1 and 4 have received 12 and 11 faults, respectively, and thus they are in the inactive status.

After a node is set to inactive status, testing will be performed on the faulty node. With a certain probability, a node will pass the test. If a node passes the test, it will be reset to active and reintegrated into the system. The reintegration will cause a task reallocation. If a node fails the test, it will be tested again until it passes the test.

The system will stop if the all packets generated have been processed or if the total number of active nodes drops to an extent that the redundant tasks cannot find a sufficient number of nodes to accommodate their replicas. Replicas of the same tasks must be allocated on different nodes. In the former case, the system will report the final statistics.

Figure 5 shows the final statistic window containing the total number of packets allowed through the firewall, the total amount of time, and the average amount of time. In the experiment showing in the windows, 1615 of 2000 packets have been accepted by the firewalls.

In the following sections, we discuss the details of the dynamic task allocation of the present modeling and the simulation results we have obtained.

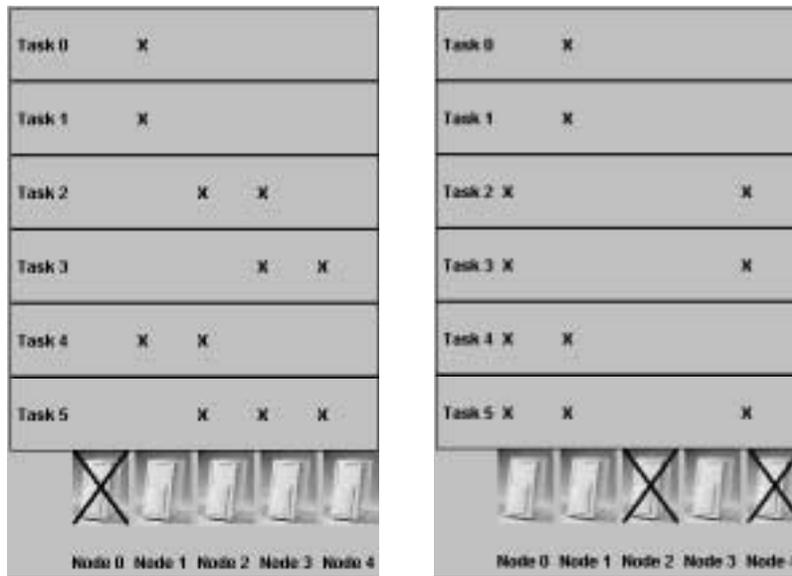


Figure 3. Dynamic task allocation

	Node 0	Node 1	Node 2	Node 3	Node 4	Fault Table
Status:	Active	Inactive	Active	Active	Inactive	Node 0 : 4
IP Packet:	70	61	58	45	21	Node 1 : 12
Tasks:	2:1	--	4:2	0:1	--	Node 2 : 7
	5:3	--	5:2	3:2	--	Node 3 : 4
	4:1	--	2:2	5:1	--	Node 4 : 11
	1:1	--	3:1	--	--	
	--	--	--	--	--	
	--	--	--	--	--	

Figure 4. Node status table and fault table

Simulation Statistics
No. of pkts collected: 1615
No. of secs: 106.003
No. of pkts per sec: 15.235417865532108
Total Response Time: 8000
Avg. Response Time: 0.0040

Figure 5. Final statistics of the experiment above

#### 4. Task Allocation

Figure 6 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  $c_4$ ; and replicas of tasks  $t_6$  and  $t_7$  are running on node  $c_5$ . As we can see, 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 6 is balanced.

Now let us 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

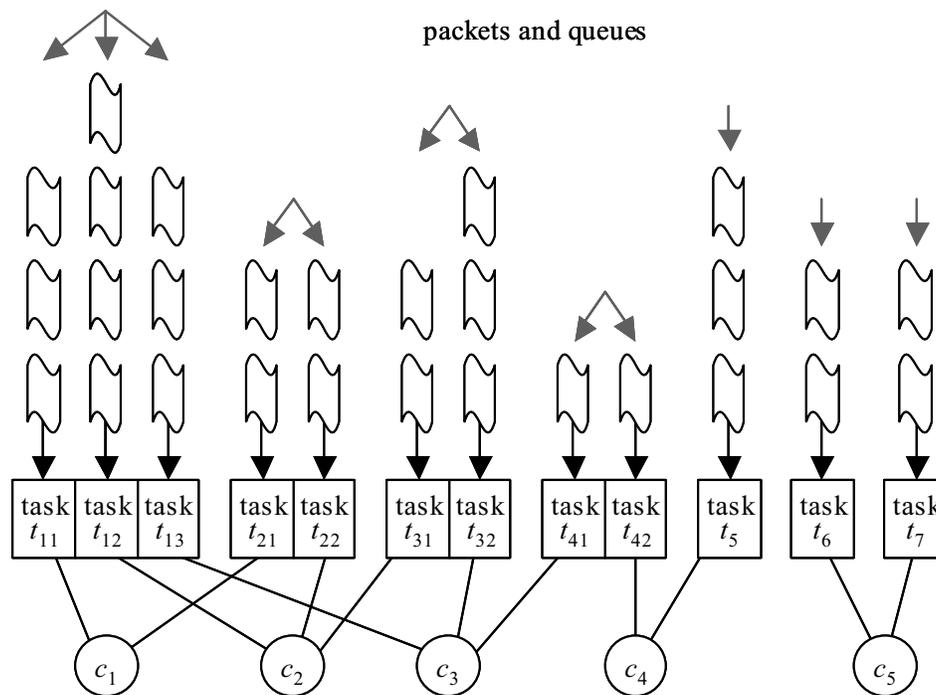


Figure 6. Task allocation and load distribution

$t_1$  will be replicated into three queues and checked by the three replicas of the task, respectively. The results (decisions) will be voted on 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 simple tasks.

There is a queue of packets associated with 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 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 reallocated 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  $t_7$  is 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 having two or three packets. The load among tasks  $t_2$ ,  $t_3$ , and  $t_4$  is 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  $t_4$ .

#### 4.1 Balanced Task Allocation

We generally assume that there are  $m$  tasks  $t_1, t_2, \dots, t_m$  in the system and that the tasks will be allocated onto the  $n$  nodes. Each task  $t_i$  has  $k_i$  redundant copies,  $t_{i1}, t_{i2}, \dots, t_{iki}$ , running on different nodes, where  $i = 1, 2, \dots, m$ . In the example in Figure 6, 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  $k_7 = 1$ .

The sequence of task copies is as follows:

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

Then the total number of copies of all tasks is

$$K = \sum_{i=1}^m k_i. \quad (1)$$

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:

$$p_{\min} \leq p \leq p_{\max}, \quad (2)$$

where

$$p_{\min} = \lfloor K/n \rfloor, \quad (3)$$

$$p_{\max} = \lceil K/n \rceil \leq p_{\min} + 1. \quad (4)$$

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 node can only have one more task than any other node.

#### 4.2 Initial Task Allocation Algorithm

On the basis of equations (1) through (4), we can use the following initial allocation algorithm (IAA) to achieve a balanced task allocation. We assume that the task copies are stored in the array  $T[1..K]$ .

```
IAA(T, n) // T is the task sequence
           // and
           // n is the number of nodes
For j = 1 to K
  For i = 1 to n do
    For r = 0 to  $\lfloor K/n \rfloor$ 
      If  $j == i+r*n$  Then
        Allocate T[j] to  $c_i$ 
      Exit the two inner for-loops
```

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

$$t_{11}, \dots, t_{1k1}, t_{21}, \dots, t_{2k2}, t_{31}, \dots, t_{3k3}, t_{m1}, \dots, t_{mkm}.$$

$K$  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  $c_i$  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) mod n + 1
  Allocate T[j] to  $c_i$ 
```

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

The IAA guarantees that the task allocation is balanced and that the condition in equations (2) through (4) is satisfied, allocates copies of the same task to different nodes if  $n \geq p_{\max}$ , and can be done in  $O(K)$  time.

**Table 1.** The task allocation in Figure 6

Tasks	$t_{11}$ $t_{21}$	$t_{12}$ $t_{22}$ $t_{31}$	$t_{13}$ $t_{32}$ $t_{41}$	$t_{42}$ $t_{51}$	$t_{61}$ $t_{71}$
Node	$c_1$	$c_2$	$c_3$	$c_4$	$c_5$

In other words, the algorithm balances 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 task allocation shown in Table 1. There are five computer nodes and seven tasks. Task 1 has three replicas:  $t_{11}$ ,  $t_{12}$ , and  $t_{13}$ . Tasks 2, 3, and 4 have two replicas each:  $t_{21}$ ,  $t_{22}$ ,  $t_{31}$ ,  $t_{32}$ ,  $t_{41}$ , and  $t_{42}$ . Tasks 5, 6, and 7 have one replica each:  $t_{51}$ ,  $t_{61}$ , and  $t_{71}$ .

Now, we apply algorithm IAA to achieve a balanced task allocation. We assume that input sequence (array) is

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

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

Please note that this allocation is different from the allocation in Figure 6. However, both allocations are balanced according to the definition of load balance.

#### 4.3 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 as follows:

1. allocate the copies of the same task to different nodes,
2. maintain a balanced task allocation,
3. reallocate tasks on the failed node only.

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 (aims).

```
DAA(T, c) // T is the task sequence and
           // c is the fault node
For each task t on c {
  For i = 1 to n {
    If ( $c \neq c_i$ ) and
        ( $c_i$  has no duplicate of t) and
        ( $(|c_i| = p_{\max} - 1)$  or ( $p_{\min} = p_{\max}$ ))
      Then
        Allocate t to  $c_i$ 
        Exit the inner for-loop
  }
}
```

The complexity of the algorithm is  $O(n)$ .

**Table 2.** A balanced task allocation

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

**Table 3.** Tasks reallocated after  $c_2$  ( $c_4$ ?) fault

Task	$t_{11}$ $t_{21}$ $t_{42}$	$t_{12}$ $t_{22}$ $t_{31}$	$t_{13}$ $t_{32}$ $t_{41}$	$t_{61}$ $t_{71}$ $t_{51}$
Node	$c_1$	$c_2$	$c_3$	$c_4$ $c_5$

Let us consider the example in Table 1 and assume node  $c_4$  becomes faulty. Then tasks  $t_{42}$ ,  $t_{51}$  need to be reallocated. Following the algorithm DAA1,  $t_{42}$  will be reallocated to  $c_1$ , and  $t_{51}$  will be reallocated to  $c_5$ , as shown in Table 3.

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

The initial task allocation is shown in Table 4. Assume now that node  $c_6$  becomes faulty. According to the algorithm DAA, tasks  $t_{23}$  and  $t_{52}$  will be allocated to nodes  $c_4$  and  $c_5$ , which 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 condition 1.

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

By removing the condition test

$$((|c_i| = p_{\max} - 1) \text{ or } (p_{\min} = p_{\max}))$$

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

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

```
DAA2(T, c) // T is the task sequence
// and
// c is the fault node
If (p_min = p_max) 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 ≠ c_i) and (|c_i| = p_max - 1)
      Then
        Allocate t to c_i
        Exit the inner for-loop
      Else
```

**Table 4.** Example in which condition 1 cannot be met

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	$c_1$	$c_2$	$c_3$	$c_4$	$c_5$	$c_6$

```
If i = n then
  p_max = p_max + 1
  reallocate current task t
}
}
```

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

```
DAA3(T, c) // T is the task sequence
and
// c is the faulty node
call DAA2(T, c)
If there is a violation of
condition 1) by task t_d
Then
  For i = 1 to n
    For each task t copy on c_i
      If swap(t_d, t) will remove
      violation
      Then swap(t_d, t)
```

Although there are two nested loops in the algorithm, the total number of swap operations will not exceed the total number of task replicas. Thus, the complexity of the algorithm is  $O(K)$ .

## 5. 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 node, we now turn to discuss the load balancing that manages the packet queue to each task.

As shown in Figure 6 and discussed at the beginning of section 3, we have different classes of queues and 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,  $q_1, q_2, \dots, q_p$ ; the relationship between the components that constitutes the response time can be devised as follows:

$$\begin{aligned} \text{Time}_{\text{response}} &= \text{Time}_{\text{service}} + \text{Time}_{\text{queue}} \\ \text{Time}_{\text{service}} &= \text{Average time to service a task} \\ \text{Service rate } \mu &= 1/\text{Time}_{\text{service}} \\ \text{Time}_{\text{queue}} &= \text{Average time per task in the queue} \\ \text{Length}_{\text{response}} &= \text{Length}_{\text{service}} + \text{Length}_{\text{queue}} \\ \text{Length}_{\text{service}} &= \rho_i, \text{Length}_{\text{queue}} = (\rho_i)^2/(1 - \rho_i), \\ \text{Length}_{\text{response}} &= \rho_i/(1 - \rho_i), \end{aligned}$$

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

Assume the packet arrival rate is  $\lambda$ ; then according to Little's law, we have

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

We can define the service utilization as  $\rho = \lambda/\mu$ . Then, suppose that the packet arrival rate  $\lambda = 20$  packets per second, and the average time to check a packet is 10 ms; then, the service rate is  $\mu = 1/10 \text{ ms} = 100$  packets per second. Then

$$\begin{aligned} \rho &= \lambda/\mu = 10/20 = 0.5, \\ \rho &= \lambda/\mu = 20/100 = 0.2. \end{aligned}$$

We assume that  $\lambda t_i$  is the packet arrival rate for one task  $t_i$ ,  $i = 1, \dots, m$ , and  $\mu$  is the service rate for one service node  $c_i$ , so the entire system service rate is  $n\mu$ .

Following the algorithms IAA and DAA, we can consider the system as  $n$  different M/M/1 queuing models:

Queue 1 for node  $c_1$ :  $\text{sum}(\lambda t_j)$ , where  $t_j$  is allocated to  $c_1$ , and  $\lambda_1 = \text{sum}(\lambda t_j)$ . Parameters for this queue are  $(\lambda_1, \mu)$ .

...

Queue  $i$  for node  $c_i$ :  $\text{sum}(\lambda t_j)$ , where  $t_j$  is allocated to  $c_i$ , and  $\lambda_i = \text{sum}(\lambda t_j)$ . Parameters for this queue are  $(\lambda_i, \mu)$ .

...

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

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, that is, for all  $q_i$  and  $q_j$  in the same class:

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

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

For each queuing model M/M/1 with parameters  $(\lambda, \mu)$

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

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

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 a simulation on the response time of packets going through the distributed firewall using two queuing models:

1. the practical M/M/ $n$  queuing model with static-balanced task allocation, in which all nodes are fault free,

2. the practical M/M/ $n$  queuing model with dynamic balanced task allocation, in which node faults and task reallocation are considered.

In the rest of the section, we present the two queuing models and simulation results that were obtained. In the two models, we used a sample number of  $K = 500$ , the total arrival number  $N = 1200$ , and 10 sets of simulation results to build the average.

### 5.1 Model 1: Practical M/M/ $n$ Queue Model for Static Balanced Task Allocation without Faulty Nodes

Now we consider that we have  $n$  independent M/M/1 models, and we use the practical M/M/ $n$  to evaluate the packet response time:

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

We assume that each single task arrival rate is  $\lambda = 20$  packets per second for each task  $T_1, T_2, \dots$ , and  $T_7$ , and the service rate is  $\mu = 1/10 \text{ ms} = 100$  packets per second. The task allocation that we use is given in Table 1. The simulation results of  $\text{Time}_{\text{response}}(q_i)$  for five nodes  $c_1, c_2, c_3, c_4$ , and  $c_5$  is summarized in Table 5.

For the same parameters, the simulation results of  $\text{Time}_{\text{response}}(q_i)$  for tasks  $T_1, T_2, T_3, T_4, T_5, T_6$ , and  $T_7$  are given in Table 6. The columns are numbered 1, 2, ..., and 10, corresponding to 10 experiments.

How accurate are these simulation results? Let us use this example to obtain some idea.

For  $\lambda = 20$  and  $\mu = 100$  packets per second, the accuracy can be expressed by

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

The result is 0.025 for  $c_2$  and  $c_3$  and 0.0167 for  $c_1, c_4$ , and  $c_5$ . The maximum difference between the nodes is 0.004, or 0.4%, which is very accurate.

We also obtained the simulation with variable task arrival rates  $\lambda = 5, 10, 15, 20, 25, 30$ , and 35 packets per second for tasks  $T_1, T_2, \dots$ , and  $T_7$ , respectively, and the service rate  $\mu = 1/10 \text{ ms} = 100$  packets per second. The simulation results of  $\text{Time}_{\text{response}}(q_i)$  for five nodes  $c_1, c_2, c_3, c_4$ , and  $c_5$  are summarized in Table 7.

For the same parameter values, the simulation results of  $\text{Time}_{\text{response}}(q_i)$  for tasks  $T_1, T_2, T_3, T_4, T_5, T_6$ , and  $T_7$  are given in Table 8. These results are consistent with the results with a fixed arrival rate.

### 5.2 Model 2: Practical M/M/ $n$ Queue Model for Dynamic Balanced Task Allocation with a Faulty Node

Now we discuss the dynamic task allocation after a node becomes faulty. We assume that each single task's arrival rate is  $\lambda = 20$  packets per second, and the service rate is  $\mu = 1/10 \text{ ms} = 100$  packets per second.

The dynamic balanced task allocation that we use is given in Table 3, where node  $c_4$  is faulty. The simulation results of  $\text{Time}_{\text{response}}(q_i)$  for the four nodes  $c_1, c_2, c_3$ , and  $c_5$  are given in Table 9. Node  $c_4$  is faulty and thus has

**Table 5.** Time<sub>response</sub> ( $q_i$ ) for five nodes under the M/M/5 model

Node	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$c_1$	0.0166	0.0167	0.0166	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.1920
$c_2$	0.0249	0.0249	0.0250	0.0249	0.0252	0.0251	0.0248	0.0249	0.0250	0.0248	0.0249	0.0250	0.4000
$c_3$	0.0252	0.0251	0.0249	0.0249	0.0248	0.0251	0.0249	0.0250	0.0250	0.0250	0.0250	0.0250	0.2600
$c_4$	0.0167	0.0167	0.0167	0.0167	0.0167	0.0168	0.0166	0.0167	0.0166	0.0167	0.0167	0.0167	0.2580
$c_5$	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.0167	0.1680

**Table 6.** Time<sub>response</sub> ( $t_i$ ) for tasks under the M/M/5 model

Task	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$T_1$	0.0252	0.0251	0.0250	0.0249	0.0252	0.0251	0.0249	0.0250	0.0250	0.0250	0.02504	0.0250	0.1600
$T_2$	0.0249	0.0249	0.0250	0.0249	0.0252	0.0251	0.0248	0.0249	0.0250	0.0248	0.02495	0.0250	0.2000
$T_3$	0.0252	0.0251	0.0250	0.0249	0.0252	0.0251	0.0249	0.0250	0.0250	0.0250	0.02504	0.0250	0.1600
$T_4$	0.0252	0.0251	0.0249	0.0249	0.0248	0.0251	0.0249	0.0250	0.0250	0.0250	0.02499	0.0250	0.0400
$T_5$	0.0167	0.0167	0.0167	0.0167	0.0167	0.0168	0.0166	0.0167	0.0166	0.0167	0.01669	0.0167	0.0599
$T_6$	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0166	0.0167	0.01669	0.0167	0.0599
$T_7$	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0166	0.0167	0.01669	0.0167	0.0599

**Table 7.** Time<sub>response</sub> ( $q_i$ ) for five nodes under the M/M/5 model with a variable arrival rate

Node	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$c_1$	0.0118	0.0118	0.0118	0.0118	0.0118	0.0118	0.0118	0.0118	0.0118	0.0118	0.0118	0.0118	0.0510
$c_2$	0.0143	0.0143	0.0144	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.1330
$c_3$	0.0167	0.0168	0.0166	0.0166	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.0167	0.0167	0.1860
$c_4$	0.0182	0.0181	0.0182	0.0181	0.0182	0.0181	0.0183	0.0182	0.0183	0.0181	0.0182	0.0182	0.2915
$c_5$	0.0286	0.0284	0.0285	0.0289	0.0285	0.0288	0.0288	0.0283	0.0284	0.0283	0.0286	0.0286	0.2415

**Table 8.** Time<sub>response</sub> ( $t_i$ ) for seven tasks under the M/M/5 model with a variable arrival rate

Task	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$T_1$	0.0167	0.0168	0.0166	0.0166	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.01668	0.0167	0.11976
$T_2$	0.0143	0.0143	0.0144	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.01431	0.0143	0.06993
$T_3$	0.0167	0.0168	0.0166	0.0166	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.01668	0.0167	0.11976
$T_4$	0.0182	0.0181	0.0182	0.0181	0.0182	0.0181	0.0183	0.0182	0.0183	0.0181	0.01818	0.0182	0.10989
$T_5$	0.0182	0.0181	0.0182	0.0181	0.0182	0.0181	0.0183	0.0182	0.0183	0.0181	0.01818	0.0182	0.10989
$T_6$	0.0182	0.0181	0.0182	0.0181	0.0182	0.0181	0.0183	0.0182	0.0183	0.0181	0.01818	0.0182	0.10989
$T_7$	0.0182	0.0181	0.0182	0.0181	0.0182	0.0181	0.0183	0.0182	0.0183	0.0181	0.01818	0.0182	0.10989

no data for the node. For the same parameter values, the simulation results of the Time<sub>response</sub> ( $q_i$ ) for the seven tasks  $T_1, T_2, T_3, T_4, T_5, T_6,$  and  $T_7$  are summarized in Table 10.

We also obtained the simulation for variable task arrival rates  $\lambda = 5, 10, 15, 20, 25, 30,$  and  $35$  packets per second for task  $T_1, T_2, \dots,$  and  $T_7,$  respectively, and the service rate  $\mu = 1/10 \text{ ms} = 100$  packets per second. The simulation results of Time<sub>response</sub> ( $q_i$ ) for the four nodes  $c_1, c_2, c_3,$  and  $c_5$  are given in Table 11. Node  $c_4$  is faulty and thus has no data for the node. The simulation results of Time<sub>response</sub> ( $q_i$ ) for the seven tasks  $T_1, T_2, T_3, T_4, T_5, T_6,$  and  $T_7$  are summarized in Table 12. These results are consistent with the results with a fixed arrival rate. Each column numbered 1, 2, ..., and 10 is a separate experiment result, and 10 experiments are summarized in the table.

## 6. Conclusion

In this article, we first briefly introduced the simulation of the dependable distributed system based on a prototype we developed in the past few years, as well as a more sophisticated simulation version of the dependable distributed system that we recently developed. The simulation version allows us to develop applications involving larger numbers of computing nodes and tasks. For the currently implemented firewall application, we can experiment with different packet allocation algorithms, different failure rates, and different rulebases. We discussed redundant task allocation algorithms and load-balancing strategies, including 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

**Table 9.** Time<sub>response</sub> ( $q_i$ ) for five nodes under the M/M/4 model

Node	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$c_1$	0.0249	0.0249	0.0250	0.0250	0.0248	0.0249	0.0250	0.0251	0.0252	0.0250	0.0250	0.0250	0.3600
$c_2$	0.0249	0.0250	0.0251	0.0250	0.0250	0.0248	0.0250	0.0250	0.0250	0.0251	0.0250	0.0250	0.2280
$c_3$	0.0251	0.0251	0.0250	0.0250	0.0248	0.0249	0.0248	0.0248	0.0248	0.0251	0.0249	0.0250	0.4600
$c_4$													
$c_5$	0.0249	0.0249	0.0250	0.0249	0.0249	0.0251	0.0248	0.0249	0.0251	0.0252	0.0250	0.0250	0.3040

**Table 10.** Time<sub>response</sub> ( $t_i$ ) for seven tasks under the M/M/4 model

Task	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$T_1$	0.0251	0.0251	0.0251	0.0250	0.0250	0.0249	0.0250	0.0251	0.0252	0.0251	0.02506	0.0250	0.24
$T_2$	0.0249	0.0250	0.0251	0.0250	0.0250	0.0249	0.0250	0.0251	0.0252	0.0251	0.02503	0.0250	0.12
$T_3$	0.0251	0.0251	0.0251	0.0250	0.0250	0.0249	0.0250	0.0250	0.0250	0.0251	0.02503	0.0250	0.12
$T_4$	0.0251	0.0251	0.0250	0.0250	0.0248	0.0249	0.0250	0.0251	0.0252	0.0251	0.02503	0.0250	0.12
$T_5$	0.0249	0.0249	0.0250	0.0249	0.0249	0.0251	0.0248	0.0249	0.0251	0.0252	0.02497	0.0250	0.12
$T_6$	0.0249	0.0249	0.0250	0.0249	0.0249	0.0251	0.0248	0.0249	0.0251	0.0252	0.02497	0.0250	0.12
$T_7$	0.0249	0.0249	0.0250	0.0249	0.0249	0.0251	0.0248	0.0249	0.0251	0.0252	0.02497	0.0250	0.12

**Table 11.** Time<sub>response</sub> ( $q_i$ ) for five nodes under the M/M/4 model with a variable arrival rate

Node	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$c_1$	0.0154	0.0154	0.0154	0.0154	0.0154	0.0154	0.0155	0.0155	0.0154	0.0155	0.0154	0.0154	0.0065
$c_2$	0.0144	0.0143	0.0143	0.0144	0.0144	0.0143	0.0143	0.0143	0.0143	0.0144	0.0143	0.0143	0.0210
$c_3$	0.0168	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.0167	0.0167	0.1440
$c_4$													
$c_5$	0.0930	0.0917	0.0914	0.0920	0.0909	0.0935	0.0918	0.0960	0.0974	0.0915	0.0929	0.1000	7.1430

**Table 12.** Time<sub>response</sub> ( $t_i$ ) for seven tasks under the M/M/4 model with a variable arrival rate

Task	1	2	3	4	5	6	7	8	9	10	Average	Accuracy	Error (%)
$T_1$	0.0168	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.0167	0.0167	0.3437
$T_2$	0.0154	0.0154	0.0154	0.0154	0.0154	0.0154	0.0155	0.0155	0.0154	0.0155	0.0154	0.0154	0.0909
$T_3$	0.0168	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.0167	0.0167	0.3437
$T_4$	0.0168	0.0167	0.0167	0.0167	0.0167	0.0167	0.0167	0.0166	0.0167	0.0167	0.0167	0.0167	0.3437
$T_5$	0.0930	0.0917	0.0914	0.0920	0.0909	0.0935	0.0918	0.0960	0.0974	0.0915	0.0929	0.1000	7.1430
$T_6$	0.0930	0.0917	0.0914	0.0920	0.0909	0.0935	0.0918	0.0960	0.0974	0.0915	0.0929	0.1000	7.1430
$T_7$	0.0930	0.0917	0.0914	0.0920	0.0909	0.0935	0.0918	0.0960	0.0974	0.0915	0.0929	0.1000	7.1430

queuing time and queuing length of packets to be processed by redundant firewall tasks. Simulation results based on two 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.

This is an ongoing project, and we are working on the different parts of the system. On the implementation side, we are adding new function modules in the system, including different applications other than a firewall. We are also exploring the application of the system in the embedded system area. On the modeling and evaluation side, we are exploring new algorithms that can lead to improved performance of the system.

## 7. Acknowledgments

This work is partly supported by the National Science Foundation of China under grant number 60173060. The distributed simulation system described in section 3 was implemented by Rajanikanth Mitta, Jeff Casimir, Brad Johnson, and Andres Tamayo within their MSc and senior research projects, supervised by Yinong Chen, in the Computer Science and Engineering Department at Arizona State University.

## 8. References

- [1] Laprie, J.-C. 1985. Dependable computing and fault tolerance: Concept and terminology. In *IEEE 15th Annual International Symposium on Fault-Tolerant Computing (FTCS-15)*, June, MI, pp. 1-11.

- [2] Bowen, N., D. Sturman, and T. Liu. 2000. Towards continuous availability of Internet services through availability domains. In *International Conference on Dependable Systems and Networks*, June, New York, pp. 559-66.
- [3] Chen, Y. 1999. Dependable computing a necessity in the automotive industry. *Elektron: Journal of South African IEE*, February, 55.
- [4] Chen, Y., and R. Mateer. 2001. Performance simulation of a dependable distributed system. *SIMULATION* 77 (5-6): 230-7.
- [5] Hazelhurst, S., A. Attar, and R. Sinnappan. 2000. Algorithms for improving the dependability of firewall and filter rule lists. *International Conference on Dependable Systems and Networks*, June, New York, pp. 576-85.
- [6] Sinnappan, R., and S. Hazelhurst. 2001. A reconfigurable approach to packet filtering. In *Proceedings of FPL 2001: 11th International Conference on Field Programmable Logic and Applications*, August, Belfast, UK, pp. 638-42.
- [7] Chen, Y., V. Galpin, S. Hazelhurst, R. Mateer, and C. Mueller. 1999. Modeling software development of a decentralized virtual service redirector for Internet applications. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, December, Cape Town, South Africa, pp. 235-41.
- [8] Chen, Y., Z. He, and Y. Tian. 2001. Efficient reliability modeling of the heterogeneous autonomous decentralized systems. *IEICE Transactions on Information & Systems* E84-D (10): 1360-7.
- [9] Chen, Y., and Z. He. 2003. Simulating highly dependable applications in a distributed computing environment. In *Proceedings of the 36th Annual Simulation Symposium 2003*, April, Orlando, FL, pp. 101-8.

**Yinong Chen** is a lecturer for the Computer Science & Engineering Department at Arizona State University, Tempe.

**Zhongshi He** is a professor in the College of Computer Science at Chongqing University, Chongqing, People's Republic of China.