A Service Scheduler in a Trustworthy System

Yinong Chen

Computer Science & Engineering Department, Arizona State University
Tempe, AZ 85287-8809, USA, yinong@asu.edu

Abstract

The aim of the research is to investigate techniques that support efficient service scheduling algorithms in a service-oriented fault-tolerant real-time distributed system. Techniques we developed include deadline- based real-time scheduling, priority-based scheduling, and redundant resource allocation for fault-tolerance. The system model and scheduling algorithms are designed, and a prototype is implemented to facilitate the investigation and experimentation.

Keywords: Scheduling algorithm, resource allocation, distributed system, fault-tolerant system

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. The dependability attributes include reliability, availability, safety, security, confidentiality, integrity, and maintainability [1]. Different kinds of software and hardware dependable techniques have been developed to produce various kinds of highly dependable systems emphasizing on different depend-ability attributes. For example, a mission-critical flight control system requires the system to have an extreme high reliability in a short period time. Frequent maintenances are necessary to reassure the reliability. A long-life unmanned spacecraft control system must work correctly over a few years without any maintenance, and a telephone exchanging system can accept short-term failure but requires a high availability over a long period of time [2].

Highly dependable techniques are traditionally used in dedicate control and monitoring systems. The recent developments in pervasive computing, embedded systems, database, high-speed networks, wireless communication, and internet have resulted in large-scale distributed systems used for operating the society's critical infrastructures, such as transportation, communication, finance, healthcare, energy distribution, and the combinations of such applications [2-4]. As a result, the consequences of failures are becoming increasingly severe. A trustworthy system is a large- scale distributed, real-time, and dependable system integrating a variety of safety-critical or business-critical applications and emphasizing on all dependability attributes, including reliability, availability, safety, security, confidentiality, integrity, and maintainability [3-4].

The design of large-scale 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 of combinatorial complexity, and the need for a higher degree of dependability and real-time performance pose significant scientific and engineering challenges that are far from being met.

In the past a few years we have developed and implemented a prototype of a dependable distributed system on a local area network [5-9]. The components used were diskless Intel Pentium computers connected by redundant Ethernet network cards. The overall system design was proposed in [5]. The reliability modeling was reported in [6]. The prototype implementation and the performance measured on the prototype were presented in [7]. The implementations of the firewall rulebase based on the system were investigated in [8-9]. Although the prototype gave us realistic data on the dependability and performance of the system, it was complex to use and difficult to add new experiments on the system. On the hardware prototype, we only collected data to evaluate the throughput and reliability of connections between directly connected pairs of computing nodes [7].

We have recently implemented a more sophisticated version of the dependable distributed system and a redundant firewall application using simulation [10]. The simulation system allows us to experiment new algorithms and techniques flexibly and quickly. Load balancing algorithms and their performances under the redundant and parallel task allocation were studied in [10]. This system is outlined in the next section. Since we only implemented a single application, the redundant firewall, the task scheduler was limited to allocate the redundant copies of the firewall to different nodes.

In this paper, we extend the redundant task scheduler

into a full service scheduler with many different types of services, including real-time services, fault-tolerant services, and ordinary prioritized services. The purpose of this extension is to allow our distributed system to simulate large-scale trustworthy applications in the future. The rest of the paper will dedicate to this topic.

In the next section, the structure and the main components of the simulated distributed system that we have developed are outlined. Then, the model of the full service scheduler is presented in section 3. Section 4 elaborates the scheduling algorithms used in the scheduler. Section 5 outlines the architecture and the implementation of a prototype of the service scheduler. Section 6 concludes the paper.

2. Simulation of a Dependable System

The simulation system we have recently developed [10] is depicted in figure 1. Each module in the diagram is a program 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. The incoming packets are generated by the packet generator and approved packets by the firewalls are sent to the packet collector. The packet generator and the packet collector 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 one of the fault-tolerant protocols.

To simulate Internet applications, 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. Within each group, multiple (parallel) tasks can be running. For example, we can run two single mode, three double mode, and one triple mode firewalls. Generally, the tasks do not have to be firewalls. They can be any kind of distributed applications. If they are different applications, we have to send different data to different application. This paper explores this extension by presenting a full service scheduler that handles different types of applications.

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. A typical rulebase consists of several thousands of conditions and is the most time consuming part of the firewall operation.

A comparison protocol and a voting protocol are built on the underlying communication system. It exchanges, compares and votes the output of redundant copies of tasks in double and triple redundant modes, respectively. 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. The node will be marked two error ticks. The accumulation of transient errors indicates possible permanent fault and reconfiguration requirement. When the number of ticks associated to a node exceeds the given threshold, e.g., 10, the node will be considered faulty.

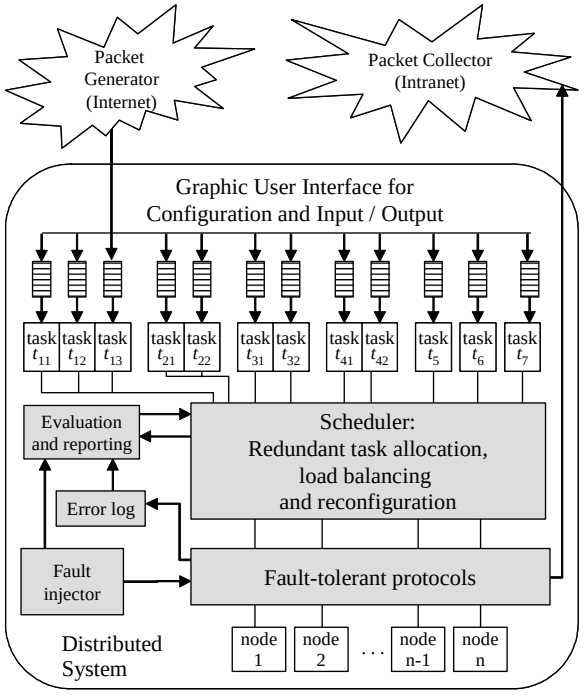


Figure 1. Overview of the simulation system

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 need to reallocate the task to include new nodes.

In the current implementation, we only implemented a single application, the redundant firewall. Thus, the task scheduler was focused on how to allocate the redundant

copies of the firewall to different nodes. In this paper, we will extend the scheduler into a service-oriented scheduler with many different types of services, including real-time services, fault-tolerant services and ordinary prioritized services. The rest of the paper will dedicate to this topic.

3. Design and Modeling of a Service Scheduler

The system we outlined in section 2 can be abstracted as a client-server system, as shown in figure 2. The server consists of three major components: service manager, service scheduler, and service agents. For example, the packet generator is the client that sends request to the packet collector, which is a service agent for certain services. The firewall is a part the service manager that checks the legitimacy of a packet sent to the service agent. When many requests are generated at the same time, we need the scheduler to define the order of the request's processing. In this section, we will discuss the service scheduler in a more generic context without limiting to the applications implemented on our distributed system.

The following is a scenario how the system works, as shown in the sequence numbers in figure 2. (1) A client reads the services published by the service manager and registers for a list of services. The service manager opens an account for the client, adds the service types in its account, and sends back the client an id and a password. (2) Using the user id and the password, the client requests a service. The service manager verifies the client by checking the id, the password, and the service types registered. (3) The service manager forwards the request, if approved, to the service scheduler. The service scheduler puts the request into a queue. (4) The request is scheduled according to its deadline, priority, and available resources. The request is forwarded to the relevant service agent to provide the service requested. (5) The service agents will set up direct communication with the client to complete the service requested. On completion, (6) the service agent will inform the scheduler, and (7) send the accounting information to the service manager.

In general, the service manager publishes services available, registers clients, performs security check, and keeps the accounting information of clients, etc. In our current design, the service scheduler maintains three queues: a simple queue that buffers the approved requests from the service manager, a real-time queue for real-time service requests; and a priority queue for the priority based non-real-time service requests. A request buffered in the simple queue is forwarded to the real-time queue or the priority queue, according the nature the requests. The scheduler also has a dispatcher that selects a service request, reserve resources, and forward it to the service agent according to the deadline or priority. The service agent then acquires the resources and completes the actual services requested.

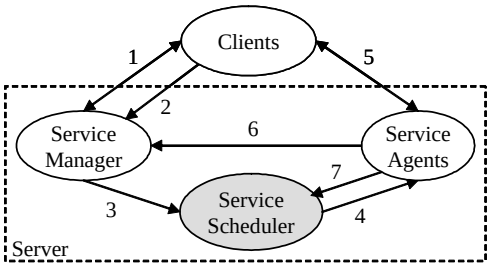


Figure 2. Overview of the client-server

This section specifies and outlines our design of the service scheduler.

3.1 The specification of the service scheduler

The functional specification of the scheduler is defined by the preconditions that the service manager must meet and the postconditions that the service scheduler must meet.

Preconditions

Approved service requests by the service manager are forwarded to the scheduler. The format of the requests is a vector consisting of

- stype: the service code indicating what type of services is being requested;
- dest: the destination identifying the service agent if the name is known to the client. If the destination is not specified, the service manager will allocate the service agent according to the service code;
- sender: the source identifying the client that is requesting the service;
- rset $\in 2^{CRS}$: the requested resource set needed to complete the service.
- dl: deadline, if the request is a real-time request; or
- pri: priority, if the request is a non-real-time request.

The deadline is an integer of greater than or equal to zero that decreases with time. It is used to represent the urgency of predetermined real-time applications. The deadline of a request is initialized by the system according to the nature of the service. The integer decreases with the time and the request must be scheduled before the integer drops to zero. For example, assume that the service is to collect environment information in which some real-time objects are moving and to issue commands to control the next movement of the objects according the collected data. The data must be collected, processed, and delivered in the given time frames. It is assumed that the system is well equipped with dedicate resources to handle the predetermined real-time requests. The general resources

like CPU will be freed soon after the request is started and the dedicate resources, e.g., the video/audio, communication channel, and direct memory access (DMA) coprocessors will execute the service to its completion. For example, if an emergency call request is made, the system must schedule the call, say, within 100 mini seconds. After the emergency call is scheduled, a CPU will be needed for a short period of time only to handle the initial set up. Then the CPU can be freed while the dedicate resources like audio coprocessor and the communication channel continue to serve the request.

The priority is an integer between [0 .. p], $p \geq 0$, which can increase with the time. It is used to represent the urgency of non-real-time applications from client's point of view. The initial priority is set by the client and it could be linked to the price of the service. The priority-based services will be given a quantum of execution. When the quantum expires, the service is put back into the ready queue. To ensure the fairness, the priority of a request increases with the time in the ready queue. That is, the longer a request has waited, the higher the priority will be.

Postconditions

A selected service request is forwarded to the service agent. The format of the request is a vector:

- stype: the service type;
- dest: the destination identifying the service agent;
- sender: the source identifying the client;
- rset $\in 2^{CRS}$: the requested resource set to complete the service.

The selected request R_i must meet conditions:

- $dl(R_i) > 0$ if the task has a deadline, or
- $pri(R_i) \geq \max(pri(R_j))$ for j = 1, 2, ..., m, and $j \geq i$.

3.2 Definition of resource and service mapping

This section will define the scope and the operational procedure of the scheduler.

- There are n types of resources in the system: R_1, R_2, ..., R_n, where $n \geq 1$. For example, the CPUs, coprocessors, shared memory, communication channels, and disks are different types of resources.
- Each resource type R_i has p_i equivalent resources, denoted by R_i^1, R_i^2, ..., R_i^{pi}, where i = 1, 2, ..., n and $p_i >= 1$. For example, if we have 5 type i resources and 3 type j resources, then $p_i = 5$ and $p_j = 3$. Thus, the Complete Resource Set (CRS) can be represented by

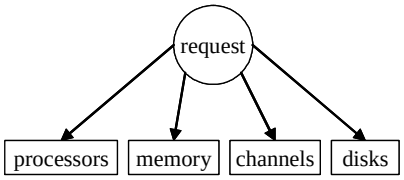
  CRS = {R_1^1, R_1^2, ..., $R_1^{p_1}$, R_2^1, R_2^2, ..., $R_2^{p_2}$, …, R_n^1, R_n^2, ..., $Rn_i^{p_n}$}

  The power set of CRS is represented by 2^{CRS}, which is the set of all subsets of CRS.

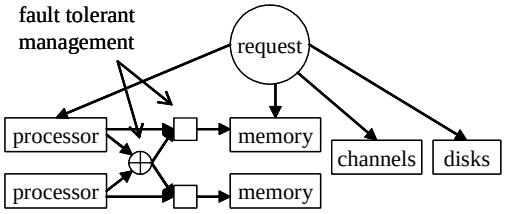
- two sets of service requests RRQ = {rr_1, rr_2, ..., rr_g} and PRQ = {pr_1, pr_2, ..., pr_h}, representing the real-time and priority-based (non-real-time) requests, respectively.
- A service request S_i can request a set of q_i resources {R_{i1}, R_{i2}, ..., R_{iq}}, where $R_{ij} \in$ CRS, or {R_{i1}, R_{i2}, ..., R_{iq}} $\in 2^{CRS}$.
- A service mapping is a function SM: RRQ $\cup$ PRQ $\rightarrow$ 2^{CRS}. The mapping function must meet the postconditions defined in section 3.1.

3.3 Redundant resource scheduling

To support the high reliability requirement and fault tolerant computing, the scheduler may schedule redundant resources to gain extra reliability for certain services. Figure 3 shows an example where a service request can be met functionally with the basic resource allocation. The redundant resource allocation allows the service to be processed simultaneously by two sets of resources, e.g., use two processors to compute the results and stored the results in two different memory locations.



(a) Basic resource allocation



(b) Redundant resource allocation

Figure 3. Basic and redundant resource allocation

In figure 3, the fault tolerant management part shows that two processors are allocated to perform duplicate execution of a critical service. The duplicate results are compared by the comparison protocol and the results are written redundant memory locations if the comparison produces an agreement.

The redundant task allocation algorithms studied in [10] guarantee to allocate replicas of the same task on different computing nodes. In the service scheduler presented in this paper, this requirement is guaranteed, because different CPUs and other resources are considered as separate resources and a resource can only be allocated once to any task. In other words, the same resource cannot be allocated to the replicas of the redundant tasks.

4. Service Scheduling Algorithms

The inputs of the scheduling algorithm are two sets of service requests RRQ = {rr_1, rr_2, ..., rr_g} and PRQ = {pr_1, pr_2, ..., pr_h}, representing the real-time and priority-based requests, respectively, and the set of available resources CRS = {R_1^1, R_1^2, ..., $R_1^{p_1}$, R_2^1, R_2^2, ..., $R_2^{p_2}$, …, R_n^1, R_n^2, ..., $Rn_i^{p_n}$}.

The parameter list of each service request S_i is (stype, sender, des, dl, pri, rset), where

- stype: the type of service.
- sender: sender id;
- dest: the service provider's id;
- dl: the deadline.
- pri: the priority.
- rset $\in 2^{CRS}$: the requested resource set to complete the service.

In this section, several scheduling algorithms are defined, emphasizing different criteria of optimization.

4.1 Deadline-first scheduling

The deadline is the most important criterion that must be satisfied. This algorithm first considers the request whose deadline value is the lowest.

Three states are considered: running, ready, and blocked, as showing in figure 4. Initially, all requests are in the ready state and all resources are available. Then the scheduling process enters into a loop of allocating and de-allocating resources.

In each iteration of the loop, if there are real-time requests, the request with the lowest deadline is selected for dispatching. If the resources needed by this request are available, the request is then dispatched into the running state and the resources allocated to the running requests are subtracted from the available resource set. Otherwise, the priority-based services that is being executed and that have the resources needed by the real-time request will be preempted and their resource released to ensure the execution of the real-time request.

If no real-time requests in the queue, the scheduler will schedule the non-real-time requests according to their priorities. The request with the highest priority will be selected for dispatching. If the resources needed by this request are available, the request is then dispatched into the running state, and the resources allocated to the running requests are subtracted from the available resource set. Otherwise, the request is moved into the blocked state. When a priority-based request is dispatched, the quantum timer will be started so that a long service would not occupy the resources for too long. When the quantum of the request expires, the request is moved from running state back into the ready state, the resources allocated to this request is released, and the requests that are waiting for the released resources are moved from the block state to the ready state. During the execution of a non-real-time service, the resources could be preempted and the serving request is put into the blocked state if a real-time request is dispatched.

Dynamic set data structures are used to hold the requests in the ready, running and blocked states. The ready state consists of the real-time queue and the priority-based queue. They are implemented by two heap-based priority queues that have a GetMinDeadline method that returns the request with the lowest deadline and a GetMinDeadline method that returns the request with the highest priority, respectively.

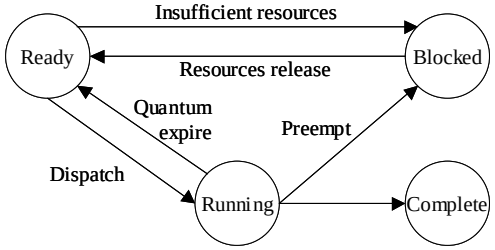


Figure 4. States of the requests being executed

These analyses lead to the following resource scheduling Algorithm 1.

Algorithm 1

```
Input
  RRQ = {rr1, rr2, ..., rrg};
  PRQ = {pr1, pr2, ..., prh},
  // RRQ ∪ PRQ forms the Ready set
  CRS = {R1^1, R1^2, ..., R1^p1, R2^1, R2^2, ...,
         R2^p2, …, Rn^1, Rn^2, ..., Rni^pn}
Resource := CRS;
Running := {};
Blocked := {};
While (True) Do
  If RRQ ≠ empty
     RealTimeSchedule(RRQ, Resource);
  Else
     PrioritySchedule(PRQ, Resource);
Endwhile
Subroutine RealTimeSchedule
                    (RRQ, Resource);
  MinS := GetMinDeadLine(RRQ);
  // return request with lowest deadline
  If rset(MinS) ∈ 2^Resource Then
     Resource := Resource – rset(MinS);
     Ready := RRQ - MinS;
     Dispatch MinS;
```

```
  Else
     Preempt the priority-based services
EndSubroutine RealTimeSchedule;
Subroutine PrioritySchedule
                    (PRQ, Resource);
  MaxS := GetMaxPriority(PRQ);
  // return request of highest priority
  If rset(MaxS) ∈ 2^Resource Then
     Resource := Resource – rset(MaxS);
     PRQ := PRQ - MaxS;
     Dispatch MaxS;
     Start timer(quantum)
                    for this request;
     Increase the priority of all
                    requests in PRQ;
  Else
     Blocked := Blocked ∪ {MaxS};
  If the quantum of the running request S
times up;
     PRQ := PRQ ∪ S;
     Resource := Resource ∪ rset(S);
EndSubroutine PrioritySchedule.
```

4.2 Deadline and Priority Scheduling

Algorithm 1 simply schedules all real-time requests first and then schedules the priority-based requests. Obviously, this algorithm can guarantee the deadlines of the requests if they can be guaranteed at all. However, if there are many real-time requests, the priority-based requests may be significantly relayed or not be executed at all, even if the deadlines of some requests are not very tight. Algorithm 2 below tries to address this problem by executing priority-based requests between the real-time requests, as long the execution does not result in the deadline misses of real-time requests. The two subroutines `RealTime-Schedule` and `PrioritySchedule` used in algorithms 2 are the same subroutines used in algorithm 1.

Algorithm 2

```
Input
  RRQ = {rr1, rr2, ..., rrg};
  PRQ = {pr1, pr2, ..., prh};
  // RRQ ∪ PRQ forms the Ready set
  CRS = {R1^1, R1^2, ..., R1^p1, R2^1, R2^2, ...,
            R2^p2, …, Rn^1, Rn^2, ..., Rni^pn}
Resource := CRS;
Running := {}; Blocked := {};
While (True) Do
  If RRQ ≠ empty
     MinS := GetMinDeadLine(RRQ);
     If dl(minS) ≤ quantum
        RealTimeSchedule( RRQ,
                          Resource);
     Else
        PrioritySchedule(PRQ,
                          Resource);
  Else
     PrioritySchedule(PRQ, Resource);
Endwhile.
```

The correctness of Algorithm 2 is based on the assumption that the real-time requests on the system are deterministic and the deadlines of requests can be met if a request with lower deadline can be scheduled before it deadline. More realistic models of real-time applications are being studied and the service scheduler will be extended based the new type of services.

5. A Prototype of the Service Scheduler

This section outlines the design and the implementation of a prototype of the service scheduler.

5.1 Design of the prototype

A prototype of the service scheduler is implemented using Microsoft .Net framework and C#. To test the service scheduler, we implemented a very simple client node and a few simple service agents on the server. The client in the current implementation is a program that continuously generates different kinds of requests in required formats. The client resides on one computer and the server resides on another computer. The client and server are connected through the Internet, as shown in figure 5.

The requests are encoded into character strings and sent to the Server through the TCP/IP protocol. A different group is implementing the clients, service manager and service agents. This prototype mainly implements the service scheduler. Each component in the service scheduler is implemented by a thread or a group of threads. The requests are buffered in the Input Buffer Queue (IBQ). The Distributor reads the requests in IBQ. If a request is a priority-based non-real-time request, it is added into the Priority Request Queue (PRQ). If a request is a real-time request, the deadline is computed and the request added into the Real-time Request Queue (RRQ). RRQ and PRQ represent the Ready state. The Dispatcher that implements the Algorithm 2 discussed in section 4 is the core component of the scheduler. Requests in the ready state (in one of the queues) are selected and dispatched into the running state. Multiple requests can be processed at the same time, depending on the available resources. The real-time requests will be processed to completion once they are dispatched. On the other hand, a non-real-time request will go back to the Ready state if its quantum expires or it will go to the Blocked state if it is preempted due to scheduling of a real-time request that needs the resources held by the non-real-time service.

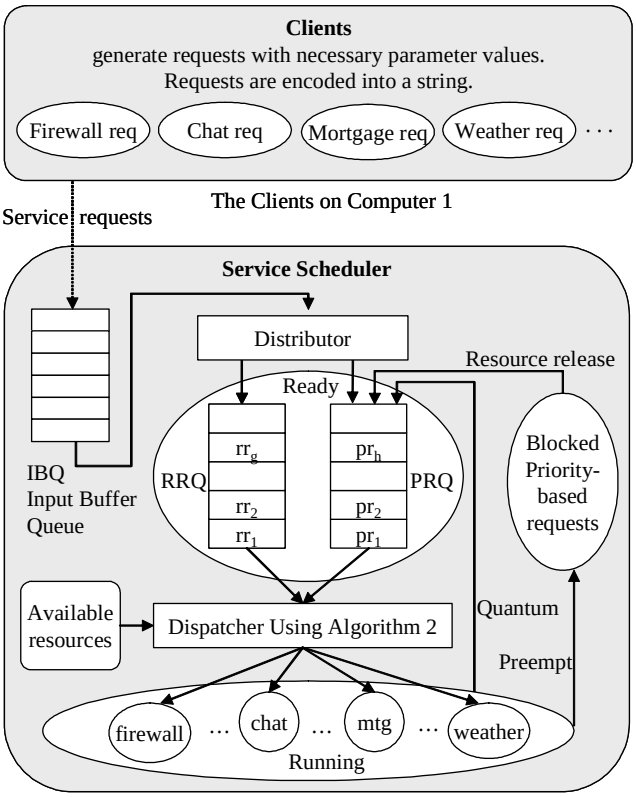


Figure 5. The prototype on two computers

The Input Buffer Queue (IBQ) is a simple first-in-first-out queue with one member of string type. It is implemented as an array of strings with a front pointer and a rear pointer.

The Priority Request Queue (PRQ) is a standard priority queue using the user specified priority as the key. A GetMaxPriority() method is used to dequeue the request with the highest priority value. The heap data structure is used to support the efficient execution of the GetMaxPriority() operations. Since an incomplete request can be sent back to the PRQ, the queue is a field to store the breakpoint information so that the request can be processed from the breakpoint when it is dispatched next time.

The Real-time Request Queue is a standard priority queue using the deadline as the key. A GetMinDeadline() method is used to dequeue the request with the minimum deadline value and heap data structure is used to support the GetMinDeadline() operations. In the implementation, the deadlines of requests are not decreased with the clock. Instead, a RemainingTime() method is used to compute the remaining time to serve a request according the initial value of its deadline, the time when the request is added into the RRQ, and the current time.

5.2 The graphic interface to the prototype

To demonstrate our service scheduler, we have implemented a graphic interface and a few simple applications: a firewall application as described in section 2, a chat room service that can open multiple windows for different chat topics, a simple mortgage calculator, and a simple weather service. We consider the firewall and chat room applications require real-time response while the other two services do not require real-time response. More sophisticated clients, service manager, and service agents are being developed by other groups in our research project. Figure 6 shows the registration window of the service manager. A client must register to the service manager and select the desired services before it can request the services. The service manager then creates an account for the user and keeps accounting information of the user. The accounting information includes user names, user login id, password, registered services, and the lengths of accessing services. After a client registered to the server, a client can request a service that it has registered using its user name and password.

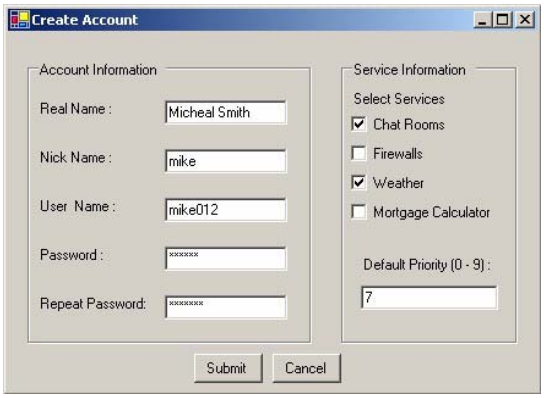


Figure 6. Registration GUI on each client's machine

Figure 7 shows a scenario of the chat application. The GUI shows the nick names of all users who have entered the chat room and the text each user has sent to the chat room.

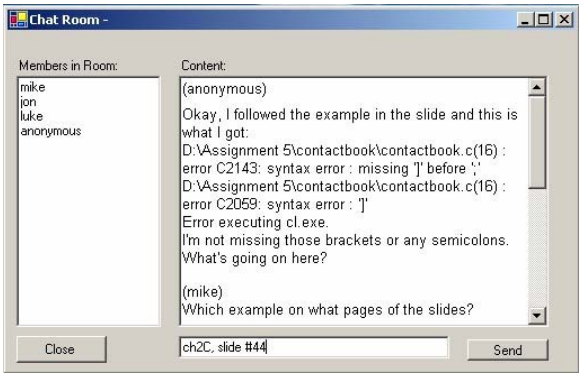


Figure 7. Chat room GUI on each client's machine

Figure 8 is the administration GUI on the sever machine. It dynamically shows the status of the input buffer queue IBQ, the real-time request queue RRQ, and the priority-based request queue PRQ. It approximately illustrates the percentage of the fullness of each queue. On the left-hand side of figure 8, the services currently available are listed and briefly explained of its input requirements.

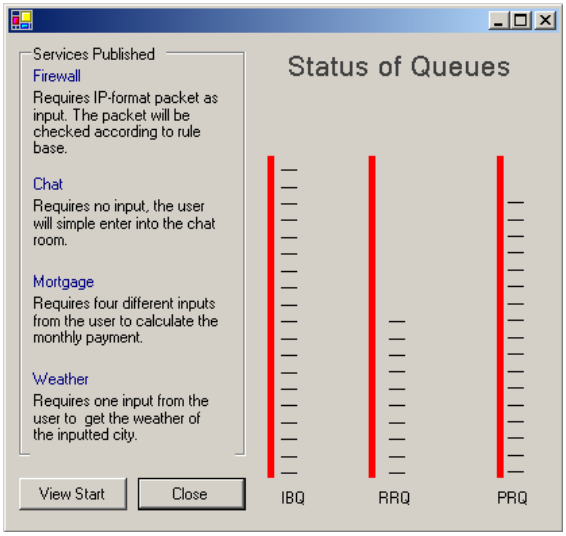


Figure 8. Queue status GUI on the server

6 Conclusions and Future Work

In this paper, we first briefly introduced a simulated dependable distributed system based on a prototype we developed recently. A single application and a simple scheduler were implemented in the system. In this paper, we extended the simple scheduler to a full service scheduler that could schedule different types of services, including ordinary priority-based services, real-time services, and fault-tolerant services. With the addition of the service scheduler, the dependable distributed system could be used to develop a service-oriented trustworthy system. This work is a part of a large project aimed at developing a full service-oriented trustworthy system, including various client applications, service management, and service agents. The scheduler will be integrated into the underlying operating system to provide the resource and service scheduling. Further fault-tolerant mechanisms such as automatic checkpointing and recovering will be implemented.

Acknowledgement

The requirement of a full service scheduler and the ideas of developing such a service scheduler come from numerous discussions with the colleagues Yann-Hang Lee, Kyung Ryu, Wei-Tek Tsai, and Stephen Yau, in the Computer Science and Engineering Department at the Arizona State University. My students Manvendu Bharadwaj, Christopher Boone, Michael Covarrubias, Raquel Pena, and Jake Schwartz contributed to the implementation of the service scheduler and the sample applications.

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), Ann Arbor, Michigan, June 1985, pp. 1 - 11.

[2] D.P. Siewiorek and R.S. Swarz, Reliable Computer Systems: Design and Evaluation, third edition, A K Peters, 1998.

[3] 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.

[4] F. Schneider, S. Bellovin, A. Inouye, Building Trustworthy Systems: Lessons from the PTN and Internet, IEEE Internet Computing, November-December 1999, pp.53 - 61.

[5] 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.

[6] Y. Chen, Z. He, Y. Tian, Efficient Reliability Modeling of the Heterogeneous Autonomous Decentralized Systems, IEICE Transactions on Information & Systems, Vol. E84-D, No. 10, October 2001, pp.1360 - 1367.

[7] 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., Vol.77, No.5 - 6, November/December 2001, pp.230 - 237.

[8] 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.

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

[10] Y. Chen, Z. He, The Simulation of a Highly Dependable Distributed Computing Environment, Simulation, Trans. of the Society for Modeling and Simulation International, Vol.79, No. 5-6, May – June, 2003, pp.316-327.