

Scheduling Simulation in a Distributed Wireless Embedded System

Yinong Chen, Hai Huang, Wei-Tek Tsai

Computer Science & Engineering Department, Arizona State University
Tempe, AZ 85287-8809, USA

Abstract

The aims of the research are to develop a distributed simulation environment and to investigate techniques that support efficient task scheduling algorithms in fault-tolerant, real-time, distributed, and wireless embedded systems. Techniques we developed include deadline-based real-time scheduling, priority-based scheduling, redundant resource allocation for fault-tolerance, energy-aware, and signal-strength-aware scheduling. A system model is proposed and a prototype of the distributed simulation environment is implemented. Three scheduling algorithms and sample applications are designed and implemented in the environment. Experiment data are collected for analyzing the effectiveness of the algorithms.

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

1. Introduction

A dependable computer system is often associated with fault-tolerant, real-time, and distributed computing. With the development of wireless and embedded technologies, dependable systems are becoming more remotely reconfigurable and distributed. 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 dependability 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 dedicated 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][3][4].

The design of large-scale distributed systems is known to be challenging 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 [5][6][7]. Simulation is a powerful tool used to model, design, and experiment such complex systems [8][9]. Due to the complexity of the problems, task scheduling and resource allocation are often studied separately [10][11][12].

In the past a few years we have developed and implemented a prototype of a dependable distributed system on a local area network [13][14][15][16][17]. The components used were diskless Intel Pentium computers connected by redundant Ethernet network cards. The overall system design was proposed in [13]. The reliability modeling was reported in [14]. The prototype implementation and the performance measured on the prototype were presented in [15]. The implementations of the firewall rule-base based on the system were investigated in [15][16]. 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 collected data to evaluate the throughput and reliability of connections between directly connected pairs of computing nodes [15].

We have recently implemented a sophisticated version of the dependable distributed system and a redundant firewall application using simulation [18]. The simulation system allows experimentation of new algorithms and

techniques flexibly and quickly. Load balancing algorithms and their performances under the redundant and parallel task allocation were studied in [18][19]. This system is outlined in the next section.

In this paper, we extend the simulation environment to cover both fixed and wireless network. The new task scheduler can schedule different types of tasks and allocate resources at the same time, including real-time tasks, fault-tolerant tasks, energy-aware tasks, signal strength-aware tasks and ordinary prioritized services. The purpose of this extension is to allow our distributed system to simulate large-scale distributed applications with wireless communication links 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 developed are outlined. Then, the model of the task 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 task scheduler. Section 6 presents the experiment results. Section 7 concludes this paper.

2. The Simulation Framework

The simulation system developed [18] is depicted in figure 1. It can simulate a wired distributed system, a wireless network, and the combination of wired and wireless networks. Different applications can be simulated on the system. Consider a remotely controlled embedded system. The remote task center can issue tasks and send to the control center, which then dispatches the tasks to the task executors. In this application, the control center may either receive the tasks from the remote task center or have all tasks pre-stored. The task executor could be complex system like a vehicle and a plane, or a simple system consisting of a sensor and an actuator only. In another application, the system can simulate a client-server system. The remote task center consists of individual clients. The control center plays the role of the server or service broker. The executors are the service agents that actually perform the services requested by the clients.

The tasks are assigned resources, including processors and memory in the control center and executors outside the control center. An executor is mobile node with limited computing capacity, sensors to collect information, actuators to take actions, and wireless link to the control center. In the simulation, each module is a program thread or a group of threads. A graphic interface is used to configure the system by assigning the number of executors, the number of tasks and the number of replicas of each task. It also displays the states of the system including the working and failed components, the tasks in task queues,

the replicas on each task, the residual energy of each mobile node, the signal strength of each wireless link from each node, and the experiment data measured. In the current simulation system, we implemented the graphic interface, the remote task center, the control center, and a number of applications, including a quarterback calculator, graphic rooms with controllable characters moving between the rooms, and a firewall application with redundant copies executed in multiple executors. To test the firewall, a packet generator in the task center generates sequence of packets and the packets are sent to the redundant firewalls running on a set of executors. The results from redundant copies of firewalls will be checked by one of the fault-tolerant protocols in the control center. The packets passing the firewall check will be sent back to a packet collector in the task center.

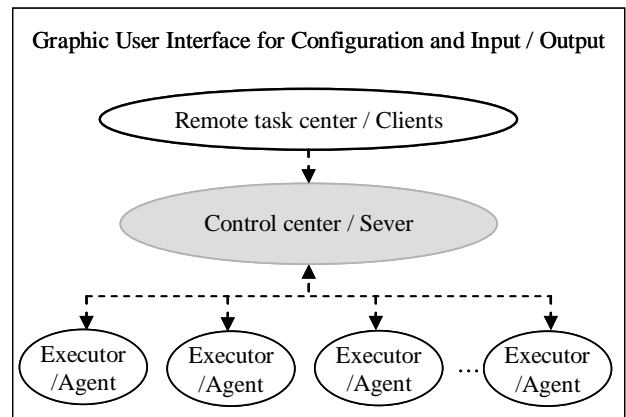


Figure 1. Overview of the simulation system

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 rule-base. The rule-base is a set of complex conditions that define whether a packet should be accepted or rejected. A typical rule-base consists of several thousands of conditions and is the most time and energy consuming part of the firewall operation.

A comparison protocol and a voting protocol are in the

distributed processors in the control center. It exchanges, compares and votes the output of redundant copies of tasks in double and triple redundant modes, respectively, in the mobile executors. A disagreement in comparison indicates a transient error in one of the computing nodes or wireless 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 wireless 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 an executor exceeds the given threshold, e.g., 10, the executor will be considered faulty. If the residual energy of an executor drops below a predefined energy threshold, or the accumulated signal strength of a wireless node drops below a predefined signal strength threshold, the node will be considered faulty as well.

After a fault in an executor is detected, a reconfiguration will be performed. The reconfiguration is implemented by a task reallocation that excludes the faulty executors from participating in executing the tasks, or detects a new communication route to the executor. Workloads need to be rebalanced among surviving nodes. Repaired, replaced, recharged, or signal-regained 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 implemented one redundant application, the firewall. The other applications run one copy only on one executor. Thus, the redundant task scheduler was focused on how to allocate the redundant copies of the firewall to different executors. Different requirements will be considered, including real-time, fault-tolerant, energy-aware, signal strength-aware, and ordinary prioritized requirements.

3. Design and Modeling of Task Scheduler

The scheduler in the simulation system outlined in figure 1 is elaborated in figure 2. A typical scenario of the task execution is labeled by the numbers on the arcs in the figure. The scenario is explained as follows:

1. The task center sends a sequence of tasks to the task manager, which performs security check.
2. The tasks passing the check are sent to the task scheduler.
3. The task scheduler allocates computing resources (CPU, memory, etc.) to perform necessary preparation, according to their deadlines and priorities.
4. The task scheduler allocates the executor to perform the required tasks.
5. The executors send their results back to the computing

resource to complete the computation.

6. The computing resources send the results back to the task manger.
7. The task manger verifies the results and sends them back to the task center.

The task manager also collects status information of the executors, including the remaining energy and signal strength. This paper will focus on the task scheduler.

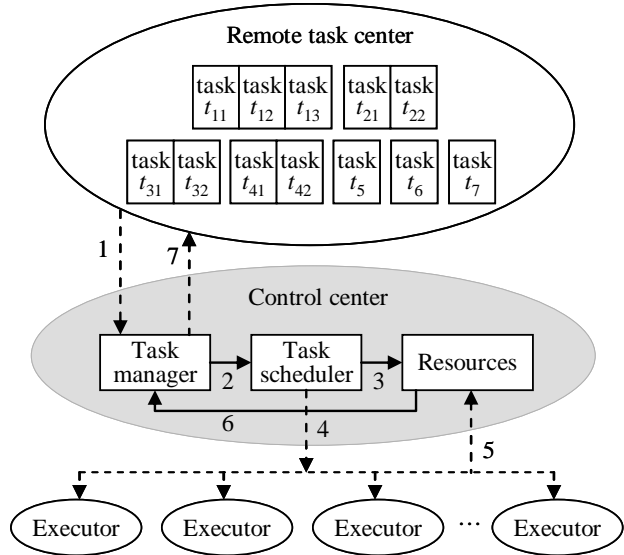


Figure 2. Components of the scheduler

3.1 Definition of resource and task mapping

This section will define the scope and the operational procedure of the scheduler. Different from traditional scheduler that separates resource allocation from task scheduling, our scheduler considers processors (executors) as resources too and thus combines task scheduling and resource problems, defined as follows.

- There are n types of resources in the system: R_1, R_2, \dots, R_n , where $n \geq 1$. For example, the processors, shared memory, communication channels, and executors are examples of different types of resources. Executors are special types of resources that are connected via wireless network that thus two extra factors need to be considered: the residual energy and the signal strength. Here we assume all resources are releasable. We assume there are sufficient un-releasable resources to accommodate all tasks. This assumption is valid because tasks in a real-time system must be predictable and the resources must be sufficient to guarantee the deadlines if the scheduler properly schedules the tasks.
- Each resource type R_i has p_i equivalent resources, denoted by $R_i^1, R_i^2, \dots, R_i^{p_i}$, where $i = 1, 2, \dots, n$, and $p_i \geq 1$. 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

$$\text{CRS} = \{R_1^1, R_1^2, \dots, R_1^{p_1}, R_2^1, R_2^2, \dots, R_2^{p_2}, \dots, R_n^1, R_n^2, \dots, R_n^{p_n}\}$$

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

- There are two sets of task requests $\text{RTQ} = \{rt_1, rt_2, \dots, rt_g\}$ and $\text{PTQ} = \{pt_1, pt_2, \dots, pt_h\}$, representing the real-time and priority-based (non-real-time) requests, respectively.
- A task request S_i can request a set of q_i resources $\{R_{i1}, R_{i2}, \dots, R_{iq_i}\}$, where $R_{ij} \in \text{CRS}$, and $\{R_{i1}, R_{i2}, \dots, R_{iq_i}\} \in 2^{\text{CRS}}$.
- A task mapping is a function $\text{TM}: \text{RTQ} \cup \text{PTQ} \rightarrow 2^{\text{CRS}}$. The mapping function must meet the preconditions and postconditions defined in section 3.2.
- Each executor N_i , $1 \leq i \leq m$ is associated with a dynamically measured metrics, $\text{eng}(N_i)$ to represent the current residual energy. For fixed node, $\text{eng}(N_i) \equiv \infty$.
- Each executor N_i , $1 \leq i \leq m$ is associated with a dynamically measured metrics, $\text{sigstreng}(N_i)$ to represent the current accumulated signal strength. For fixed node, $\text{sigstreng}(N_i) \equiv \infty$.

We measure only the residual energy and accumulated signal strength for wireless nodes. In particular, all modules, the task initiator, the task manager, the task scheduler, and the task executors, can be wireless nodes. For the purpose of simplicity, we consider the possibility that executors are wireless nodes only. We justify the simplification as follows. The task manager and the service scheduler are critical components of the entire system, so it is highly desirable to have fixed nodes carrying them. If in some extreme circumstances, the task manager and the service scheduler are carried by wireless nodes, it is still feasible to use them as origin when measuring signal strength of other nodes. That is, the signal strength of a mobile node is high if we can observe high signal strength to that task executor from either task manager or task scheduler (step 4 and 7 in figure 2). If multiple hops are involved within the route from the task manager or task scheduler to a task executor, then the minimum signal strength along the route is defined to be the *accumulated signal strength* of the task executor. Since the precondition of a client to be assigned to a task is that the client has enough signal strength to reach the task manager and hence task scheduler (step 1 and 2 in figure 2), the signal strength of clients can be ignored. That is, client has little signal strength is simply invisible from the task manager and hence the task scheduler. If both the client and the task executor have sufficient signal strength (the signal strength of the task executor is lower bounded by the accumulated

signal strength), then it is safe to assume that they can communicate directly without difficulty (step 5 in figure 2).

3.2 The specification of the task scheduler

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

Preconditions

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

- tid : task id that uniquely identifies the task to be executed;
- sid : sender id that uniquely identifies the requestor that initiated the request. The execution results will be sent back with the sender id and the task id.
- $\text{rset} \in 2^{\text{CRS}}$: the requested resource set needed to complete the task.
- dl : deadline, if the task is a real-time task; or
- pri : priority, if the task is a non-real-time task.
- eng (optional): the requirement on the minimum residual energy.
- sigstreng (optional): the requirement on the minimum accumulated signal strength.

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 task. The integer decreases with the time and the request must be scheduled before the integer drops to zero. For example, assume the system in figure 2 simulates a robot soccer game. Each executor is a robot player. The position of each player and the ball, their moving directions, and speeds must be computed in real-time to decide each player's movement. The data must be collected, processed, and delivered in the given time frames. It is assumed that the system is well equipped with dedicated resources to handle the predetermined real-time requests. Real-time tasks are unsuspendable, i.e., once it is scheduled, it will execute until the task terminates. The task may release certain resources during its execution if they are no longer needed. For example, the CPU can be freed soon after the request is started and the dedicated resources, e.g., the video/audio, communication channel, and direct memory access (DMA) coprocessors will execute the task 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 dedicated resources like audio coprocessor and the communication channel continue to serve the request.

The priority is an integer between $[0 .. p]$, $p \geq 0$. The priority of a task increases with the time. It is used to represent the urgency of non-real-time applications from task initiator's point of view. The initial priority is set by the task initiator. The priority-based tasks are suspendable. They will be given a quantum of execution. When the quantum expires, the task is suspended and 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 for, the higher the priority will be.

Postconditions

A selected task is executed. The format of the request is a vector of following components:

- tid: the task id;
- sid: sender id;
- rset $\in 2^{CRS}$: the requested resource set to complete the task.

The selected request R_i must meet conditions:

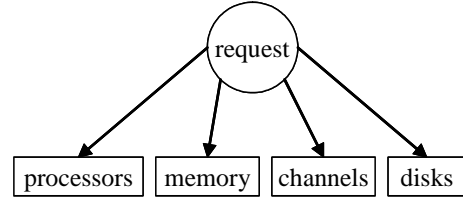
- $dl(R_i) > 0$ if the task has a deadline, or
- $pri(R_i) \geq \max(pri(R_j))$ for $j = 1, 2, \dots, m$, and $j \geq i$.
- $eng(N_i) \geq eng$
- $sigstreng(N_i) \geq sigstreng$

3.3 Redundant resource scheduling

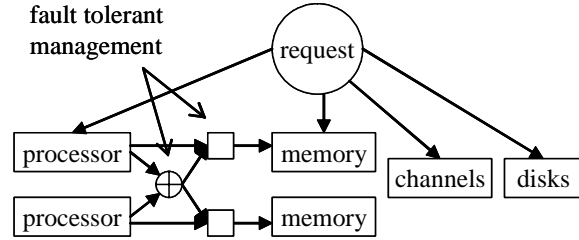
To support the high reliability requirement and fault tolerant computing, the scheduler may schedule redundant resources to gain extra reliability for certain tasks. figure 3 shows an example where a task request can be met functionally with the basic resource allocation. The redundant resource allocation allows the task 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.

In figure 3, the fault tolerant management part shows that two processors are allocated to perform duplicate execution of a critical task. 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 [18] guarantee to allocate replicas of the same task on different computing nodes. In the task scheduler presented in this paper, this requirement is guaranteed, because different CPUs and other resources are considered as separate resources and a resource can be allocated only once to any task. In other words, the same resource cannot be allocated to the replicas of the redundant tasks.



(a) Basic resource allocation



(b) Redundant resource allocation

Figure 3. Basic and redundant resource allocation

4. Task Scheduling Algorithms

The inputs of the scheduling algorithm are two sets of task requests $RTQ = \{rt_1, rt_2, \dots, rt_g\}$ and $PTQ = \{pt_1, pt_2, \dots, pt_h\}$, representing the real-time and priority-based requests, respectively, and the set of available resources $CRS = \{R_1^1, R_1^2, \dots, R_1^{p_1}, R_2^1, R_2^2, \dots, R_2^{p_2}, \dots, R_n^1, R_n^2, \dots, R_n^{p_n}\}$.

The parameter list of each task request S_i is (tid, sid, des, dl, pri, rset, eng, sigstreng), where

- tid: task id.
- sid: sender id;
- dl: the deadline.
- pri: the priority.
- rset $\in 2^{CRS}$: the requested resource set to complete the task.
- eng (optional): the required residual energy.
- sigstreng (optional): the requested signal strength.

In this section, three scheduling algorithms are defined, emphasizing different criteria of optimization. The first algorithm already schedules the real-time tasks first. The second algorithm will schedule a real-time task only if its deadline requires it to be scheduled, thus reducing unnecessary delays for high priority tasks. In the third algorithm, algorithms 1 and 2 are extended by including the parameters for residual energy and signal strength.

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.

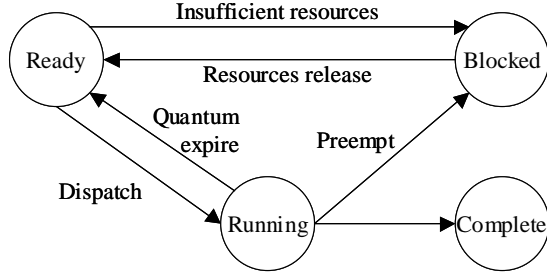


Figure 4. States of the requests being executed

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 tasks 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 task 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 task, 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.

These analyses lead to the following resource scheduling Algorithm 1.

Algorithm 1

```

Input
  RTQ = {rt1, rt2, ..., rtg};
  PTQ = {pt1, pt2, ..., pth},
  // RTQ ∪ PTQ forms the Ready set
  CRS = {R11, R12, ..., R1p1, R21, R22, ...,
         R2p2, ..., Rn1, Rn2, ..., Rnpn}
Resource := CRS;
Running := {};
Blocked := {};
While (True) Do
  If RTQ ≠ empty
    RealTimeSchedule(RTQ, Resource);
  Else
    PrioritySchedule(PTQ, Resource);
Endwhile
Subroutine RealTimeSchedule
  (RTQ, Resource);
MinS := GetMinDeadLine(RTQ);
// return request with lowest deadline
If rset(MinS) ∈ 2Resource Then
  Resource := Resource - rset(MinS);
  Ready := RTQ - MinS;
  Dispatch MinS;
Else
  Preempt the priority-based tasks
EndSubroutine RealTimeSchedule;
Subroutine PrioritySchedule
  (PTQ, Resource);
MaxS := GetMaxPriority(PTQ);
// return request of highest priority
If rset(MaxS) ∈ 2Resource Then
  Resource := Resource - rset(MaxS);
  PTQ := PTQ - MaxS;
  Dispatch MaxS;
  Start timer(quantum)
  for this request;
  Increase the priority of all
  requests in PTQ;
Else
  Blocked := Blocked ∪ {MaxS};
  If the quantum of the running request S
  times up;
  PTQ := PTQ ∪ 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 delayed 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 as the execution does not result in the deadline misses of real-time requests. The two subroutines `RealTimeSchedule` and `PrioritySchedule` used in algorithms 2 are the same subroutines used in algorithm 1.

Algorithm 2

```

Input
RTQ = {rt1, rt2, ..., rtg};
PTQ = {pt1, pt2, ..., pth};
// RTQ ∪ PTQ forms the Ready set
CRS = {R11, R12, ..., R1p1, R21, R22, ...,
      R2p2, ..., Rn1, Rn2, ..., Rnpn}
Resource := CRS;
Running := {}; Blocked := {};
While (True) Do
  If RTQ ≠ empty
    MinS := GetMinDeadLine(RTQ);
    If dl(minS) ≤ quantum
      RealTimeSchedule(RTQ, Resource);
    Else
      PrioritySchedule(PTQ, Resource);
  Else
    PrioritySchedule(PTQ, Resource);
Endwhile.

```

In this algorithm 2, the following two major subroutines have been defined in algorithm 1:

```

PrioritySchedule(PTQ, Resource);
PrioritySchedule(PTQ, Resource);

```

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 the lowest deadline is scheduled before its deadline. We also assumed that priority tasks are suspendable while real-time tasks are unsuspendable. More complex models of real-time applications are being studied and the task scheduler can be extended based on the new type of tasks.

4.3 Energy and signal strength aware scheduling

The execution of certain tasks may be restricted either the residual energy or the signal strength of the task executors to guarantee the power accommodation or the seamlessness communication between the client and the task executor. In this section we consider the impact of these factors on the scheduling algorithms.

A mapping `Attach(executor)` from executors to resources is defined. For an executor N_i , $\text{Attach}(N_i) \subseteq \text{CRS}$ returns the set of resources attached to the executor N_i . A new subroutine `Mask` is defined to extend algorithms 1 and 2, that masks out those task executors that do not satisfy either the energy request or signal strength energy.

Algorithm 3 (The new subroutine only)

```

Subroutine Mask(ENG, SIGSTRENG);
  ELIGI := {};
  // return eligible resources
  For each Ni Do
    If Eng(Ni) ≥ ENG
      AND SigStreng(Ni) ≥ SIGSTRENG Then
        ELIGI := ELIGI ∪ Attach(Ni);
      Endif
    Endfor
  Return ELIGI;
END

```

The `Mask` subroutine is used in the `RealTimeSchedule()` and `PrioritySchedule()` in algorithms 1 and 2, where the comparison $\text{rset}(\text{MaxS}) \in 2^{\text{Resource}}$ is replaced by $\text{rset}(\text{MaxS}) \in \text{Mask}(\text{eng}, \text{sigstreng})$. With this change, the algorithms 1 and 2 will assign task that satisfy the energy and signal strength requirements.

In other words, Algorithm 3 will work as follows. It checks each executor and mark the resource attached to the executor as eligible if the executor satisfies both the energy requirement and the signal strength requirement. The `Eng()` subroutine returns the current residual energy of an executor while `SigStreng()` subroutine returns the current signal strength of an executor. The problem to be addressed is that the current residual energy and signal strength may not sustain till the complete of the task a quantum of execution. Since the execution time of real-time tasks is known and the priority tasks are scheduled per quantum, we estimate the required residual energy and signal strength. The requirement on residual energy can be computed based on the task execution time or quantum. For signal strength, the problem is more complicated. Wireless nodes may move around during the execution of the task. The change of distance to its neighboring nodes affects the signal strength. We compute the requirement based on the maximum moving speed of the executors and the contribution to the signal decay of

the movement in the task execution time or quantum time.

5. A Prototype of Task Scheduler

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

5.1 Design of the prototype

A prototype of the task scheduler is implemented using Microsoft .Net framework and C#. To test the task scheduler, we implemented different kinds of tasks. The task center consists of several threads, continuously generating requests of using the tasks running one the. The task center resides on one computer and the control center resides on another computer. The executors can run on the same computer or different computers or on PocketPCs. Separate computers and PocketPCs are connected through wireless network, as shown in figure 5.

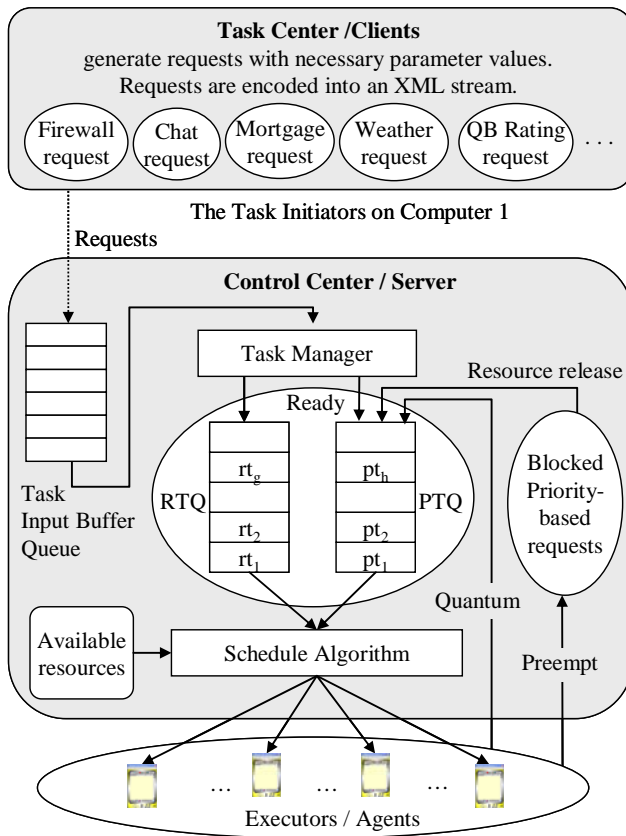


Figure 5. The prototype on two computers

The requests are encoded into character strings and sent to the control center through the TCP/IP protocol. A different group is implementing the clients, task manager and task executors. figure 6 show the implementation of the task manager as a thread which communicates with task center, manages available executors, informs the

scheduler the available task executors.

Each component in the task scheduler is implemented by a thread or a group of threads as well. 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 (PTQ). If a request is a real-time request, the deadline is computed and the request added into the Real-time Request Queue (RTQ). RTQ and PTQ 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 task.

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.

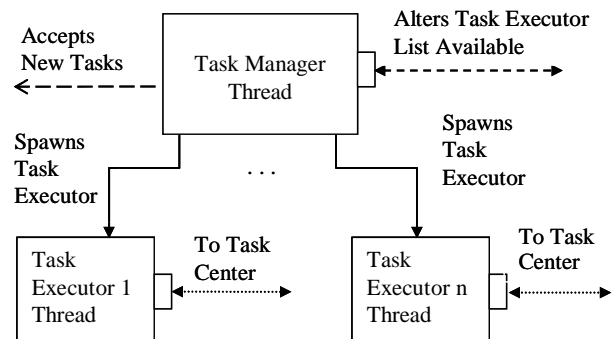


Figure 6. The Task Manger

The Priority Request Queue (PTQ) 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 PTQ, 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 RTQ, and the current time.

5.2 The graphic interfaces to the prototype

To demonstrate our task 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 tasks do not require real-time response. More sophisticated tasks, task manager, and task executors are being developed by other groups in our research project. figure 7 shows the registration window of the task manager. This interface is designed for the client-server model, where the clients must register and be given a password to access the services. Not all the services implemented are shown in this interface.

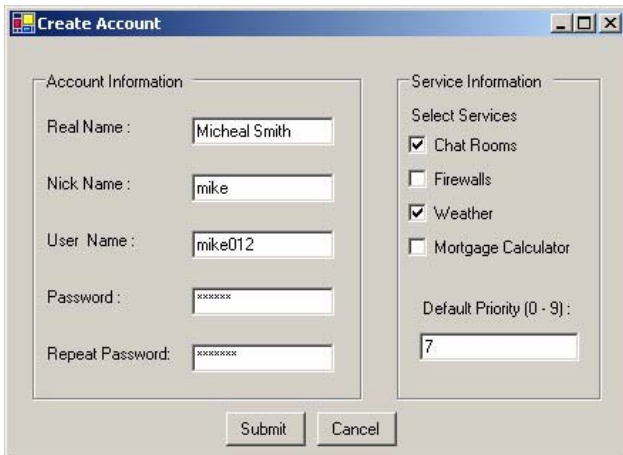


Figure 7. Registration GUI on each client's machine

A task initiator (client) must register to the task manager and select the desired tasks before it can request the tasks. The task 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 tasks, and the lengths of accessing tasks. After a client registered to the task manager, a client can request a task that it has registered using its user name and password.

Figure 8 shows graphic chat rooms running two wireless-connected PocketPC (as executors). The two executors can communicate through the control center. The chat room is implemented as an interactive game. Each player can control the move of the character and send message to the other player. The executors periodically

report their states back to the task manager, including the position of the character and the remaining energy and signal strength at each position.

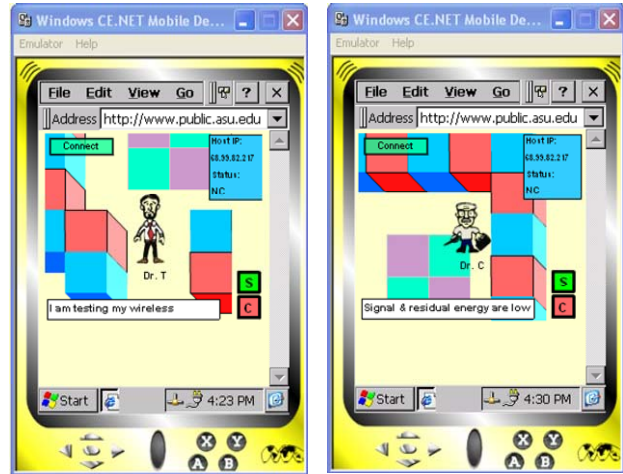


Figure 8. Graphic chat room GUI on executors

Figure 9 is the administration GUI on the sever machine. It dynamically shows the status of the input buffer queue TIBQ, the real-time request queue RTQ, and the priority-based request queue PTQ. It approximately illustrates the percentage of the fullness of each queue. On the left-hand side of figure 9, the tasks currently available are listed and briefly explained of its input requirements.

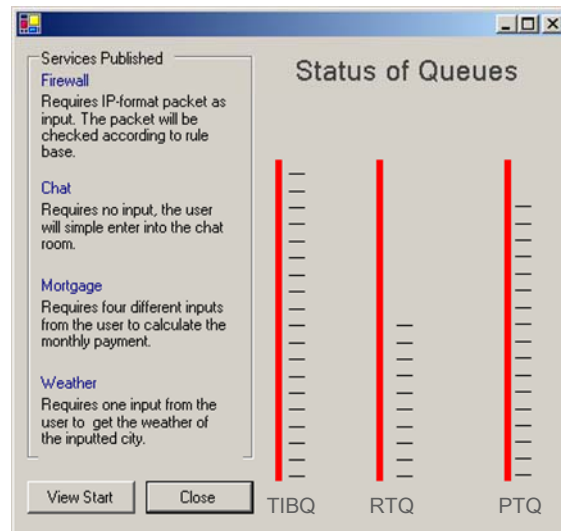


Figure 9. Queue status GUI on the control center

Figure 10 uses an example to show the entire execution process. In the diagram, the XML Write and the XML Reader linked to the PDA is a part of the executor (implemented in the PDA). XML Write and the XML Reader linked to the task manager is a part of the control center. The Quarterback Rating Calculator is another

executor that can provide the information. Assume that the wireless device (the PDA) need to access the information stored in the control center or in another executor. The following scenario elaborates the execution steps as follows.

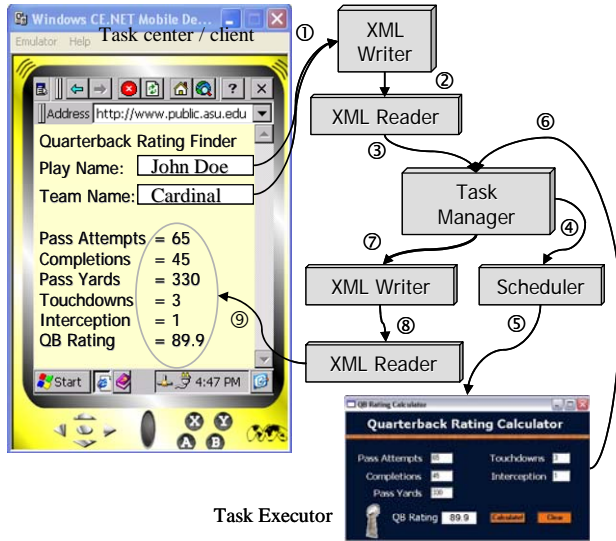


Figure 10. The execution process

1. A client in the task center wishes to know the Quarterback rating of the player John Doe in the football team Cardinal.
2. The request is encoded into XML stream and sent to the server.
3. The XML stream is decoded and sent to the task manager.
4. The task manager verifies the request, identifies its priority or deadline and then sent to the priority queue of the task scheduler.
5. The task scheduler, following the constraints, allocates an executor that can carry-out the task.
6. The task executor looks up the scores of the player and then uses the quarterback calculator to compute the final QB rating. The results are sent back to the task manager.
7. The task manger performs bookkeeping/accounting work on the task.
8. The scores and final rating are encoded into an XML stream and send back to the client in the task center.
9. The XML stream is decoded and the data are display on the client's GUI.

6 Experimental Results

We conducted experiments to observe the performance of all algorithms. We monitor the minimum wait time, the maximum wait time, and the average wait time for both

real-time tasks and priority tasks. The factors indicate the efficiency and fairness of the schedule algorithms. We also vary the load of the system and observe how the factors change according to the load. The changes indicate the scalability of the schedule algorithms.

The setting of our experiments is as follows. The system has 16 different types of resources. Each resource has k identical copies, where $1 \leq k \leq 4$. We randomly generate 32 tasks, each of which has equal probability to be either real time task or priority task. Each real time has a randomly generated deadline \leq current time plus 128. Each priority task has a randomly generated priority \leq 16. Both real time and priority tasks have a randomly generated runtime. The maximum running time of all tasks is 64. A quantum time is of 4 units. The system load is modeled by the generating rate of new tasks, which varies from 2 quanta per new tasks to 64 quanta time per new tasks. We model the wireless environment by randomly decreasing available copies of resources. The wait time for a task is defined as

$$WallClockTime - CPUTime,$$

where $CPUTime$ is the time the task is at running state, and $WallClockTime$ is the overall turn around time to finish the task.

Figure 11 shows the minimum, maximum, and average wait time of real tasks with Algorithm 1. Figure 12 shows the average wait time of real tasks with Algorithm 1 and 2. Figure 13 shows the average wait time of priority tasks with Algorithm 1 and 2. Figure 14 compares the average wait time of real time tasks and priority tasks with Algorithm 2.

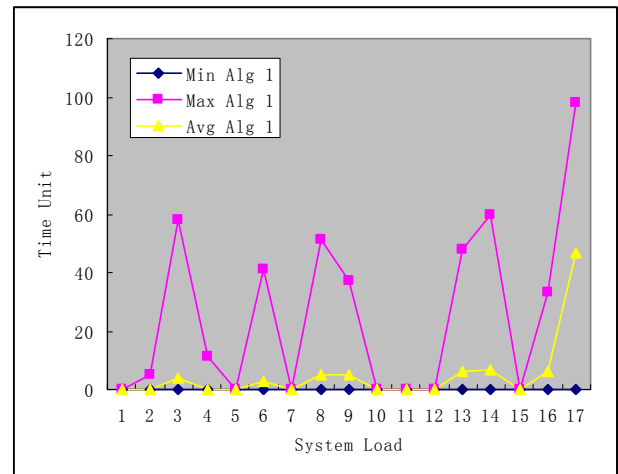


Figure 11. Wait Time of Real Time Tasks of Algorithm 1

The general trend of wait time is increasing with respect to system load for both real time tasks and priority tasks. The experimental data suggested that in either Algorithm 1 or 2, the wait time of priority task increase dramatically high when the system load increases. The experimental data shows that in either Algorithm 1 or 2, the wait time of

real time tasks are favored than that of priority tasks. However, Algorithm 1 tends to more favor real time tasks (figure 12). Both Algorithm 1 and Algorithm 2 exhibit almost the same effect on priority tasks (figure 13).

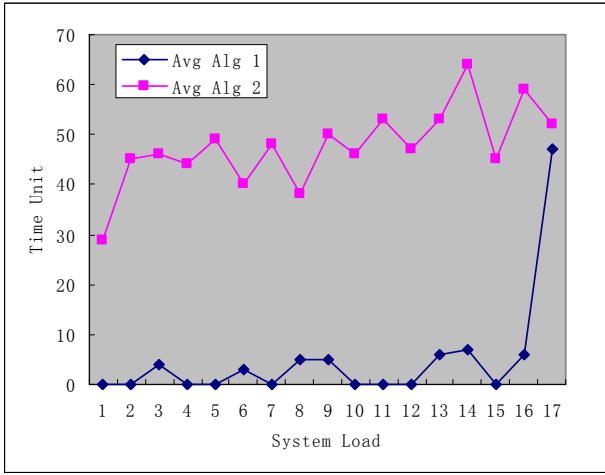


Figure 12. Average Wait Time of Real Time Tasks

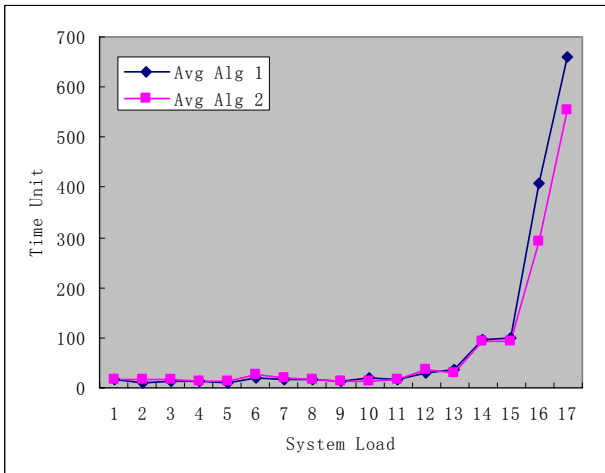


Figure 13. Average Wait Time of Priority Tasks

Figure 14 shows that Algorithm 2 tends to be more fairness where the average waiting time of real time tasks and priority tasks are comparable when system load is within a proper range. When system load increases after a limit, the priority tasks are punished severely because real time tasks are guaranteed to be executed before deadline and the system cannot handle the extreme system load.

The experimental data suggests that Algorithm 1 or 2 scales within a fairly large range of system load. When the system load exceeds a certain limit, the real-time tasks can still be scheduled to meet their deadlines while the priority tasks will be significantly delayed.

The experimental data suggests that Algorithm 3 has comparable efficiency and fairness with that of Algorithm

2. This is a positive sign that our algorithms scale well in wireless environments.

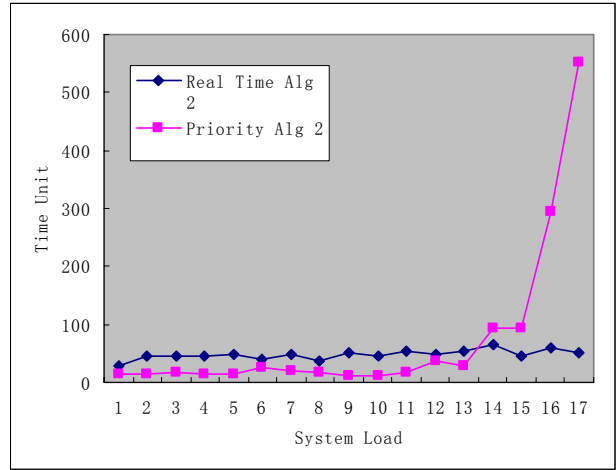


Figure 14. Average Wait Time with Algorithm 2

Figure 15 shows that the wireless environment generally increases the average wait time. Because the instable communication strength and energy, some copies may be temporally unavailable, which may delay the execution of both real time and priority tasks.

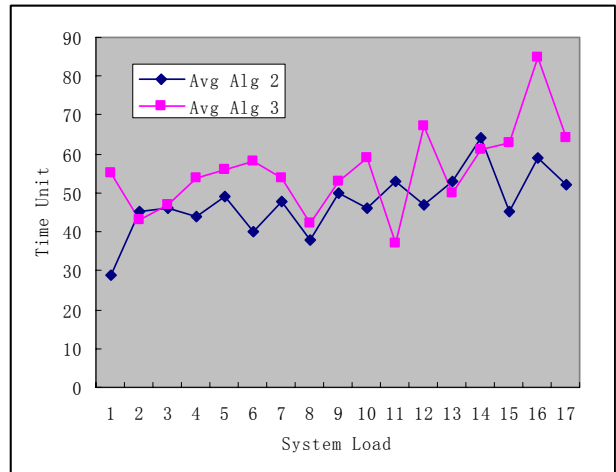


Figure 15. Average Wait Time of Real Time Tasks

7 Conclusions and Future Work

In this paper, we first introduced a distributed simulation environment and a dependable distributed system simulated in this environment.. Three task scheduling and resource allocation algorithms were implemented and evaluated, including the schedule of ordinary priority-based, real-time, fault-tolerant, energy-aware, and signal-strength-aware tasks. Several sample applications are also implemented on the

simulation system, including a fault-tolerant firewall, graphic chat rooms, and a quarterback rating task, etc. The main purpose of the simulation is to prove the concept and feasibility of scheduling different kinds of tasks with complex requirements. This work is a part of a large project aimed at developing mission-critical embedded systems, including reconfigurable and recomposable applications. In the current implementation, the tasks are preloaded into the executors and the schedule of a task is simply an invocation of the task. The future work include real-time modification of existing tasks at binary code level, the deployment of new tasks to replace existing tasks, and the real-time verification and validation of dynamically modified and recomposed code [9].

References

- [1] J. -C. Laprie, "Dependable Computing and Fault Tolerance: Concept and Terminology," IEEE 15th Annual international 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," 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] Sha Liu, "Open Challenges in Real Time Embedded Systems", SIGBED Review, Volume 1, Number 1, April 2004
- [6] Sha Liu: "Upgrading Embedded Software in the Field: Dependability and Survivability," EMSOFT 2002: 166-18.
- [7] A. Mok, F. Wang, and E. A. Emerson, "Distributed real-time system specification and verification in APTL," ACM Transactions on Software Engineering and Methodology, vol. 2, no. 4, October 1993, pp. 346-378
- [8] W.T. Tsai, C. Fan, Z. Cao, B. Xiao, H. Huang, X. Liu, X. Wei, R. Paul, Y. Chen, and J. Xu, "A Scenario-Based Service-Oriented Rapid Multi-Agent Distributed Modeling and Simulation Framework for SoS/SOA and Its Applications," Foundations 2004: A Workshop for VV&A in the 21st Century, Oct. 2004, www.scs.org/confernc/foundations/foundations04.htm
- [9] W.T. Tsai, X. Liu, Y. Chen, R. Paul, "Dynamic Simulation Verification and Validation by Policy Enforcement", 38th Annual Simulation Symposium, San Diego, CA, April 2005
- [10] H. Karatza, R. Hilzer, "Parallel Job Scheduling in Homogeneous Distributed Systems," Simulation, Trans. of the Society for Modeling and Simulation International, Vol.79, No. 5-6, May - June, 2003, pp.287-298.
- [11] A. Cheng, R. Wang, "A New Scheduling Algorithm and a Compensation Strategy for Imprecise Computation," The 20th International Computer Software and Applications Conference (COMPSAC), Hong Kong, September 2004, pp. 167 - 172.
- [12] W. Zhu, "Allocating Soft Real-Time Tasks on Cluster, Simulation," Volume 77, Number 5-6, Nov/Dec. 2001, pp. 219 - 229.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] S. Hazelhurst, A. Attar, R. Sinnappan, "Algorithms for Improving the Dependability of Firewall and Filter Rule Lists," International conference on dependable systems and networks, New York, June 2000, pp. 576 - 585.
- [17] 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.
- [18] 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.

- [19] Y. Chen, "A service scheduler in a trustworthy system," 37th Annual Simulation Symposium, Arlington VA, April 2004, pp. 89-96.