

# The Design and Implementation of A Real-Time Data Dispatching System

Nei-Chiung Perng, Neung-Tsung Tsai, Jen-Wei Hsieh, and Tei-Wei Kuo  
{d90011,r89074,d90002,ktw}@csie.ntu.edu.tw

Department of Computer Science and Information Engineering  
National Taiwan University Taipei, Taiwan, ROC 106

## Abstract

*This paper describes the design of a real-time data dispatching system (RTDDS), which is motivated by the needs of a performance guarantee on real-time data services by front-end clients. RTDDS consists of homogeneous machines with a quality of service performance guarantee. We address the resource allocation problems and consider Bin-Packing algorithms to assign requests to machines of RTDDS. Processes scheduling over each machine is done autonomously by SRP-based algorithms. The goal is to maximize the number of concurrent clients and to meet the individual quality of service requirements of clients at the same time.*

## 1. Introduction

The popularity of Internet has triggered the great demand of information technology in recent years. Various vendors now provide services in different styles over the Internet, free or charged, to customers. Issues in how to improve system performance in many ways, such as scalability, availability, reliability, etc., are under serious investigation. Most work focuses on how to increase the capability of servers, clustered or not, to serve more clients, instead of multicasting data to clients based on their individual quality-of-service requirements. Such an observation underlies the goal of this research.

Clients over the Internet usually face different statuses of networks and different qualities of services from providers. They often need to deal with different specifications of services, even under the same service type, such as music, and have to search for service providers which provide their needed services. Nodes with directory services and quality control are needed in the entire infrastructure for information-services over the Internet. The design and implementation of the relaying-service system investigated in

this paper aims at meeting the quality of service (QoS) requirements of selected clients with a scalable architecture. We assume that the relaying-service nodes, at one end, are attached to networks with a sufficient network bandwidth to acquire services from service providers. At the other end, the relaying-service nodes cache data from the service providers and relay the data to clients according to the QoS requirements of clients. The relaying-service nodes regulate network traffic from the nodes to clients according to the requirements of each client, e.g., a specified number of bytes per specified time units (so that clients are not overflowed with data). A naming service should be provided as well, and multicasting services must also be supported to reduce the workload of the relaying-service nodes. Different from the research work in proxy servers, streaming services, or network bandwidth management, this work is mainly on the design and implementation of relaying services from service providers to front-end clients, and it could be considered as an extension of ordinary proxy servers.

This paper aims at the design and implementation of a relaying-service system from service providers to front-end clients with QoS guarantees. A real-time data dispatching system (RTDDS) is implemented to demonstrate the feasibility of this work. RTDDS consists of a collection of machines, which collectively serve as relaying-service nodes to regulate network traffic from the nodes to clients according to the requirements of each client. RTDDS could be considered as a variation or an extension of proxies; the main goal of RTDDS is to subscribe services on behalf of clients and pass (or multicast) data to clients based on their individual QoS requirements. The basic assumption is to have sufficient network bandwidth for service delivery from service providers, similar to that for powerful caching servers for communities. We consider Bin-Packing approximation algorithms to partition workload among machines of RTDDS. We adopt the stack-based resource protocol (SRP) [2] for autonomous workload scheduling on each machine, where SRP provides a uniform treatment of various multi-unit resources and could be associated with dynamic and fixed priority assignment schemes. RTDDS is designed

as a message-oriented distributed system which provides load balancing with a broker-based mechanism. We adopt a pure-Java solution which delivers the promise of "Write Once, Run Anywhere" through the Java Virtual Machine (JVM).

The rest of the paper is organized as follows. Section 2 describes the detailed system design of RTDDS. The system implementation issues are discussed in Section 3. Section 4 provides the experimental results of the system to demonstrate the capability of RTDDS on quality-of-service guarantee and graceful degradation. Section 5 is the conclusion.

## 2. Real-Time Data Dispatching System

### 2.1 System Architecture

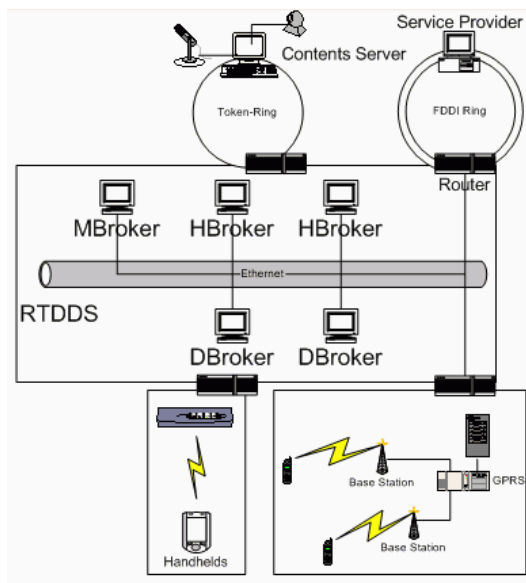


Figure 1. System Architecture

Figure 1 shows the system architecture of RTDDS. The objective is to provide fixed-rate relaying-services for clients over different platforms with different QoS requirements, independent of the data-transmission rates and variations of service providers. RTDDS consists of a collection of homogeneous or heterogeneous machines over a local area network to provide a scalable solution for on-time data delivery. Note that these machines could be heterogeneous in our design. At this time we implement the system with homogeneous machines. A combination of master-slave and broker-based architectures is adopted, where a *master broker* (MBroker) runs on one machine of RTDDS to assign workloads to slave machines so that each slave machine will

then acquire data from service providers by itself and multicast the acquired data to clients according to their individual QoS requirements.

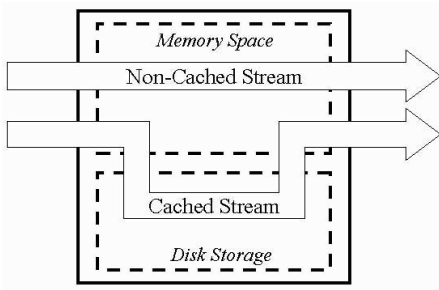
Two kinds of brokers for data acquisition and delivery called *home broker* (HBroker) and *destination broker* (DBroker) are introduced. When a request comes from a client, it first goes to MBroker. The MBroker will assign a HBroker to acquire data from the corresponding service provider and assign a DBroker to send data to the client according to the QoS requirements of the client. The HBroker may cache data from the service provider at its residing machine, depending on the services (which will be discussed in Section 2.2). The responsibility of a DBroker are to acquire data from a proper HBroker and to send/multicast the data to clients according to the QoS requirements of clients. The criteria in the assignments of HBrokers and DBrokers are not to overload each slave machine and to maximize the number of serviced clients based on the capability of machines and the QoS requirements of clients. The assignment of HBrokers and DBrokers to slave machines will be discussed in Section 2.3.

The naming service of RTDDS is provided by the MBroker. It is the responsibility of the MBroker to keep a directory of services. When a client request does not come with a specific host address, and the directory of the MBroker does not contain service providers for the request, the current design of the MBroker will simply reject the request.

### 2.2 Handling of Streams

Services in RTDDS can be roughly categorized into two types: regular and timely services. *Timely services* are associated with services of data with aging problems so that every piece of (continuous or discontinuous) data of the services must be delivered to a selected front-end in an on-time fashion. Note that data with aging considerations will have their validity passing with time. For example, the delivery of stock quotes is an timely service because no one wants to complete a transaction with out-of-date data. *Regular services* are services for data without aging problems, e.g., impressive scenes of classic movies. However, we must point out that there could be timing constraints in the delivery of data under timely or regular services. For example, the delivery of a video stream is associated with timing constraints for each frame or each group of frames although the delivery could be classified as a regular service.

Two kinds of delivery schemes are defined for timely and regular services: *non-cached-stream* and *cached-stream* deliveries. *Cached-stream* delivery requires RTDDS to cache data from the service providers during the services to the clients. The HBroker who is responsible for acquiring a ser-



**Figure 2. Data Dispatching over RTDDS**

vice (from a service provider) during a cached-stream delivery should do the caching, and the caching process should include the notification to and a yes-or-no decision from the MBroker. The cached data could be saved for later requests from clients. When later on a new request on the cached data comes in, MBroker would assign the HBroker to handle the request. As astute readers may notice, cached-stream delivery should only serve for regular services because the reusing of cached data, e.g., video streams or a popular song, implies that data have no aging problems. Obviously, each machine has a limited capacity in the storage of cached data and the handling of cached data, i.e., I/O operations on retrievals and saving of cached data. The decision of caching should be done by the MBroker because it manages the entire workload of the system. When the capacity of a machine is full, the corresponding HBroker should notify the MBroker. The MBroker would determine the removing of the selected cached data and notify the HBroker.

Non-cached-stream delivery could be associated with either timely or regular services. No data are cached at any machine. As a result, any such delivery only consumes CPU time, memory space, and network bandwidth. Note that DBrokers are not involved in the decision of caching of data. The main responsibility of DBrokers is on the regulating of traffic to clients. All data are directly from HBrokers, regarding of whether HBrokers supply DBrokers data directly from their caching storage or from the corresponding service providers.

## 2.3 Resource Allocation

### 2.3.1 Workloads and Resource Requirements

HBrokers (and their corresponding machines) are units in workload assignments. The workload of each client request  $\tau_i$  can be defined by two parameters  $N_i$  and  $P_i$ , where  $\tau_i$  requests RTDDS to provide the client  $N_i$  bytes per  $P_i$  units of time. Several requests of the same data would be from

the same service provider (the same HBroker) but have different parameters  $N_i$  and  $P_i$ . When one or multiple clients request a data, the corresponding HBroker must acquire a service from a proper service provider. The service provider could have fixed or variable rates in passing data to the HBroker.

Let us use three parameters to model the workload from a service provider:  $N^{max}$ ,  $N^{min}$  and  $P$ .  $N^{max}$  and  $N^{min}$  denote the maximum and minimum numbers of bytes transferred from the service provider within each  $P$  units of time. Given a collection of  $n$  client requests  $T_i = \{(N_{i,1}, P_{i,1}), \dots, (N_{i,n_i}, P_{i,n_i})\}$ , let  $N_i^{max}$ ,  $N_i^{min}$  and  $P_i$  be the parameters modelling the workload of the HBroker (where  $N_i^{max}$  and  $N_i^{min}$  denote the maximum and minimum numbers of bytes transferred from the service provider within each  $P_i$  units of time). Suppose that a well-known dual buffer scheme, which uses one buffer to receive data and another to send data, is adopted<sup>1</sup>. The following formula must be satisfied for the minimum size of the buffer:

$$N_i^{min} \geq \max_{j=1}^n \left[ \frac{P_i}{P_{1,j}} \right] N_{i,j}$$

where  $P_i \geq P_{i,j}$  for all  $1 \leq j \leq n$ . For the simplicity of implementation, let  $P_j | P_{i,j}$  for all  $1 \leq j \leq n$ . The buffer requirement of the HBroker on the above service is  $2N_i^{max}$ . Let the workloads of the client requests  $T_i = \{(N_{i,1}, P_{i,1}), \dots, (N_{i,n_i}, P_{i,n_i})\}$  and their corresponding HBroker workload  $(N_i^{max}, N_i^{min}, P_i)$  be denoted collectively as  $DW_i = ((N_i^{max}, N_i^{min}, P_i), T_i)$ . Each  $DW_i$  is called a *HBroker delivery workload* in the system.

Suppose that the  $i_{th}$  HBroker is assigned a collection of  $m_i$  services  $H_i = \{(N_{i,1}^{max}, N_{i,1}^{min}, P_{i,1}), \dots, (N_{i,m_i}^{max}, N_{i,m_i}^{min}, P_{i,m_i})\}$  for clients. The minimum memory space (for the dual buffer scheme) for the HBroker is  $2 \sum_{j=1}^{m_i} N_{i,j}^{max}$ . Let  $T_{i,j} = \{(N_{i,j,1}, P_{i,j,1}), \dots, (N_{i,j,n_{i,j}}, P_{i,j,n_{i,j}})\}$  be the collection of client requests which corresponds to a workload from a service provider  $S_{i,j} = (N_{i,j}^{max}, N_{i,j}^{min}, P_{i,j}) \in H_i$ . Let  $CH_i$  be the maximum subset of  $H_i$  for cached-stream service. When the Stack-Based Resource Policy (SRP) is used for scheduling of all resources on one machine, where SRP could use either a dynamic priority assignment scheme (such as the earliest deadline first algorithm) or a fixed priority scheduling scheme (such as the rate monotonic algorithm) [7]. The minimum I/O bandwidth requirement is

<sup>1</sup>The implementation of the dual buffer scheme could be done by a well-known circular buffering approach where a less amount of memory space is needed. However, when the variation of data transmission from the service provider is large, then the buffer size must be increased to prevent the overflowing of the buffer.

$$\sum_{(N_{i,j}^{max}, N_{i,j}^{min}, P_{i,j}) \in CH_i} \left( \frac{N_{i,j}^{max}}{DiskAccessRate} \right. \\ \left. + \sum_{(N_{i,j,k}, P_{i,j,k}) \in T_{i,j}} \frac{N_{i,j,k}}{P_{i,j,k}} \right) + B_i$$

where  $DiskAccessRate$  is the worst-case disk access rate (i.e., that including the worst-case seek time and latency delay, unless a good placement scheme is adopted).

Note that  $B_i = \frac{OneBlockAccessTime}{MinPeriod}$ , where  $OneBlockAccessTime$  is the worst-case access time for one block access, and  $MinPeriod$  is the minimum periods of all client and service workloads. The correctness of the above formula follows from the schedulability analysis results in the research results of the stack-based resource policy and the constant utilization servers [4] and when the earliest-deadline-first algorithm (EDF) is used in scheduling disk I/O operations. Note that EDF assigns the request (or process) with the closest deadline the highest priority. We assume that CPU time is always sufficient for each HBroker under the assumption that disk I/O could be handled for the HBroker because I/O is much slower (However, when the assumption is improper, the formula for the CPU bandwidth requirement could be derived in a similar way as that for I/O bandwidth requirement).

### 2.3.2 Assignments of HBrokers and DBrokers

For the simplicity of implementation, we assume that each machine is assigned a HBroker in this paper. *The technical question here is how to assign service workloads to HBrokers without overloading any HBroker.* Given a fixed set of service workloads, the HBroker assignment problem could be shown as a NP-Complete problem.

**Theorem 1** *The HBroker assignment problem is NP-Complete.*

**Proof.** An instance of the HBroker assignment problem has a collection of workloads  $\{(N_1^{max}, N_1^{min}, P_1), \dots, (N_n^{max}, N_n^{min}, P_n)\}$ . It is to partition the  $n$  service notices over a given number of HBrokers, under the constraint that the total utilization of each HBroker does not exceed its capacity of corresponding HBroker. This problem is a NP problem since we could guess a partition and verify it in a polynomial time.

We could show that the HBroker assignment problem is NP-Complete by reducing it from the Bin-Packing problem.

An instance of the Bin-Packing problem has a finite set  $U$  of items, a size  $s(u) \in Z^+$  for each  $u \in U$ , a positive integer bin capacity  $B$ , and a positive integer  $K$  [3, 6]. The Bin-Packing problem asks to have a partition of  $U$  into disjoint sets  $U_1, U_2, \dots, U_k$  such that the sum of the sizes of the items in each  $U_i$  is no more than  $B$ ?

The reduction is straight forward because the HBroker assignment problem is exactly the same as the Bin-Packing problem with  $DiskAccessRate$  being a very large number such that the I/O bandwidth requirement could be virtually removed.  $\square$

Since the HBroker assignment problem is NP-Complete, approximation algorithms are needed for the assignments. There are excellent approximation algorithms already exist for the Bin-Packing problem, such as *Next-Fit* and *First-Fit* [9]. Because the advance of hardware technology already makes memory space a minor issue, compared to the performance of disks, we propose to apply Bin-Packing approximation algorithms to assign workloads to HBrokers with only the considerations of I/O bandwidth requirements. We assume that the memory space is sufficient in the implementation.

The assignment of DBrokers to machines in RTDDS is independent of the assignment of HBrokers in the considerations of the architecture design. However, when implementations are considered, each DBroker should be resident on the same machine as the HBroker which is assigned client requests corresponding to the workloads of the HBroker. When several requests of the same data come from clients with a distance such that they had better to receiving data from different machines in RTDDS, the assignment of DBrokers, once again, becomes a very difficult problem (which can be, in fact, reduced from a weighted set covering problem given the following informal definition for the DBroker assignment problem):

Let each DBroker have a set of client requests, and each HBroker have a set of workloads for the client requests, where the sets of some DBrokers could have non-empty overlapped subsets. Assume that the underlying network is a broadcast network such that the considerations of network overheads could be ignored. When a DBroker and a HBroker are assigned on the same machine, the number of overlapped client requests (for the DBroker) and workloads (for the HBroker) is called the *matched number*. *The problem is how to assign DBrokers to machines, where the assignment of HBrokers on machines given, such that the sum of matched numbers of all machines is maximized.* Note that each machine will be assigned exactly one HBroker and one DBroker.

For the purpose of the design goals of RTDDS, we assume that each machine is equally good in servicing any

client request, where RTDDS is mainly designed to regulate network traffic according to the QoS requirements of clients. Machines of RTDDS work as an integrated clustered "virtual" machine in processing client requests. We do not allow client requests of the same data being assigned to more than one DBroker. Furthermore, let a DBroker reside at the same machine as the HBroker which services its client requests. The HBroker assignment problem defined above could be extended as the HBroker/DBroker assignment problem, except that the minimum memory space (for the dual buffer scheme) for the HBroker becomes  $2 \sum_{j=1, m_i} N_{i,j}^{max} + 2 \sum_{(N_{i,j,k}, P_{i,j,k}) \in T_{i,j}} N_{i,j,k}$ . The approximation algorithms discussed in the previous paragraphs could be used similarly.

### 3. System Implementation

#### 3.1 A Master-Slave Architecture

An asymmetric master-slave architecture is adopted in the design and implementation of RTDDS. One master machine is assigned to run MBroker. MBroker must assign workloads to HBroker and DBrokers. The system architecture of RTDDS could be supported by two well-known distributed system architectures: OMG's Common Object Request Broker Architecture (CORBA) [8] and Sun Microsystems's Java Message Service (JMS) [5]. CORBA is an open, vendor-independent specification for an architecture and infrastructure which applications can adopt to work together over networks. With the OMG Interface Definition Language and standardized protocols GIOP and IIOP, CORBA allow any CORBA-based program from any vendor, operating system, and programming language to cooperate with other CORBA-based programs. JMS, which consists of a set of APIs, provides a reliable, flexible service for the asynchronous exchange of critical business data and events throughout an enterprise. JMS also provides a common API and provider framework that enables the development of portable, message based applications in the Java programming language.

CORBA and JMS are products designed for different purposes. We choose JMS for the implementation of RTDDS for two reasons: (1) JMS, which is mainly based on asynchronous communication, provides a connectionless communication approach, where most CORBA implementations depend on TCP-based communications. Note that TCP-based communication is better for reliable communication, but connectionless communication, such as UDP, in general provide better performance over reliable networking environments. Since the implementation of RTDDS is over reliable intranet, JMS is chosen as the implementation framework and platform. (2) Java and JMS together

provide a better cross-platform development environment. They provide better portability for the implementations of RTDDS.

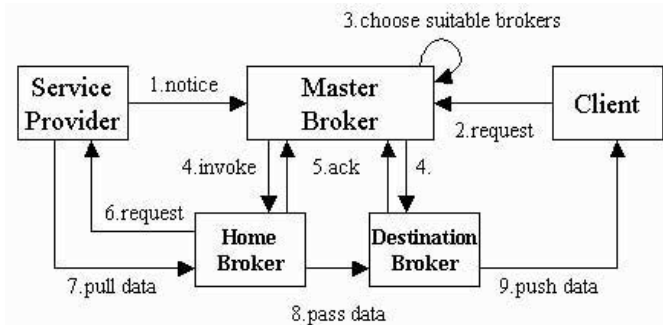


Figure 3. Control Flow of RTDDS

Numerous vendors provide JMS products which follows the JMS standard. OpenJMS [1] version 0.7.2 build 14 was used to implement RTDDS, where OpenJMS is an open source implementation of JMS API 1.0.2 specification developed by ExoLab Group. Over the bottom structures provided by OpenJMS, brokers negotiate with each another by instant message exchanging. Figure 3 is the flow of control information when a service provider publishes data to RTDDS: The first step is a registration to the MBroker, in which a service provider sends a service notice to MBroker. A service notice consists of the host address, binding port number, and service name of the provider. After receiving a request from a client for a published service (Step 2), MBroker chooses a suitable HBroker and a suitable DBroker for service, with the methodology described in the previous section (Step 3). The assignment of a HBroker and the DBroker is done by an invoke message to the HBroker and the DBroker (Step 4). The HBroker and the DBroker both need to send back an acknowledgement back to the MBroker (Step 5). The HBroker then acquires data from the service provider (Steps 6 and 7) and sends the data to the DBroker (Step 8). The DBroker then sends the data to the client based on its QoS requirements (Step 9).

#### 3.2 Implementations of Brokers

To regulate the traffics of real-time data dispatching to clients, each broker is designed as a multithreaded server to handle concurrent client requests to increase responsiveness and to reduce blocking, as shown in Figure 4. Whenever a new client request comes in for a HBroker or DBroker, a thread is assigned to handle the request. A thread pool is maintained.

The communication between a service provider and a HBroker is done by a TCP-based connection to provide

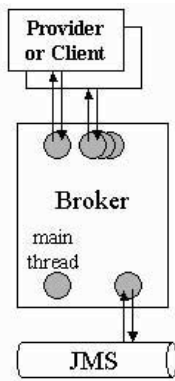


Figure 4. Broker Design Diagram

a reliable communication. The communication between a DBroker and its client is also done by a TCP-based connection because of the same reason. The communication between a HBroker and its corresponding DBroker is done by JMS-based asynchronous communication to maximize the performance, where we assume the existence of reliable intranet for HBrokers and DBrokers. Note that when clients and servers adopt the Real-Time Transport Protocol (RTP) (which is over TCP and UDP), the implementation of RTDDS could utilize the flow control of RTP for multimedia data transmissions.

MBroker, HBrokers and DBrokers adopt the following JMS framework for the exchanging of asynchronous messages. MBroker, HBrokers, and DBrokers which send and receive asynchronous messages inherit the interface *javax.jms.MessageListener*. Users should implement the *onMessage* method. While a message delivered to brokers, *onMessage* method will be called.

```

import javax.jms.*;
public class RtdsBroker implements MessageListener {
    ...
    public void onMessage(Message message) {
        /* write todo jobs here */
    }
    ...
}
  
```

## 4. Performance Evaluation

### 4.1 Experimental Environments and Performance Metrics

The purpose of the experiments was to evaluate the performance (or QoS guarantee) of RTDDS and its overheads.

The RTDDS was implemented in pure-Java over ExoLab OpenJMS version 0.7.2 build 14. The evaluation platform was over several Pentium-III 600MHZ personal computers. In the current version, we generated dedicated Service providers to provide data periodically. They were deployed over a remote network. One of them was connected directly to Hinet (which is provided by Chung-Hwa Telecom ISP) with ADSL (64kbps uplink and 512kbps downlink). Our RTDDS was set up at the Real-Time and Embedded Systems Laboratory at the National Taiwan University. RTDDS was connected to a LAN in the laboratory. The LAN was connected to Hinet through TANET (which is provided by the ROC Education Ministry). Clients of RTDDS were connected to RTDDS over LAN in the laboratory.

The major performance metric was the stability of traffics, i.e. the QoS guarantee, from RTDDS to clients. Clients requested for 4096, 8192 and 16384 bytes of data transfer for every 20 ms from RTDDS (or service providers). When RTDDS was adopted, RTDDS acquired data from the service providers on behalf of clients and forwarded to clients according to the requirements of clients. We run the requests for 1000 times with or without RTDDS. We then take the results in the middle 500 samples to remove improper experimental factors. We also evaluated the overheads of RTDDS, which was mainly the delay of data transmissions from the receiving of data (from service providers) and the forwarding of data (from RTDDS). The overheads were measured five times and averaged.

### 4.2 Experimental Results

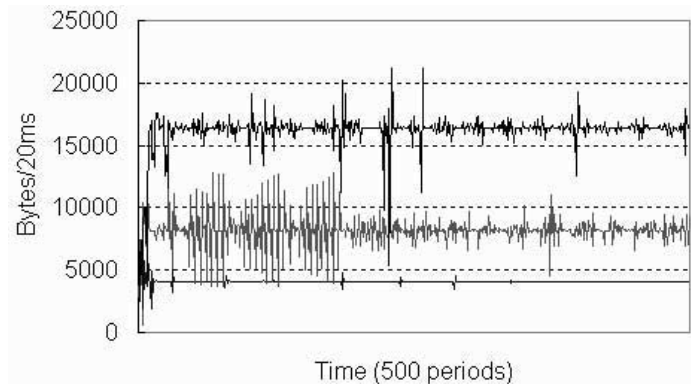
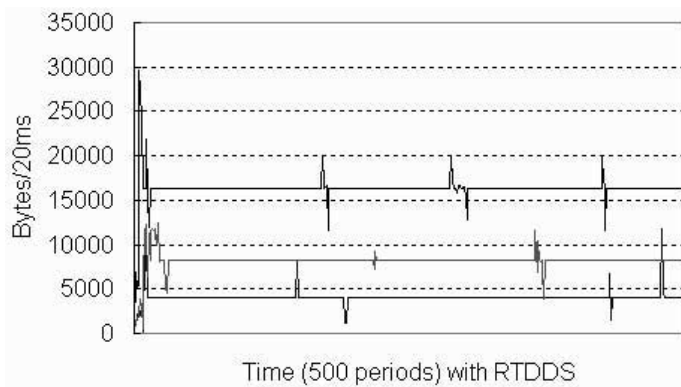


Figure 5. The distributions of data transmissions without RTDDS

Figure 5 shows the distributions of data transmissions for clients without RTDDS. In 20 ms, clients requested 4096, 8192 and 16384 bytes respectively. Figure 6 reports the results of the same experiments with RTDDS. When clients



**Figure 6. The distributions of data transmissions with RTDDS**

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Delay	2804	2684	2524	2784	2664	2692ms

**Table 1. The overheads of RTDDS in transmission delay**

requested data through RTDDS, they always received a very stable transmission rate according to the requests of clients. When RTDDS was not adopted, the data transmission rates fluctuated with a wide scale. During most of the experiment time, the rates could go up and down violently.

Table 1 shows the overheads of RTDDS, which was mainly the delay of data transmissions from the receiving of data (from service providers) and the forwarding of data (from RTDDS). The delay was about 2.5 seconds. We must emphasize that the delay only occurred when the service was first delivered. Once services were delivered, clients would no longer feel any delay. The delay could also be called start-up delay of services. It was the price paid for traffic regulation under RTDDS. The large fluctuation in the beginning of Figure 6 were due to the delay of data transmissions, as shown in Table 1.

## 5. Conclusion

This paper proposes the design and implementation of a relaying-service system from service providers to front-end clients with QoS guarantees. RTDDS consists of a collection of machines, which collectively serve as relaying-service nodes to regulate network traffic from the nodes to clients according to the requirements of each client. We consider Bin-Packing approximation algorithms to partition workload among machines of RTDDS. We adopt the stack-based resource protocol (SRP) for autonomous work-

load scheduling on each machine. RTDDS is designed as a message-oriented distributed system which provides load balancing with a broker-based mechanism over JVM. We conducted a series of experiments to show the overheads in adopting the RTDDS solution and to demonstrate the capability of RTDDS in regulating traffic for clients.

RTDDS provides extended services over ordinary proxy servers in relaying data streams from service providers to front-end clients. For future research, we shall further explore pulling services from service providers for clients with QoS supports.

## References

- [1] J. Alateras, T. Anderson, and J. Mourikis. *OpenJMS User Guide*. ExoLab Group, February 2002.
- [2] T. P. Baker. A stack-based resource allocation policy for real-time processes. *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.
- [3] E. G. Coffman, J. M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 2, pages 46–93. PWS Publishing Company, 1997.
- [4] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. *IEEE Real-Time Systems Symposium*, pages 308–319, December 1997.
- [5] M. Hapner, R. Burrige, R. Sharma, J. Fialli, and K. Stout. *Java Message Service Specification*. Sun Microsystems, 1.1 edition, April 2002.
- [6] L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM J. RES. Dev.* 21, pages 443–448, 1977.
- [7] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.
- [8] Object Management Group. *Common Object Request Broker Architecture (CORBA/IIOP)*, 3.0.2 edition, December 2002.
- [9] P. W. Shor. How to pack better than best fit: Tight bounds for average-case on-line bin packing. *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 752–759, 1991.