A DISTRIBUTED HARD REAL-TIME JAVA SYSTEM FOR HIGH MOBILITY COMPONENTS

A Dissertation

by

SANGIG RHO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2004

Major Subject: Computer Engineering

A DISTRIBUTED HARD REAL-TIME JAVA SYSTEM

FOR HIGH MOBILITY COMPONENTS

A Dissertation

by

SANGIG RHO

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Approved as to style and content by:

Riccardo Bettati
(Chair of Committee)

Wei Zhao
(Member)

A. L. Narasimha Reddy
(Member)

Rabi N. Mahapatra
(Member)

Valerie E. Taylor
(Head of Department)

December 2004

Major Subject: Computer Engineering

ABSTRACT

A Distributed Hard Real-Time Java System
for High Mobility Components. (December 2004)

Sangig Rho, B.S., Yonsei University;
M.S., Yonsei University
Chair of Advisory Committee: Dr. Riccardo Bettati

In this work we propose a methodology for providing real-time capabilities to component-based, on-the-fly reconfigurable, distributed systems. In such systems, software components migrate across computational resources at run-time to allow applications to adapt to changes in user requirements or to external events. We describe how we achieve run-time reconfiguration in distributed Java applications by appropriately migrating servers. Guaranteed-rate schedulers at the servers provide the necessary temporal protection and so simplify remote method invocation management. We describe how we manage overhead and resource utilization by controlling the parameters of the server schedulers. According to our measurements, this methodology provides real-time capability to component-based reconfigurable distributed systems in an efficient and effective way.

In addition, we propose a new resource discovery protocol, REALTOR, which is based on a combination of pull-based and push-based resource information dissemination. REALTOR has been designed for real-time component-based distributed applications in very dynamic or adverse environments. REALTOR supports survivability and information assurance by allowing the migration of components to safe locations under emergencies such as external attack, malfunction, or lack of resources. Simulation studies show that under normal and heavy load conditions REALTOR re-

mains very effective in finding available resources, and does so with a reasonably low communication overhead. REALTOR 1) effectively locates resources under highly dynamic conditions, 2) has an overhead that is system-size independent, and 3) works well in highly adverse environments. We evaluate the effectiveness of a REALTOR implementation as part of Agile Objects, an infrastructure for real-time capable, highly mobile Java components.

To My Wife Seongeun and Parents

ACKNOWLEDGMENTS

This dissertation could not have been completed without Dr. Riccardo Bettati who encouraged and challenged me throughout my academic program.

I also thank my wife, Seongeun, and my parents for their endless love.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

II ARCHITECTURE 6

A. A Model for Distributed Component-Based Applications . 8
B. Task Model 8
C. The Workload Models 10
1. Invocations of Local Methods 13
2. Invocations of Local and Single-Chained Remote Methods 13
3. Invocations of Local and Multi-Chained Remote Methods 15
D. Admission Control Policy 16

III REAL-TIME AGILE OBJECTS SYSTEM 18

A. A Model for Agile Objects 20
B. A Server-Centric Environment for Real-Time Java RMI . . 20

IV REAL-TIME INFRASTRUCTURE 23

A. Introduction 23
B. Related Work 24
C. Standard Java RMI Problems for Providing Real-Time Capability 26
1. Sun's Java RMI Implementation 26
2. Design Issues for Providing Real-Time Capable Java RMI 29
D. Real-Time Java Threads 30
1. The Creation of Real-Time Threads for Java RMI . . 31
2. The Adjustment of Priorities of Real-Time Worker Threads Based on Admitted Utilization 33
E. A Server-Centric Approach for Preserving Real-Time Timing Constraints 33
F. Guaranteed-Rate Scheduling for Sporadic Real-Time Tasks 35
1. The Total Bandwidth Server 35

CHAPTER Page

2. The EDF Scheduler for Total Bandwidth Servers . . . 37
3. A Probabilistic Approach for Characterizing Total Bandwidth Servers 39
G. Experimental Evaluation 43
1. Local Method Execution Time 44
2. Latency of Remote Method Invocation 46
a. Java VM Running One RMI Server and High Background Load 47
b. Java VM Running One RMI Server and Varying Amount of Background Load 49

V MOBILITY 53

A. Introduction 53
B. Related Work 55
C. Our Methodology for Migration 56
1. Java Object Serialization Protocol 58
2. The Deserialization of Java RMI Server Objects 61
3. Java Classes for Agile Objects Migration Mechanism . 62
a. Interface ao.migration.Migratable 62
b. Class ao.migration.DataSender 64
c. Class ao.migration.MigWrapper 64
d. Class ao.migration.RTJVM 64
D. Methodology 64
E. Experimental Evaluation 67
F. Experimental Results 69
G. Discussion of Experimental Results 73

VI RESOURCE MANAGEMENT 75

A. Introduction 75
B. Related Work 77
C. REALTOR: REsource ALlocaTOR 79
1. REALTOR Scheme 80
a. Community Protocol 80
b. Algorithm H 81
c. Algorithm P 82
2. Other Resource Discovery Schemes 83
a. Pure PUSH Scheme 83
b. Pure PULL Scheme 84

CHAPTER Page

c. Adaptive PUSH Scheme 84
d. Adaptive PULL Scheme 84
D. Analysis of Resource Discovery Message Overhead 84
E. Experimental Performance Evaluation 90
F. Implementation Experience 96

VII CONCLUSIONS AND FUTURE WORK 101

REFERENCES 103

VITA 111

LIST OF TABLES

TABLE Page

I Notations for Workload Models 12

II Message Overhead ($W = n\Delta t, (n = 1, 2, 3, \ldots)$) 88

LIST OF FIGURES

FIGURE Page

1 Middleware for Survivable Systems 2

2 The Task Model 9

3 Invocations of Only Local Methods 14

4 Local Methods and a Single-Chained Remote Method 15

5 Local Methods and a Multi-Chained Remote Method 17

6 Software Components of Agile Objects System 19

7 Agile Objects System Architecture 21

8 The UML Diagram of RMI Client Classes 27

9 The UML Diagram of RMI Server Classes 28

10 The Scheduling of Real-Time Worker Threads for Exported Remote Objects 32

11 The Procedure for Adjusting the Priority of an RMI Real-Time Worker Thread 34

12 The Illustration of Total Bandwidth Server Operations 36

13 The State Diagram of the EDF Scheduler for Agile Objects System . 38

14 Execution Time of Local Method on TimeSys 3.1-RT 44

15 Average and Standard Deviation of Local Method Execution Time on TimeSys 3.1-RT 45

16 The Experiment Environment 46

17 RMI Latency with High Background Load 47

FIGURE Page

18 Decomposition of RMI Latency 49

19 Standard Deviation of RMI Latency 50

20 RMI Latency with Varying Amount of Background Load 51

21 Decomposition of RMI Latency with Varying Amount of Background Load 52

22 Java Object Serialization Protocol 59

23 UML Diagram of Java Classes for Package *ao* 63

24 Experiment Environment for the Worst-Case Latency of Lookup with a Migration 65

25 The Worst-Case Latency of Lookup with a Migration 66

26 Latency for a Lookup with a Migration 69

27 Decomposition of Latency for a Lookup with a Migration 70

28 Decomposition of Latency for a Lookup with a Migration in Terms of Serialized Object Size 71

29 Latency for a Lookup with a Migration with Varying Amount of Background Load 72

30 Decomposition of Latency for a Lookup with a Migration with Varying Amount of Background Load 74

31 Algorithm H in REALTOR 82

32 Algorithm P in REALTOR 83

33 The Network Topology for the Simulation 91

34 Admission Probability 93

35 Number of Messages Exchanged 94

36 Communication Cost per Admitted Task 96

FIGURE Page

37 Admission Success Rate 98

38 Migration Time in Milliseconds 99

CHAPTER I

INTRODUCTION

It is to be expected that many large-scale real-time systems will be increasingly required to be adaptive to changes in operational requirements and load, environmental influences, and changes in the underlying resources. Moreover, the complexity of this type of applications will demand exceptionally high levels of self-manageability. System adaptation and reconfiguration must happen with no – or very minimal – operator input. This ability to self-reconfigure is needed, for example, whenever survivability of the application is critical; that is, when the application must keep running and guarantee the QoS (Quality of Service) despite resource failures and continuous external attacks. Similarly, applications deployed in utility-type data centers [1] may have varying numbers and types of resources at their disposal during their life-time, as the data center operators re-allocate the computing infrastructure across their clients. Applications in such environments must seamlessly adapt to the changing computational resources as well.

Increasingly, applications are designed and deployed using *component-based* approaches, where applications are modeled and realized as assemblies of software entities (components) that provide well-defined services either to a client program (which can in turn be a component) or to other components. Besides their well-defined service interface, components are opaque: They do not make internal implementation details available to their environment. Component systems typically also provide component isolation: Where not required by the component interface, the behavior of a component does not depend on the state, or even presence of another compo-

This dissertation follows the style of *IEEE Transactions on Automatic Control.*

nent. This isolation requirement allows for a *de-facto* de-coupling of the component implementation from the communication infrastructure among components. (This is illustrated by a number of component systems allowing for interception in their calling mechanisms.) Similarly, given a sufficiently effective communication infrastructure, the components can be decoupled from their execution platform: Components can migrate and execute in a location-independent fashion. In this work we study means to support application adaptability and survivability through on-the-fly reconfiguration at component level (Figure 1). We support on-the-fly application reconfiguration by migrating software components.

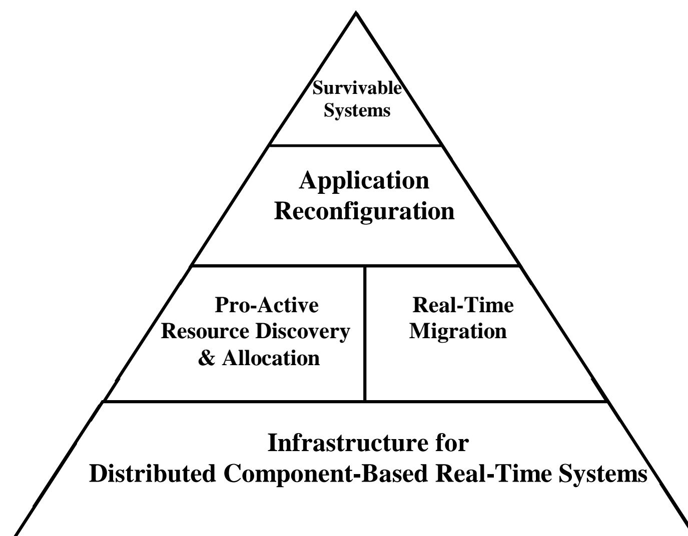

Fig. 1. Middleware for Survivable Systems

The ultimate purpose of migrating active software components is to increase the adaptability and – ultimately – the self-manageability of the application. The ability to perform software component migration increases application-survivability not only in case of external attacks, but also in case of unexpected run-time events that affect

running applications. For example, a change in the local resource allocation policy or in the security policy may require a relocation of some software components running on the local host over to neighboring hosts that have resources to spare or operate at the required security levels, respectively. At that time, on-the-fly application reconfiguration improves application-survivability.

If reconfiguration is performed pro-actively, this can benefit security as well, following from the observation that it is generally harder to attack a moving target rather than an immovable one. Therefore we can increase application-survivability further by migrating software components in a way that cyber attackers cannot easily locate and track.

While the key in supporting these capabilities is *efficient migration* of components, *resource management* is critical for providing effective survivability of the distributed applications. To make matters worse, in the type of applications described above, both resource availability and resource requirements can fluctuate widely: As nodes in the system come under attack, resources on these systems become unavailable. At the same time, components on these nodes migrate, and so further change the resource availability across the system. Since resource availability in the system varies so quickly, resource tracking schemes get easily overwhelmed. We therefore resort to resource *discovery* as a resource availability *estimation* scheme. In other words, we resort to resource discovery whenever the effective resource availability is not known with sufficient accuracy at the time of the resource request.

In real-time applications, the overhead of component migration must be carefully taken into consideration in three ways: First, sufficient resources must be allocated to the component before migration in order to have enough slack available for applications when a migration becomes necessary. Next, the migration decision and the migration execution must be low-overhead and have a low latency in order to

not unduly affect other real-time applications. Finally, sufficient resources must be available at the new host to meet the timing requirements of the migrated component as well as those of other components. The local host needs to be aware of candidate hosts at any time.

In this work we use a server-centric approach for scheduling, admission control, and migration management. That enables the isolation of the components from clients in terms of guaranteeing real-time properties of the exported services of the components. The server-centric approach adopts *component declared* real-time property model instead of *client propagated* real-time property model for the exported services: The components keep the information of real-time properties for the exported services rather than inheriting those from clients in the component declared real-time property model. As a result, clients cannot affect how the workload is executed on the component, and so temporal component isolation is provided.

In this thesis we propose a methodology for providing real-time capabilities to component-based, on-the-fly reconfigurable, distributed system. We describe how we achieve run-time reconfiguration in distributed real-time Java applications by appropriately migrating components. We also describe how we manage overhead and resource utilization by controlling the parameters of the schedulers at component level. Moreover, we propose a new discovery based resource management protocol, REALTOR, which is based on a combination of pull-based and push-based resource information dissemination. REALTOR supports survivability by allowing the migration of components to safe locations under emergencies.

This thesis is organized as follows. In Chapter II, we describe the architecture of a reconfigurable component-based system. We describe the component model, the task model, and the workload model. We also describe our admission control policy. In a number of following chapters, we elaborate on the realization of such

a real-time reconfigurable component-based systems in form of the Real-Time Agile Objects system. More specifically, Chapter III describes the Agile Objects system. We also describe our server-centric approach to component management. Chapter IV describes the design, implementation, and experimental evaluation of the necessary infrastructure needed to make a component-based system (Java RMI in our case) real-time capable. Support for component mobility is described and experimentally evaluated in Chapter V. In Chapter VI, we describe our resource discovery and allocation protocol, REALTOR. We also analyze the message overhead of REALTOR and provide a performance study of REALTOR in Chapter VI. Finally, conclusions and future work are given in Chapter VII.

CHAPTER II

ARCHITECTURE

Component-based systems design supports the development of software systems largely by assembling components that have previously been developed for integration. Reusability, one of the benefits obtained with component-based systems design, has contributed to the great success of this methodology in software systems development. The use of reusable and Commercial-Off-The-Shelf (COTS) components allows for application developers to deliver new systems to customers at less cost and more speed in comparison with more traditional approaches for developing systems, typically from scratch.

In order for components to be reusable, they must satisfy at least three characteristics, which are isolation, opaqueness, and composability: By *isolation* we mean that a component can be well separated from the execution environment and from other components. The behavior of a component therefore is independent of the underlying frame-work, from the state of other components, or, even from the presence of other components. By *opaqueness* we mean that a component conceals its implementation details from other components. The opaqueness leads clients of the component not to depend on the implementation details that are likely to change. By *composability* we mean that a component has a self-contained function unit with well-defined interfaces in order to be composable with other components.

A number of well known standards exist for component-based systems, such as the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [2], Microsoft's Component Services (COM+) [3], and Sun Microsystems' Enterprise JavaBeans (EJB) [4]. These standards provide interfaces to plug together independent components from different suppliers to suit the integra-

tion requirements of customers and even to interoperate across different execution environments.

A number of efforts have extended component-based systems to support real-time applications, for example with means to specify any real-time properties such as worst-case execution times and deadlines of services. The TAO project [5] has implemented a CORBA that preserves the priority levels of calls across component boundaries. Stankovic *et al.* [6] have proposed an approach for building embedded systems software through component-based techniques. Their VEST toolkit provides a rich set of dependency checks to support distributed embedded system development via components. The specification for real-time CORBA has also been proposed by an OMG working group [7, 8]. The real-time CORBA specification extends standard CORBA with features for management of CPU, network, and memory resources. It allows to use either server declared or client propagated model for fixed priority scheduling between client and server. All these real-time extensions provide very limited isolation across components in the temporal domain: TAO, for example, relies on static priority scheduling with priority inheritance across components. The temporal behavior of individual components therefore depends on the behavior of components, specifically on that of higher-priority ones. Due to the priority inheritance, it depends on the execution of remote components as well, for example when a remote high-priority component triggers the execution of methods provided by another local component. Insufficient temporal isolation renders admission control cumbersome and significantly complicates component migration.

In the following, we describe how we provide predictable end-to-end latencies for component services and isolation across components by adopting guaranteed-rate scheduling and on-the-fly reconfiguration.

A. A Model for Distributed Component-Based Applications

We model a distributed component-based application as a collection of n *clients*, C_1, C_2, $\cdots$, C_n and m *components*, A_1, A_2, $\cdots$, A_m. Each component A_i exports a number k_i of *remote methods*, RM_{i1}, RM_{i2}, $\cdots$, RM_{ik_i}.

We assume that clients execute outside of the reach of the resource control of the component-based system. We therefore partition the execution environment into a *client execution environment* and a *component execution environment*. We do not further consider the client execution environment.

The component execution environment consists of h hosts, H_1, H_2, $\cdots$, H_h on which components can be located. We assume a uniform processing environment, where each host H_ℓ has a relative speed Δ_ℓ. A method that takes ε time units to execute on a reference host, takes $\frac{\varepsilon}{\Delta_\ell}$ on a host with relative speed Δ_ℓ.

B. Task Model

We assume that remote methods are invoked synchronously: The execution on the caller is temporarily suspended while the thread of control transfers to the component with the remote method. Upon completion of the remote invocation, the thread of control migrates back to the caller, which in turn resumes execution.

We model the workload in our system as a set of *tasks*. Each task consists of a sequence of invocations to a remote method by one or more clients. We model each task T_i a sequence of jobs, $J_i^{(1)}$, $J_i^{(2)}$, $\cdots$, $J_i^{(k)}$, where the execution of each job is triggered by an invocation of a remote method by a client. Each job J_i of a task T_i consists of the same (possibly nested) sequence of invocations of remote method invocations on one or more components. Therefore, the workload of each job J_i is described by sum of the workload of the invoked remote method invocations on one

or more components.

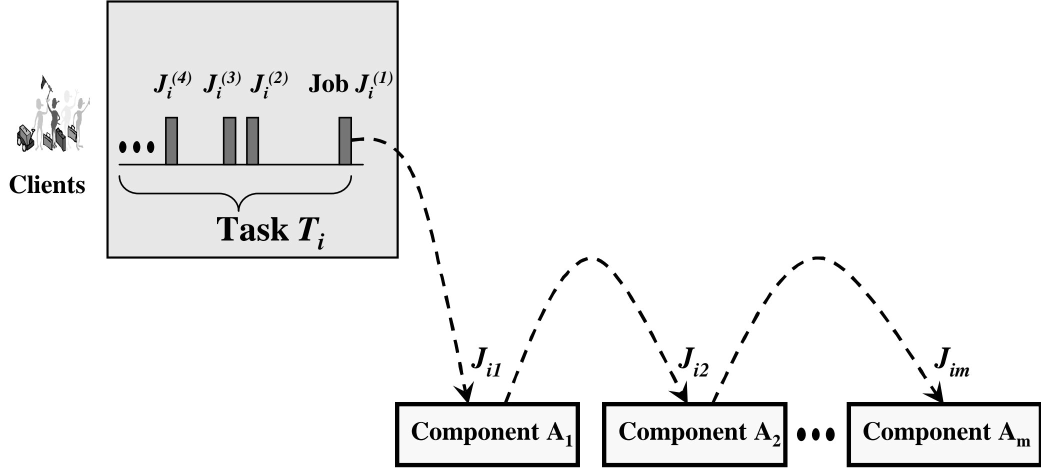

Fig. 2. The Task Model

Figure 2 illustrates this relation between tasks, jobs, and components: In this case $J_i^{(1)}$ is the first in a sequence of jobs of Task T_i. The execution of the job is triggered by the invocation of a remote method of Component A_1 by some client. The execution of this remote method in turn invokes a remote method on Component A_2, and so on.

We provide real-time guarantees in form of deadline guarantees to remote method invocations. We say that we *guarantee a deadline* D_i to a task T_i if we guarantee that every job is completed by at most D_i time units after it has been invoked. In other words, the maximum response time of the remote method is bounded by D_i. The real-time literature distinguishes between *hard* real-time guarantees, where the designer must prove that the deadline requirements are met for all jobs in the system, and *soft* guarantees, where this proof is not required. Hard real-time guarantees are

contingent upon the real-time capabilities of the underlying Operating System (OS) and runtime environment, well-known number of clients, well-behaved clients and well-known worst-case execution time of each method in every component. If one of the above conditions is not satisfied, our systems provides soft real-time guarantees at best.

When individual clients put undue load on the system, that is, when they invoke the remote method too frequently or when the method execution times exceed the worst case, the execution of the task can adversely affect the real-time guarantees for other tasks. This can be prevented by appropriate timing isolation. By *timing isolation* we mean that the worst-case response time of jobs in a task does not depend on the processor-time demands of other tasks. To provide timing isolation at scheduling level, one can use guaranteed-rate schedulers. Examples of such algorithms are the Constant Utilization Server, the Total Bandwidth Server, and the Preemptive Weighted Fair-queueing Server [9]. We have chosen the Total Bandwidth Server for our guaranteed-rate scheduling over the other two for two reasons: (1) The Total Bandwidth Server outperforms the Constant Utilization Server in terms of utilizing background time not used by periodic tasks. (2) The proportional share scheduling algorithms, such as the Preemptive Weighted Fair-queueing Server, make no QoS guarantees if the sum of total weights grows very large [10].

C. The Workload Models

A very common task model in real-time systems is the periodic model, where task invocations are assumed to arrive with a given inter-arrival time, *i.e.*, the period. For strictly periodic tasks, the inter-arrival times are always equal to the task period. For non-strictly-periodic tasks, the period is equal to the minimum inter-arrival time

of two invocations. Most scheduling and admission control schemes are robust to variations in the inter-arrival time: If the scheme works for a periodic task, if does so for any non-periodic task with minimum inter-arrival time that are at least as long as the period as well. Whenever the task invocations are bursty, that is, the minimum inter-arrival time is significantly shorter than the average, treating tasks as periodic is very inefficient, as the admission control reserves resources for workload that does not materialize at run-time. We expect task invocations to be very bursty in the system we envision, for three reasons: First, any task may consist of invocations from more than one client, thus resulting naturally in a bursty invocation pattern. Second, no explicit policing mechanism exists to shield the system from bursty invocations. Third, even if invocations from clients were periodic, cascaded invocations to subsequent components would still be jittered by execution and scheduling, and so would be bursty. As a result, we model task arrival as sporadic: The detailed arrival time of the nest invocation is unknown *a priori*, but worst-case execution time and deadline become known upon arrival. The invoked jobs of the sporadic task are scheduled by using a Total Bandwidth Server. Specifically, a Total Bandwidth Server is configured for each remote method of every component. Every Total Bandwidth Server then allocates and controls the amount of CPU time that is consumed for execution of the assigned remote method on the component. As a side effect, it shapes the service interval between successive jobs that invoke the same remote method of the same component.

In summary, the use of guaranteed-rate schedulers - in our case the Total Bandwidth Server - allows us to uniformly schedule periodic and sporadic tasks. Similarly, this allows for a simple, utilization-based, admission control for both types of tasks.

The response time of each job J_i of Task T_i is what the client experiences as latency. If we do not take into account communication delays, the relative deadline

Table I. Notations for Workload Models

Notation	Description
RM_{mx}	remote method RM_x on Component A_m
rm_{mx}	worst-case execution time of RM_{mx}
LM_{my}	local method LM_y on Component A_m
lm_{my}	worst-case execution time of LM_{my}
U_{ij}	utilization allocated by Total Bandwidth Server to RM_{ij}
$LM_{my} \in RM_{mx}$	LM_{my} is invoked during the invocation of RM_{mx}
$RM_{ny} \in RM_{mx}$	RM_{ny} is invoked during the invocation of RM_{mx}

of Job J_i on the component must bound the maximum latency, or maximum response time, of Job J_i. If a guaranteed-rate scheduler (in the following we limit the presentation to the Total Bandwidth Server) handles the jobs for a given component, and if appropriate admission control ensures that the server is not overloaded, then the relative deadline of Job J_i can be calculated by the following:

$$\begin{aligned}&\text{Relative Deadline of Job } J_i\\ &= \frac{\text{Workload of Job } J_i}{\text{Utilization of Total Bandwidth Server for Job } J_i}.\end{aligned}$$

Therefore, if we have a correct workload model for each job J_i on a component, Job J_i can be guaranteed to meet its deadline through appropriate admission control. The following shows how our workload models look like. Table I describes the notations used for our workload models.

1. Invocations of Local Methods

We can model the job J_i of Task T_i as an invocation of a remote method RM_{mx} on a component A_m, where the execution of the remote method RM_{mx} invokes only local methods on Component A_m.

In this case, the workload of Job J_i is the sum of the worst-case execution time of each local method on Component A_m. Therefore, the workload of this kind of Job J_i is calculated by the following:

$$\begin{aligned} \text{Workload of Job } J_i &= rm_{mx} \\ &= \sum_{LM_{mk} \in RM_{mx}} lm_{mk}. \end{aligned}$$

Figure 3 shows an example of Job J_i of Task T_i. Job J_i invokes the remote method RM_{mx} on Component A_m. The execution of the remote method RM_{mx} invokes three local methods on Component A_m. In this example, Component A_m is the only component in the execution of Job J_i.

2. Invocations of Local and Single-Chained Remote Methods

We can model the job J_i of Task T_i as an invocation of a remote method RM_{ny} on a component A_n, where the execution of the remote method RM_{ny} invokes local methods on Component A_n, and it also invokes one or more other single-chained remote methods on one or more other components. By *single-chained* remote method we mean that the execution of the remote method invokes only local methods.

In this case, the workload of Job J_i is the sum of the worst-case execution time of each local method on Component A_n and the maximum response time for getting the result from each single-chained remote method. Therefore, the workload of this

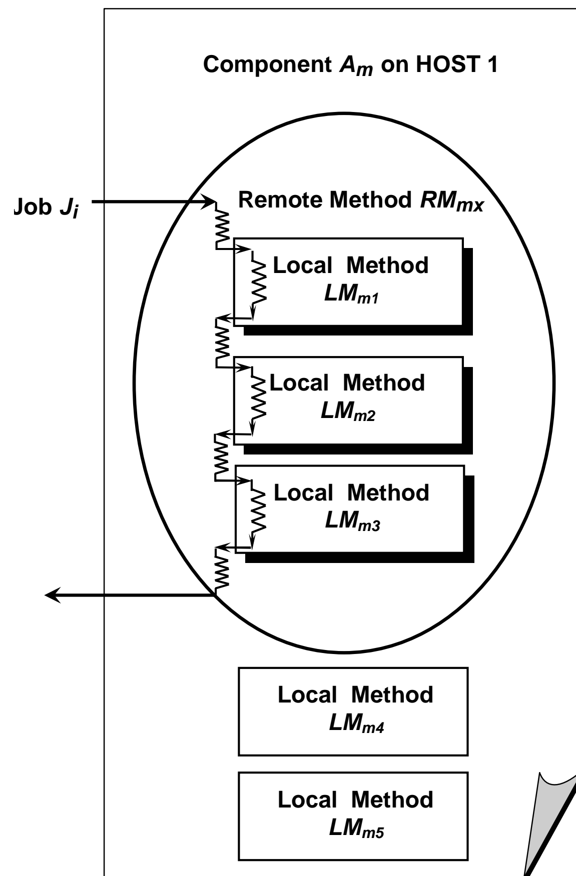

Fig. 3. Invocations of Only Local Methods

kind of Job J_i is calculated by the following:

$$\begin{aligned}\text{Workload of Job } J_i &= rm_{ny} \\ &= \sum_{LM_{nk} \in RM_{ny}} lm_{nk} + \sum_{RM_{ij} \in RM_{ny}} \frac{rm_{ij}}{U_{ij}} \\ &= \sum_{LM_{nk} \in RM_{ny}} lm_{nk} + \sum_{RM_{ij} \in RM_{ny}} \left(\frac{1}{U_{ij}} \times \sum_{LM_{it} \in RM_{ij}} lm_{it}\right).\end{aligned}$$

Figure 4 shows an example of Job J_i of Task T_i. Job J_i invokes the remote method RM_{ny} on Component A_n. The execution of the remote method RM_{ny} invokes three local methods on Component A_n and the single-chained remote method RM_{mx}

on Component A_m. Subjob J_{im} denotes the execution of the single-chained remote method RM_{mx} on Component A_m.

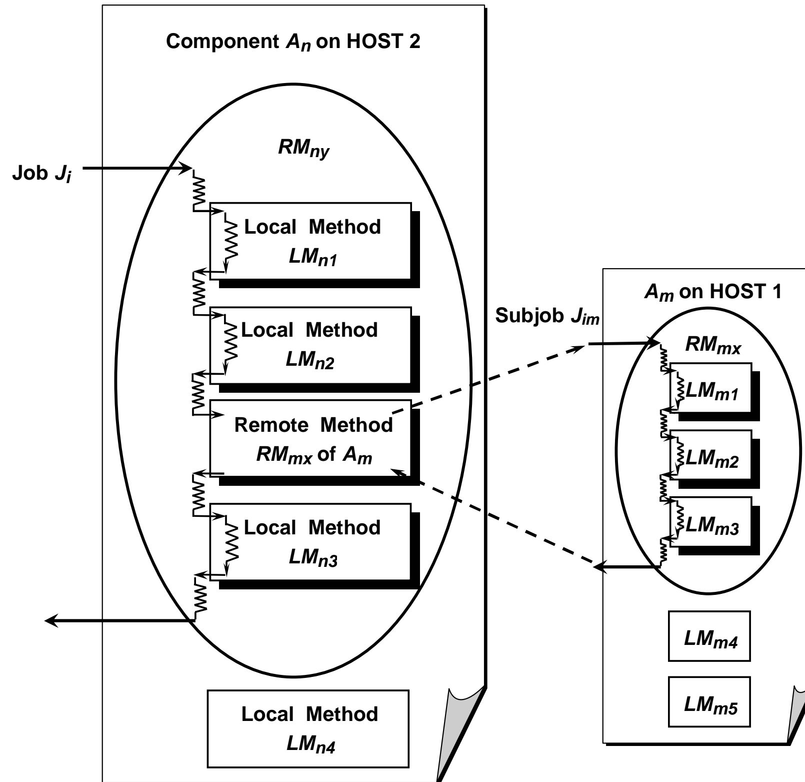

Fig. 4. Local Methods and a Single-Chained Remote Method

3. Invocations of Local and Multi-Chained Remote Methods

We can model the job J_i of Task T_i as an invocation of a remote method RM_{pz} on a component A_p, where the execution of the remote method RM_{pz} invokes local methods on Component A_p, and it also invokes one or more other multi-chained remote methods on one or more other components. By *multi-chained* remote method we mean that the execution of the remote method in turn invokes one or more other

remote methods on one or more other components.

In this case, the workload of Job J_i is the sum of the worst-case execution time of each local method on Component A_p and the maximum response time for getting the result from each multi-chained remote method. Therefore, the workload of this kind of Job J_i is calculated by the following:

Workload of Job J_i

$$
\begin{aligned}
&= rm_{pz} \\
&= \sum_{LM_{pk} \in RM_{pz}} lm_{pk} + \sum_{RM_{ij} \in RM_{pz}} \frac{rm_{ij}}{U_{ij}} \\
&= \sum_{LM_{pk} \in RM_{pz}} lm_{pk} + \sum_{RM_{ij} \in RM_{pz}} [\frac{1}{U_{ij}} \times (\sum_{LM_{it} \in RM_{ij}} lm_{it} + \sum_{RM_{vw} \in RM_{ij}} \frac{rm_{vw}}{U_{vw}})].
\end{aligned}
$$

Figure 5 shows an example of Job J_i of Task T_i. Job J_i invokes the remote method RM_{pz} on Component A_p. The execution of the remote method RM_{pz} invokes three local methods on Component A_p and multi-chained remote method RM_{ny} on Component A_n. Subjob J_{in} denotes the execution of the multi-chained remote method RM_{ny} on Component A_n. The execution of the remote method RM_{ny} also invokes single-chained remote method RM_mx on Component A_m, Subjob J_{im}.

D. Admission Control Policy

Before new components are created and installed, an admission control step has to make sure that sufficient computing resources on the host can be allocated to the remote methods of the new component without affecting other components on the host. The same holds for components that are migrating to a host from elsewhere in the system. In the following discussion, we focus on CPU resources only. Other resource requirements, such as opened files, memory and network bandwidth are not

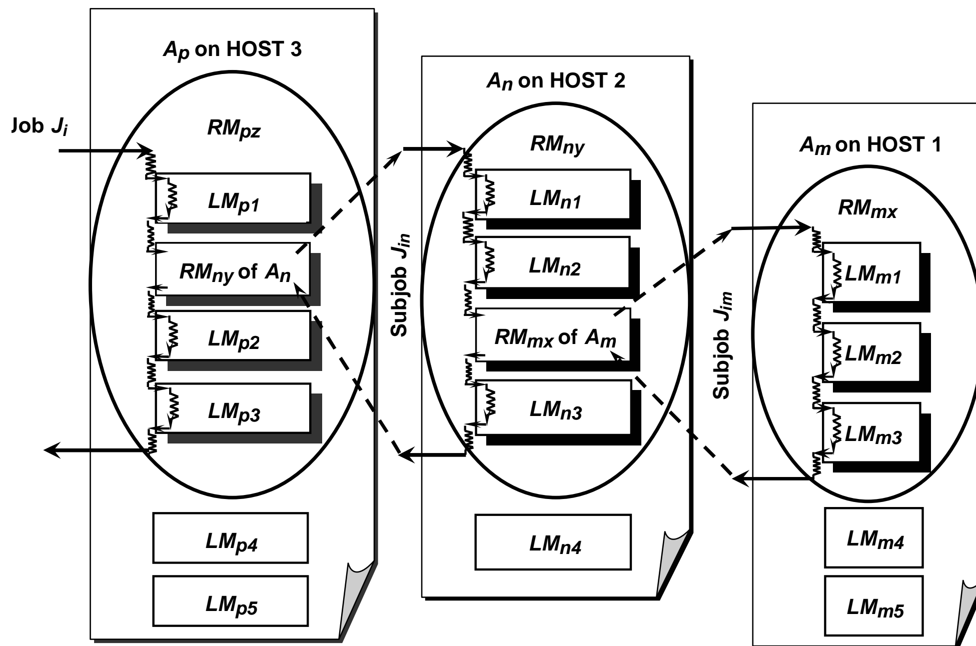

Fig. 5. Local Methods and a Multi-Chained Remote Method

considered for admission feasibility tests.

The admission control mechanism must rely on an accurate description of the workload parameters. For this, worst-case execution times of remote methods are determined either during system design or system configuration. Similarly, the utilizations allocated to components and their remote methods are defined during system configuration.

CHAPTER III

REAL-TIME AGILE OBJECTS SYSTEM

In the following, we will use the Agile Objects System [11] as platform for developing the mechanisms and resource allocation schemes to support real-time guarantees in dynamically reconfigurable distributed component-based applications. Components in the Agile Objects System are capable of migrating frequently, which provides them with *location elusiveness* [12]. The latter is greatly beneficial for both *survivability*, as the application is able to quickly reconfigure during attacks, and for *information assurance*, as the location and tracking of critical components become significantly more difficult for an attacker. In addition, component migration allows for more *flexible* response to resource overload for QoS sensitive applications: if a newly arriving task is expected to miss its deadline because of overload at a host, we can simply migrate the component that is supposed to serve the task to another host, where enough resources are available.

Figure 6 shows a high-level diagram of the Agile Objects System, with low-level mechanisms for migration and scheduling at the bottom, and higher-level ones, such as resource allocation, at the top:

- High-level resource management is performed by REALTOR (REsource ALlocaTOR). The main objective of REALTOR is to provide *proactive* resource discovery for fast migration [13].
- *Admission Control* is in charge of admission decisions during component instantiation and migration. As REALTOR monitors the resource status across hosts, the admission control during migration can be very light-weight and can be performed concurrently to the migration properly.
- The management of CPU resource is greatly simplified by the use of guaranteed-

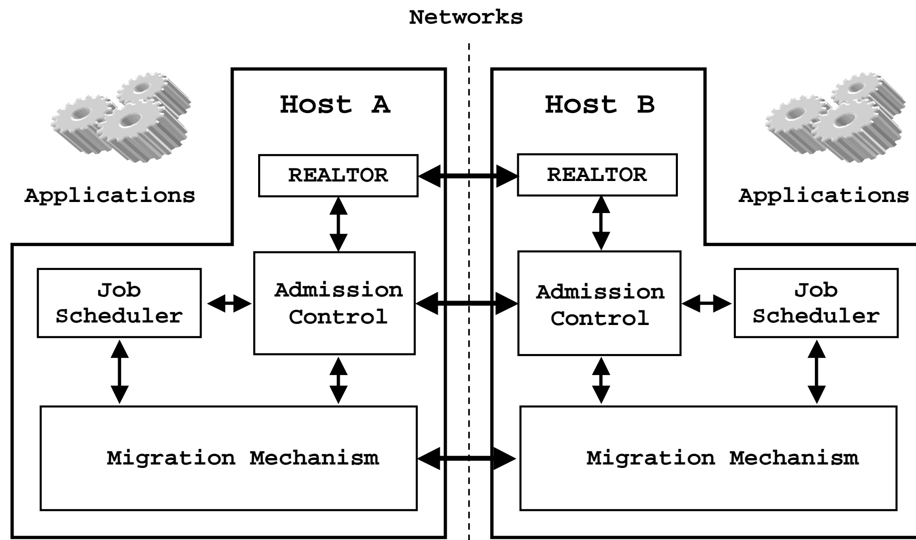

Fig. 6. Software Components of Agile Objects System

rate scheduling in the nodes. Guaranteed-rate schedulers ensure a minimum amount of CPU bandwidth to each QoS sensitive workload. This greatly reduces the admission control overhead, which becomes a simple utilization test, and available CPU resource can be directly measured in terms of unallocated utilization. The current implementation uses a Total Bandwidth Server.

- The mechanics of component migration is handled by the *migration subsystem*. During migration, the component state is moved, the necessary code and libraries at the destination are updated, and service-access points are transferred. In addition, the naming service is updated to reflect the new location of the component.

The realization of Agile Objects System is based on an extension of the Real-Time Specification for Java (RTSJ) [14], with an Earliest Deadline First (EDF) scheduler [9], Total Bandwidth Servers [9] and real-time Remote Method Invocation (RMI) [15]. The guaranteed-rate scheduling at the nodes allows for an accurate definition

of resource requirements during design and deployment time, and thus eliminates the need for cumbersome resource reallocation mechanisms during run-time and for priority inheritance extensions to RMI, such as, for example, in [16].

A. A Model for Agile Objects

The agile objects model naturally maps to the workload model described in Section C of Chapter II: Each Agile Object A_i exports a number k_i of remote methods, RM_{i1}, RM_{i2}, $\cdots$, RM_{ik_i}. Each Agile Object A_i also has a so-called *Listening Thread* LT_i that listens for incoming requests from clients on behalf of the agile object. Each *Listening Thread* LT_i spawns a so-called *Worker Thread* WT_{ijq} whenever the Listening Thread LT_i accepts an incoming request for remote method RM_{ij}. As a result, each Listening Thread LT_i creates a number of Worker Threads WT_{i11}, WT_{i12}, $\cdots$, WT_{i1v}, WT_{i21}, WT_{i22}, $\cdots$, WT_{i2w}, $\cdots$, WT_{ik1}, WT_{ik2}, $\cdots$, WT_{iky}, for remote methods RM_{i1}, RM_{i2}, $\cdots$, RM_{ik}. Each remote method RM_{ij} of an Agile Object A_i has its own Total Bandwidth Server TB_{ij}.

B. A Server-Centric Environment for Real-Time Java RMI

Java RMI system is a well-known middleware for distributed systems, so that Java RMI system has been used for making distributed real-time systems. As shown in Figure 7, our Agile Objects System is built on top of Java.

As Java was not originally designed with embedded and real-time applications in mind [17], it comes to no surprise that a number of limitations have been identified with Java's applicability in the real-time domain. Primarily, the non-deterministic behavior of the garbage collection mechanism could interrupt the execution of applications for unpredictable intervals of time [18]. Moreover, the Java Virtual Machine's

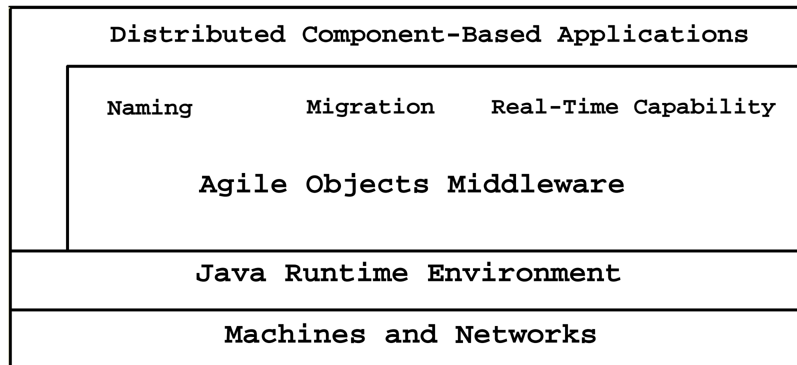

Fig. 7. Agile Objects System Architecture

thread scheduling relies on the scheduler of the host operating systems and the details of the scheduling requirements (such as priorization, resource access, *etc.*) are only very sketchily defined. To overcome these and other limitations, the Real-Time for Java Expert Group has proposed RTSJ, an extension to the Java specification and Java Application Programming Interfaces (APIs) in order to improve the time predictability of Java programs. The RSTJ focuses on seven functional areas: thread scheduling and dispatching, memory management, synchronization and resource sharing, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, and physical memory access. An area that was (intentionally) ignored in the RTSJ is that of distributed execution of real-time programs. In particular, any limitations within the Java RMI (for example lack of priority inheritance across threads that handle remote invocations) are not addressed in the RTSJ. The *Distributed Real-Time Specification for Java (DRTSJ)* [19] will introduce the Distributed Real-Time Java RMI. But the DRTSJ and its reference implementation have not yet been released as of summer 2004.

We propose a server-centric environment for real-time Java RMI system in order to support component isolation of the Agile Objects System. In our proposed

environment, we use what we call a *server-declared* model for real-time properties of components: The server components keep information for meeting real-time guarantees, instead of inheriting the real-time properties from clients. The server-declared model ensures that the resource management of real-time RMI servers can be decoupled from the client execution environment. As a result, we then can simplify admission control because we can make all admission test mostly locally.

CHAPTER IV

REAL-TIME INFRASTRUCTURE

A. Introduction

We adapt Java as the underlying platform for our reconfigurable distributed component-based systems. Java [20] has been used for developing distributed systems because of its simplicity, security and portability; object oriented methodologies from Java increase productivity; Java's safe run-time environment enables controlled access to system's critical resources; Java's byte-code allows Java to be platform independent and enables portability. In fact, Java has intrinsic features that support distributed systems, such as threads, concurrency-control mechanism and a serialization protocol [21], which eases component migration. The Java Object Serialization protocol for Java RMI supports point-to-point flows of data. Furthermore, the Java Object Serialization protocol implements a message passing mechanism by delivering messages as object instances. Java Remote Method Invocation (RMI) [15] provides distributed object systems with the control flow mechanisms required for method invocations on remote hosts. Both Java and Java RMI were not designed with support for real-time systems in mind. The recently released Real-Time Specification for Java (RTSJ) [14] provides standard real-time Java APIs for developing systems that have timing requirements. Therefore, there is a possibility of integrating Java RMI with the RTSJ. For this, the Distributed Real-Time Specification for Java (DRTSJ) [19] will propose for Java RMI to make it be a distributed real-time system model in a near future.

We propose a server-centric environment that allows for the isolation of real-time server components. In our proposed environment, the real-time server components keep information for meeting real-time guarantees using server-declared model instead

of inheriting real-time properties from clients. This component isolation allows server execution environment to be decoupled from client execution environment. Therefore, our server-centric real-time Java RMI system can simplify admission control and help our resource discovery mechanism to find a host with required resources in order to support component migration for achieving high application-survivability.

B. Related Work

Originally, Java was designed for Internet Web applications. Therefore, high-performance was not considered as one of Java's features. However, recent research projects have proven that Java applications' performance can be close to that of applications written in C language [22, 23]. While "fast" need not necessarily be equaled to "real-time," the increased performance of Java platforms has made them attractive for embedded and real-time applications. The Real-Time Specification for Java (RTSJ) [14] extends *The Java Language Specification* [20] and *The Java Virtual Machine Specification* [24] and provides application programming interfaces for real-time capabilities. Since the RTSJ has been developed in collaboration between academic and industry experts, we expect a long lifespan. In taking advantage of Java's ability to reduce efforts for development and porting across platforms, the RTSJ can be used by many industries for developing real-time and embedded systems. The RTSJ, coupled with a real-time operating system, leverages the capability of Java for developing real-time systems in the sense that it separates hard real-time, soft real-time and non-real-time threads. The RTSJ Reference Implementation (RI) from TimeSys [25] and jRate [26] have implemented the RTSJ.

However, both the RTSJ and its implementations do not support real-time capabilities for Java Remote Method Invocation (RMI) [15]. There have been several

efforts for integrating the RTSJ with the Java RMI mechanism, for example Jensen *et al.* [19], Clark *et al.* [27], and Wellings *et al.* [28]. Jensen *et al.* [19] propose the Distributed Real-Time Specification for Java (DRTSJ). The DRTSJ addresses predictability of end-to-end timeliness in dynamic distributed real-time systems. In distributed real-time computing systems, end-to-end timing constraints can be maintained over multi-node applications under each node's current environment. The DRTSJ has been designed for general cases of dynamic distributed real-time computing systems. Therefore, it is not easy to know each node's current environment *a priori*, such as latency properties of OS and network infrastructure, and system resource utilization. To achieve end-to-end multi-node timeliness, the properties of each multi-node application behavior's timeliness need to be propagated to the resource managers of the OS and the Java Virtual Machine (VM) [24] on each node.

The DRTSJ suggests three ways to integrate Java RMI with the RTSJ: The first approach does not expect timely delivery of messages and inheritance of scheduling parameters between real-time Java threads and remote objects. It therefore suggests no changes to the RTSJ and Java RMI. The second approach expects timely delivery of messages and inheritance of scheduling parameters between real-time Java threads and real-time remote objects. It therefore suggests extensions required to Java RMI but no extensions required to the RTSJ or the real-time Java VM. Borg *et al.* [29] followed this approach in developing a frame-work for real-time RMI for the RTSJ. They extended the RMI to support timely invocation of remote objects. The third approach suggests extensions required to Java RMI, the RTSJ and the real-time Java VM to support *distributed thread functionality*. In distributed thread models, a distributed thread has a system-wide ID and the feature of transparent propagation of its properties along its execution environments.

C. Standard Java RMI Problems for Providing Real-Time Capability

1. Sun's Java RMI Implementation

J2SE (Java 2 Platform, Standard Edition) RMI implementation provides three layers to implement transparent method invocations for remote objects:

First, the *Stub Layer* works as a proxy for the client side. The signatures of methods that client stubs provide are identical to the signatures of methods defined in the remote interfaces of remote objects. The stub classes are automatically generated by an RMI stub compiler. The classes are downloaded from a naming server of RMI system when client applications locate the remote objects. The stub layer marshals and unmarshals both arguments and returned values between client and server hosts. Figure 8 shows the Unified Modeling Language (UML) diagram of Sun J2SE RMI related classes at the client [30]. The *ClassImpl_Stub* classes are produced by the stub compiler, which takes the actual remote object implementation classes, *ClassImpl*, as input. All the *ClassImpl_Stub* classes should extend class *RemoteStub* since class *RemoteStub* provides the frame-work for supporting remote reference semantics.

Second, the *Remote Reference Layer* supports semantics of reference and invocation (for instance, unicast and multicast) and translates between local and remote object references using remote-object tables. Interface *RemoteRef* defines the methods for invoking methods to remote objects, and class *RemoteStub* uses interface *RemoteRef* to invoke methods to remote objects. Class *UnicastRef* implements interface *RemoteRef* and supports the semantics of unicast method invocation to remote objects. Class *LiveRef* constructs a live reference to an instance of class *RemoteObject* in another Java virtual machine if the object is not in client's Java virtual machine. Otherwise, class *LiveRef* instantiates a live reference for an instance of class *RemoteObject* in the client's Java virtual machine.

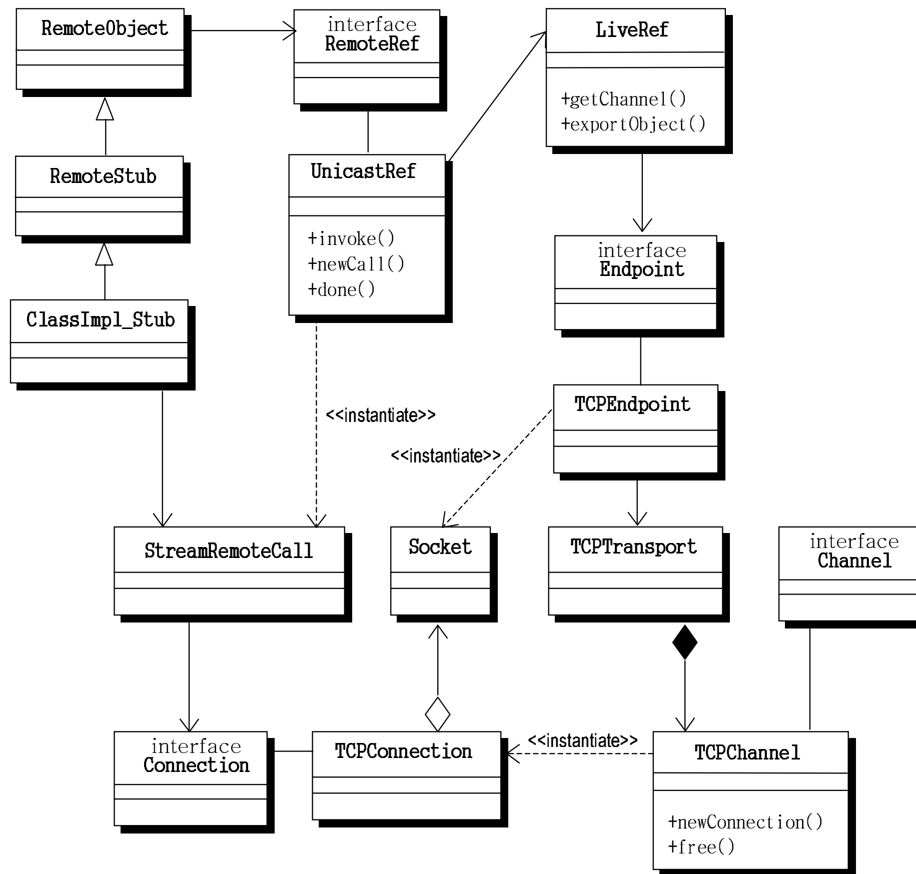

Fig. 8. The UML Diagram of RMI Client Classes

Finally, the *Transport Layer* sets up connections to remote address spaces by using classes *TCPEndpoint*, *TCPChannel* and *TCPConnection*. For TCP, classes *TCPEndpoint*, *TCPChannel* and *TCPConnection* implement methods defined in the transport layer interfaces *Endpoint*, *Channel* and *Connection*, respectively. Interface *Connection* defines methods for transferring data, and interface *Channel* defines methods for managing connections.

Figure 9 shows the Unified Modeling Language (UML) diagram of Sun J2SE's server-side RMI related classes [30]. In the server-side remote reference layer, class *RemoteServer* is used instead of class *RemoteStub*. As a superclass of RMI server

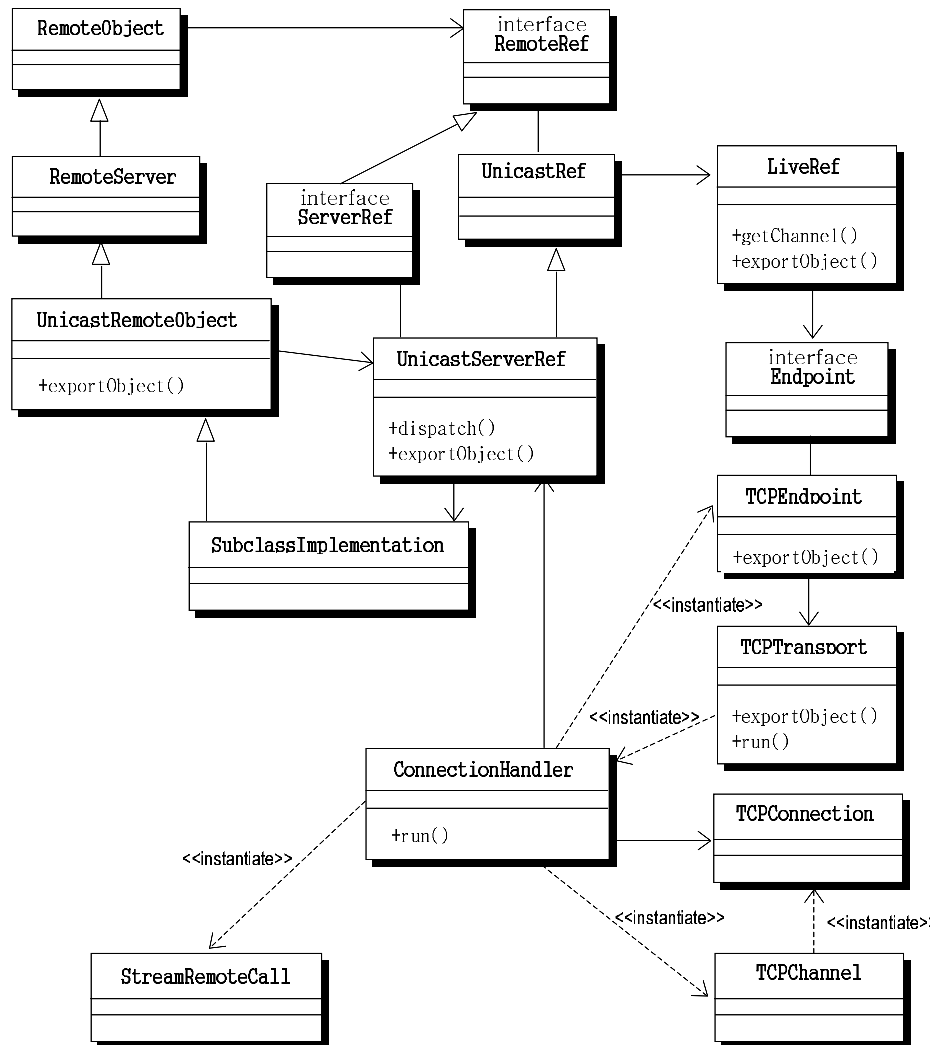

Fig. 9. The UML Diagram of RMI Server Classes

implementations, class *RemoteServer* provides the frame-work for supporting the semantics of remote reference. Class *UnicastRemoteObject* extends class *RemoteServer* and provides method *exportObject()*. The *exportObject()* method makes remote objects available to receive incoming calls from clients by calling method *exportObject()* of class *UnicastServerRef*. Class *UnicastServerRef* also supports server side behavior for remote reference layer. Ultimately, the *exportObject()* method of the instance of

class *TCPTransport* is invoked to export a remote object by referencing the instance of class *LiveRef* mapped onto the remote object. The export of a remote object from an instance of class *TCPTransport* includes creating a listening system thread to handle incoming method requests. The listening system thread executes the *run()* method of class *TCPTransport* for accepting connections to server and launches a non-system thread for each accepted connection. When the system thread spawns a new non-system thread for handling each connection, the *run()* method of class *ConnectionHandler* is executed by the non-system thread. While executing the *run()* method of class *ConnectionHandler*, the *serviceCall()* method of class *TCPTransport* is invoked to service an incoming remote call. The *serviceCall()* method locates and calls the dispatcher object of the required remote object for the incoming remote call. Class *UnicastServerRef* plays the role for the dispatcher object. The *dispatch()* method of the instance of class *UnicastServerRef* for the remote object makes an up-call to the server, class *SubclassImplementation* of RMI server application, and marshals the return result from the up-call method.

2. Design Issues for Providing Real-Time Capable Java RMI

We identified three issues that must be addressed when realizing a real-time capable Java RMI System:

First, as we mentioned above, the export of a remote object involves creating a number of Java threads. One of those threads listens for incoming calls to the exported remote object. The others are worker threads for handling each accepted incoming request separately. To guarantee real-time properties, the Java VM must support real-time capable threads.

Second, clients must be able to propagate real-time timing constraints to the remote object. This object may in turn invoke remote methods on other remote

objects in different Java VMs, thus act as client to other remote objects.

Finally, we consider *open* systems, where we have no control over number and behavior of clients. As a result, invocations can arbitrarily be bursty, due either to relative phasing of client requests, or to the invocation pattern of single clients. One of the features of client arrivals should be aperiodic. There may be bursty arrivals of clients as well. Therefore, how do we provide sporadic real-time tasks of the remote object with real-time guarantees?

In the following sections, we propose our solutions to the three issues listed above.

D. Real-Time Java Threads

Handling of incoming requests in Java RMI is thread-based. As a result, we render RMI processing real-time capable with the use of real-time threads: Whenever an incoming call arrives, the listening thread creates a real-time worker thread for handling the request from the incoming call. The real-time worker thread executes the *run()* method of class *ConnectionHandler* from Figure 9. In order to maintain a high responsiveness to incoming requests, we have three types of real-time threads, which differ by their priorities: (1) Worker threads execute the remote method invocations at their assigned priorities. (2) The EDF scheduling thread handles worker threads, and runs at a priority higher than all of them. (3) The listening thread runs at priority higher than both worker threads and EDF scheduler. In this way, the listening thread executes like an interrupt service routine, which is very short and provides system responsiveness by executing at highest priority.

1. The Creation of Real-Time Threads for Java RMI

Class *sun.rmi.transport.RMIThreadAction* is used to create Java threads in Sun's Java RMI implementation. As we mentioned before, we create real-time worker threads for handling incoming calls to exported remote objects. We have modified the *run()* method of class *sun.rmi.transport.RMIThreadAction* to create real-time threads as worker threads for handling client requests. In the *run()* method, we classify a requested thread into either a listening thread or a worker thread: If it is a listening thread, we create an instance of class *java.lang.Thread*. In addition, we assign a higher priority to the listening thread than the priorities of worker threads and the EDF scheduler. The reason is not to drop bursty client arrivals due to the delay caused by processing other higher priority tasks first. If it is a worker thread, we create an instance of class *javax.realtime.RealtimeThread*. The creation of the instances of class *javax.realtime.RealtimeThread* requires parameters, such as, an instance of class *javax.realtime.SchedulingParameters* and an instance of class *javax.realtime.ReleaseParameters*.

At the time of launching a newly created real-time worker thread, the identity of the invoked method is as yet unknown. The timing parameters for the invocations are therefore not known yet as well, and the timing parameters of the worker thread cannot be correctly instantiated. The EDF scheduler, however, considers the newly created real-time worker thread as the highest priority task regardless of its default deadline. Once demarshalling is performed, and the identity of the invoked method is obtained, the default timing parameters of the newly created real-time worker thread are properly reset according to the requested remote method. The details will be further discussed in the following section.

Figure 10 shows how the RTSJ real-time thread can be associated with Java RMI

system. Initially, the real-time worker threads generated from the listening threads are put into the EDF scheduler's *Ready* queue. The EDF scheduler assigns priorities to real-time threads based on their deadlines.

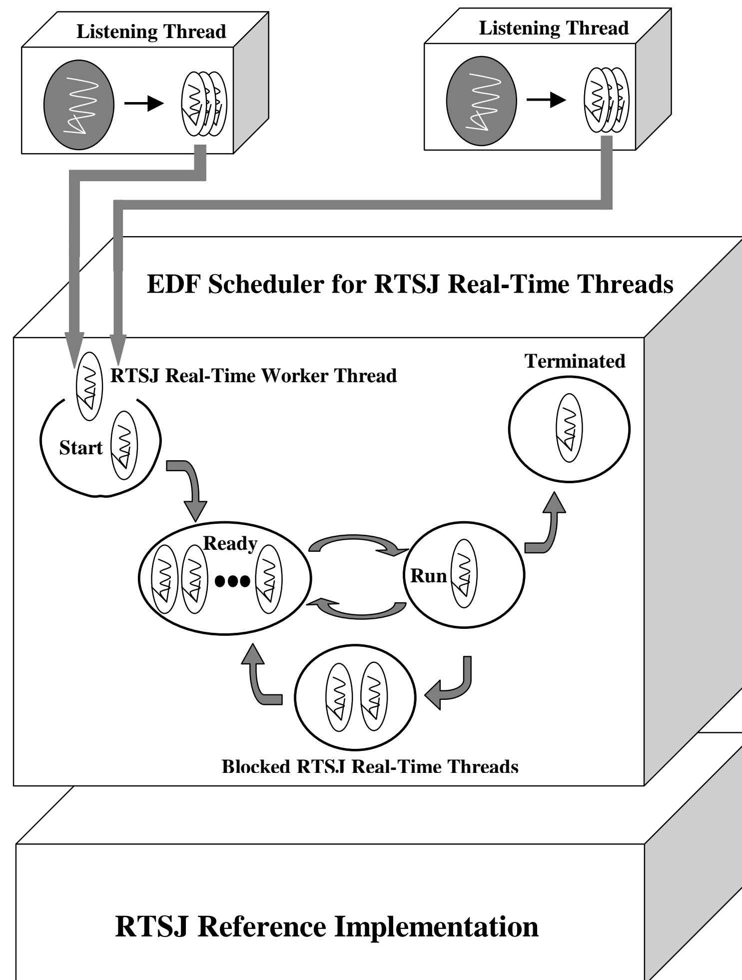

Fig. 10. The Scheduling of Real-Time Worker Threads for Exported Remote Objects

2. The Adjustment of Priorities of Real-Time Worker Threads Based on Admitted Utilization

During the invocation of an exported remote method on the server side, a real-time worker thread dispatches an up-call to the remote object. At this time, we get the object reference of the remote object and the name of the target method. To adjust the attributes of the instance of class *ao.realtor.scheduler.TotalBandwidthParameters* for the real-time worker thread, the workload and utilization of the target method are obtained. Therefore our local admission control component provides the information about the workload and utilization of the target method.

As shown in Figure 11, we set the deadline of the current running real-time worker thread accordingly after getting the workload and utilization of the target remote method from the local admission control component. Once we set the timing parameters of the real-time thread, we wake up the EDF scheduler to reflect the changed deadline of the real-time thread and to reschedule the threads accordingly.

E. A Server-Centric Approach for Preserving Real-Time Timing Constraints

We take a server-centric approach to preserve real-time timing constraints instead of propagating the real-time timing constraints between clients and server components. By *server-centric* we mean that the real-time server components keep information for meeting real-time guarantees instead of delivering and inheriting the scheduling and release parameters of the server components between clients and themselves. According to our task model, those timing constraints are defined as workload, deadline and utilization of each remote method of an exported remote object. The main reason why we choose the server-centric approach is to provide component isolation, which in turn greatly simplifies the admission control needed for component creation

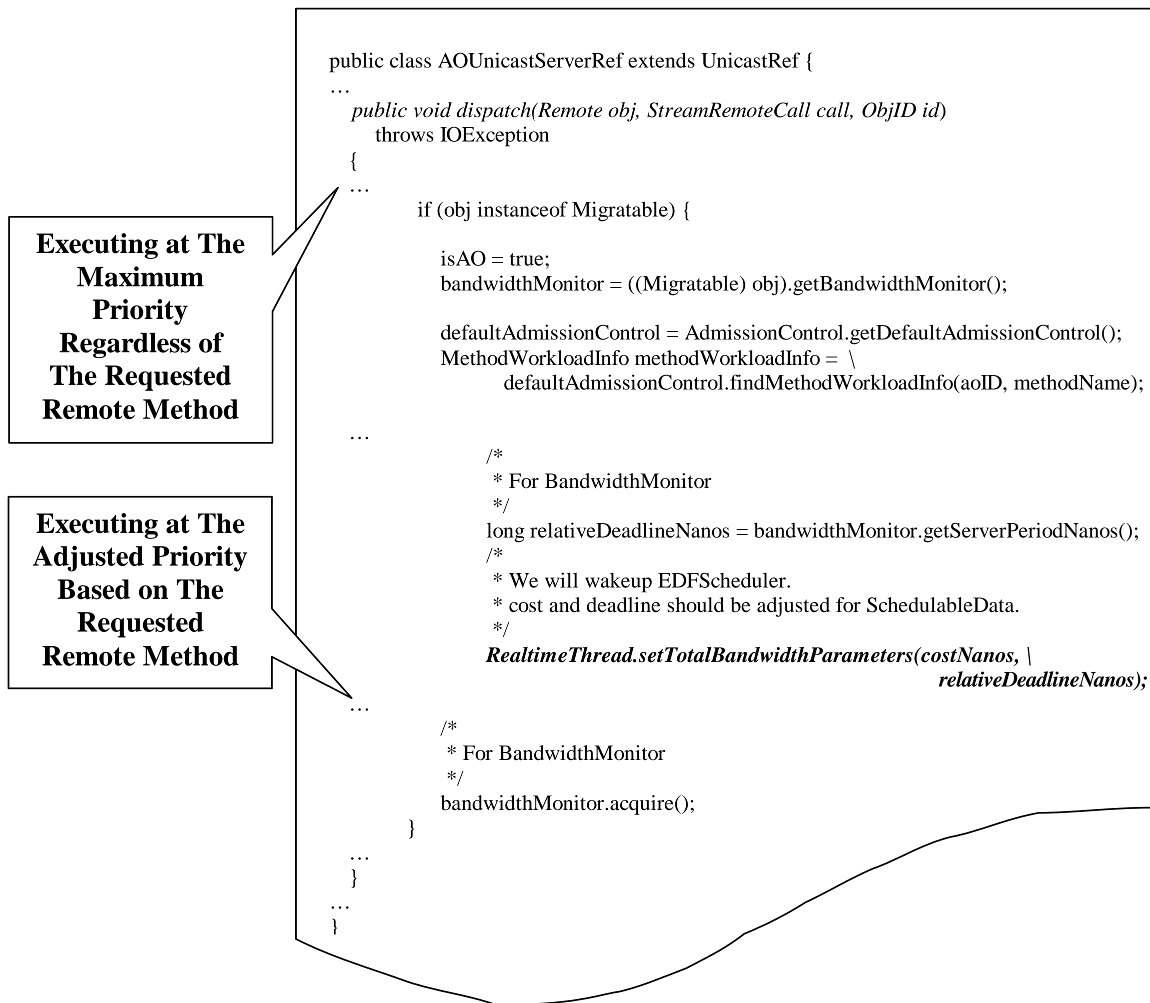

Fig. 11. The Procedure for Adjusting the Priority of an RMI Real-Time Worker Thread

and migration. We use an utilization-based admission control to guarantee real-time properties of migratable agile objects. By *utilization-based admission control* we mean that total utilization reserved for each migratable agile object should be available at the candidate host for migration before actual migration occurs. The server-centric approach also reduces the overhead of remote invocations, as there is no need to exchange timing information as part of the remote invocation at run-time.

F. Guaranteed-Rate Scheduling for Sporadic Real-Time Tasks

The real-time worker threads for handling real-time Java RMI are sporadic in nature: Their arrival is not known *a priori*, but their timing requirements are known upon arrival. There have been studies for scheduling aperiodic and sporadic real-time tasks in deadline-driven real-time systems [9]. There are two popular Bandwidth Preserving algorithms in deadline-driven real-time systems. One is the Constant Utilization Server [9] algorithm and the other is the Total Bandwidth Server [9] algorithm. Total Bandwidth Server, however, shows generally better responsiveness.

1. The Total Bandwidth Server

The Total Bandwidth Server is a periodic server and is defined by two rules, consumption and replenishment rules. In this section we follow J. Liu [9] to describe the operation of the Total Bandwidth Server.

Initially, our Total Bandwidth Server sets the server's execution budget $Budget_{Server}$ and deadline of the server $Deadline_{Server}$ to zero. When a sporadic job with execution time $ExecutionTime_{Client}$ arrives at a time t to a job queue with no backlogged jobs, our Total Bandwidth Server sets $Deadline_{Server}$ to $(max(Deadline_{Server}, t) + ExecutionTime_{Client} / Utilization_{Server})$ and $Budget_{Server}$ equal to $ExecutionTime_{Client}$. When the current sporadic job of the server finishes, and if the server is backlogged, our Total Bandwidth Server sets $Deadline_{Server}$ to $(Deadline_{Server} + ExecutionTime_{Client} / Utilization_{Server})$ and $Budget_{Server}$ equal to $ExecutionTime_{Client}$. When the current sporadic job of the server finishes and if the server is not backlogged, our Total Bandwidth Server does nothing. Our scheduler should take care of the following:

- The scheduler keeps track of the Total Bandwidth Server's budget $Budget_{Server}$.

- When the budget $Budget_{Server}$ reaches to zero, the scheduler suspends the thread of the sever.
- When the server becomes idle, the scheduler suspends the thread of the sever.
- When the budget $Budget_{Server}$ is again ready by the replenishment rules of the Total Bandwidth Server and the server becomes backlogged by arrival of a sporadic job, the scheduler changes the status of the server thread as ready.

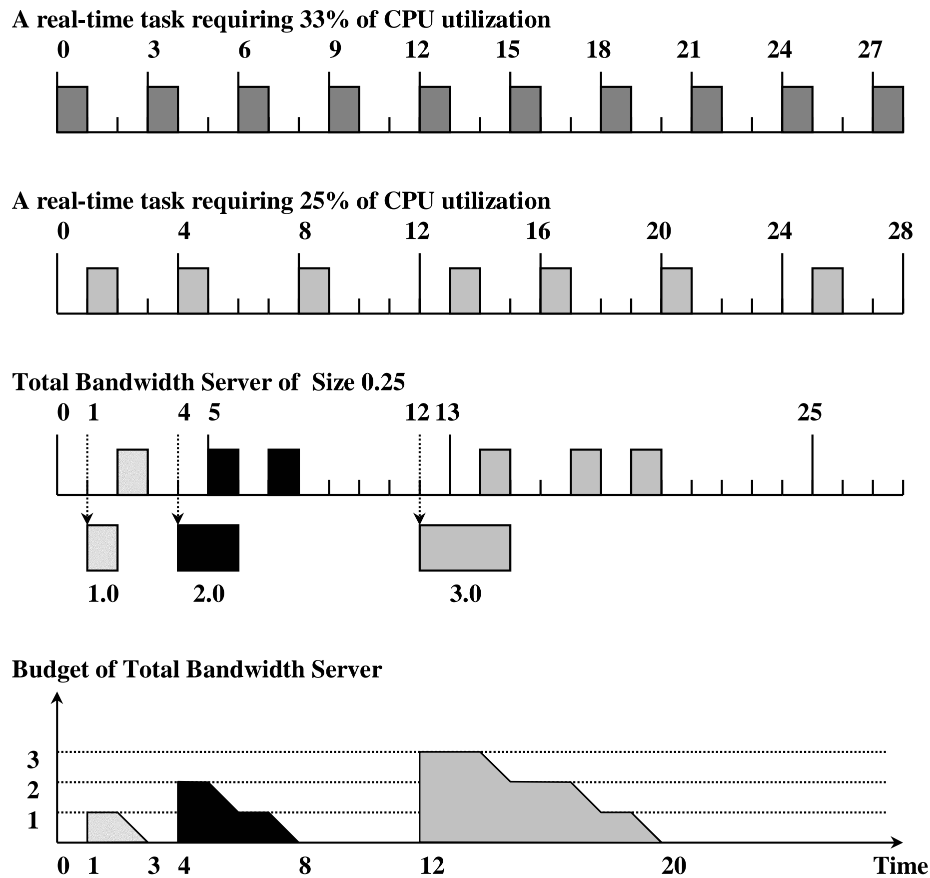

Fig. 12. The Illustration of Total Bandwidth Server Operations

Figure 12 shows how a Total Bandwidth Server works on an EDF-based schedul-

ing:

- The sum of the total utilization of periodic tasks and the utilization of the Total Bandwidth Server is less than 1.
- At time 1, a sporadic job with an execution time of 1 time unit arrives.
- The deadline of the Total Bandwidth Server is set to time 5 (= 1 + 1.0/0.25).
- The budget of the Total Bandwidth Server is set to 1 at time 1, but the job of the periodic task of period of 4 time units has the earliest deadline, therefore the periodic job has the priority over the Total Bandwidth Server.
- The first sporadic job completes at time 3 before its deadline.
- The second sporadic job arrives at time 4 with execution time of 2 time units.
- The budget of the Total Bandwidth Server is set to 2, and the Total Bandwidth Server's deadline is set to time 13.
- The second sporadic job obtains CPU control at time 5, but the control is preempted at time 6 by the EDF scheduler for the job of the second periodic task.
- At time 7, the second sporadic job again obtains CPU control.
- The second sporadic job finishes without a missed deadline.

2. The EDF Scheduler for Total Bandwidth Servers

Bandwidth-preserving schedulers (such as the Total Bandwidth Server) are typically implemented on top of an EDF scheduling mechanism. Therefore, we designed and implemented an EDF scheduler class that is compliant with the RTSJ RI.

Figure 13 shows the state diagram of our EDF scheduler. When the EDF scheduler has the control of a CPU, the EDF scheduler first checks whether there is any newly admitted instance of *javax.realtime.RealtimeThread* class or not. If it is, the

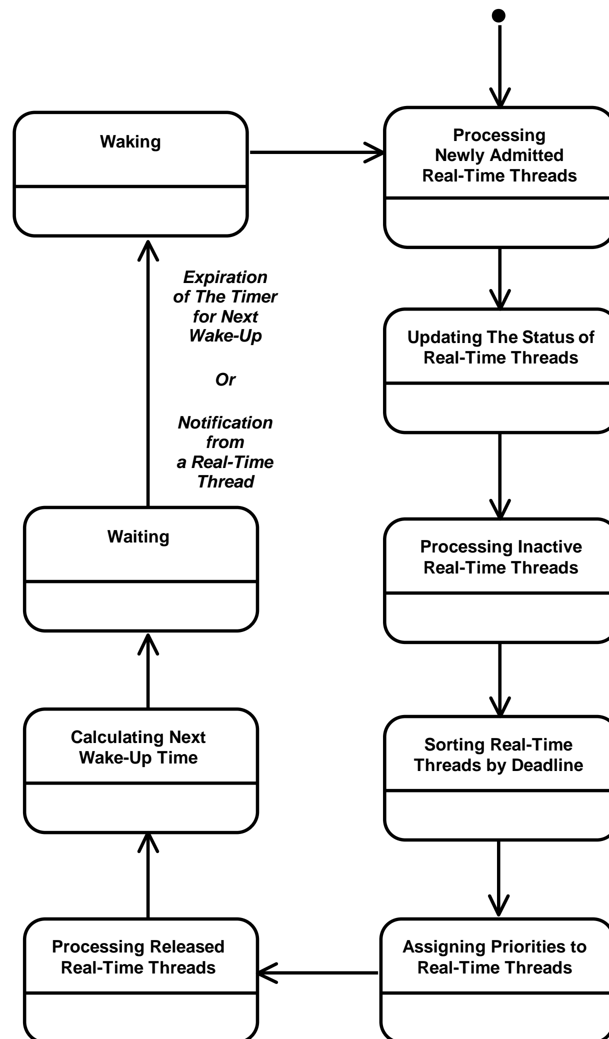

Fig. 13. The State Diagram of the EDF Scheduler for Agile Objects System

EDF scheduler creates an instance of class *ao.realtor.scheduler.EDFScheduler$SchedulableData* apiece for managing instances of class *javax.realtime.RealtimeThread*. After that, the EDF scheduler evaluates the status of all instances of class *javax.realtime.RealtimeThread*. There are several requests from instances of class *javax.realtime.RealtimeThread* for putting their operating system threads into desired operating systems' states, such as start, stop, resume, sleep and suspend. The EDF scheduler

also examines whether or not any instance of class *javax.realtime.RealtimeThread* missed its deadline and whether or not the operating system thread of an ongoing instance of class *javax.realtime.RealtimeThread* is alive. As mentioned before, the EDF scheduler has the highest priority over instances of class *javax.realtime.RealtimeThread* and instances of class *java.lang.Thread*.

3. A Probabilistic Approach for Characterizing Total Bandwidth Servers

Each Total Bandwidth Server is characterized two parameters: the maximum budget and the replenishment period. While the maximum budget can be established by an execution-time analysis of the remote methods in the agile objects, it is difficult to choose an optimal replenishment period of the Total Bandwidth Server.

If we assume that inter-arrival times of client requests for each remote method are distributed based on a given distribution function, we have two options for deciding the replenishment period: One option is to use the minimum inter-arrival time of invocations as the replenishment period. This approach is not applicable to open systems, where little is known about the client population. Setting the invocation period short enough to handle the bursty arrivals caused by bursty client invocations and by phasing of invocations from multiple clients would lead to unacceptably low utilization of host resources. Alternatively, one can take probabilistic approach. In this approach, each Total Bandwidth Server is modeled as a G/D/1 queue [31, 32]. Client requests arrive in the queue with a randomly distributed arrival time, and the Total Bandwidth Server allows for execution of the requested remote method for the given maximum budget time units in each period of the Total Bandwidth Server.

Following Abeni *et al.* [31, 32] describe the sequence of invocations as a random process $v_i = \beta_i - \alpha_i - \Phi$, where β_i and α_i denote the absolute deadline and the arrival time of the ith request, respectively, and Φ denotes the period of a Total

Bandwidth Server. By the replenishment rules of the Total Bandwidth Server, the absolute deadline is $\beta_i = max\{\alpha_i, \beta_{i-1}\} + \Phi$. If we define $\theta_i = \beta_i - \alpha_i$ as the relative deadline of the ith request, we have the distribution of θ_i through the distribution of the random process v_i by the definition, $\beta_i - \alpha_i = v_i + \Phi = \theta_i$.

Based on $\beta_{i+1} = max\{\alpha_{i+1}, \beta_i\} + \Phi$, we can have

$$\begin{aligned} v_{i+1} &= \beta_{i+1} - \alpha_{i+1} - \Phi \\ &= max\{\alpha_{i+1}, \beta_i\} + \Phi - \alpha_{i+1} - \Phi \\ &= max\{\alpha_{i+1}, \beta_i\} - \alpha_{i+1} \\ &= max\{0, \beta_i - \alpha_{i+1}\} \\ &= max\{0, \alpha_i + v_i + \Phi - \alpha_{i+1}\} \\ &= max\{0, v_i - (\alpha_{i+1} - \alpha_i) + \Phi\}. \end{aligned}$$

As we define $\delta_{i+1} = \alpha_{i+1} - \alpha_i$, we have $v_{i+1} = max\{0, v_i - \delta_{i+1} + \Phi\}$. We can consider the random process v_i as a Markov process so that we could find the stationary transition probability matrix T, where Π is the state probability matrix of the random process v_i and $\Pi^{(i)} = T \times \Pi^{(i-1)}$. If we define $R_j = P[\delta_i = j]$ and ${\pi_m}^{(i)} = P[v_i = m]$, we get

$$\begin{aligned} {\pi_m}^{(i)} &= P[v_i = m] \\ &= P[max\{0, v_{i-1} - \delta_i + \Phi\} = m] \\ &= \sum_{k=-\infty}^{\infty} P[\langle max\{0, v_{i-1} - \delta_i + \Phi\} = m\rangle \wedge \langle v_{i-1} = k\rangle] \\ &= \sum_{k=-\infty}^{\infty} P[max\{0, k - \delta_i + \Phi\} = m] P[v_{i-1} = k]. \end{aligned}$$

For the case of $m = 0$:

$$
\begin{aligned}
{\pi_0}^{(i)} &= \sum_{k=-\infty}^{\infty} P[k - \delta_i + \Phi \leq 0] P[v_{i-1} = k] \\
&= \sum_{k=-\infty}^{\infty} P[\delta_i \geq k + \Phi] P[v_{i-1} = k] \\
&= \sum_{k=0}^{\infty} \sum_{j=k+\Phi}^{\infty} P[\delta_i = j] {\pi_k}^{(i-1)} \\
&= \sum_{k=0}^{\infty} \sum_{j=k+\Phi}^{\infty} R_j {\pi_k}^{(i-1)}.
\end{aligned}
$$

For the case of $\forall m > 0$:

$$
\begin{aligned}
{\pi_m}^{(i)} &= \sum_{k=-\infty}^{\infty} P[k - \delta_i + \Phi = m] P[v_{i-1} = k] \\
&= \sum_{k=-\infty}^{\infty} P[\delta_i = k - m + \Phi] {\pi_k}^{(i-1)} \\
&= \sum_{k=0}^{\infty} R_{k-m+\Phi} {\pi_k}^{(i-1)}.
\end{aligned}
$$

The transition probability matrix T looks like the following:

$$
T = \begin{pmatrix}
\sum_{\ell=\Phi}^{\infty} R_\ell & \sum_{\ell=\Phi+1}^{\infty} R_\ell & \sum_{\ell=\Phi+2}^{\infty} R_\ell & \sum_{\ell=\Phi+3}^{\infty} R_\ell & \sum_{\ell=\Phi+4}^{\infty} R_\ell & \cdot \\
R_{\Phi-1} & R_\Phi & R_{\Phi+1} & R_{\Phi+2} & R_{\Phi+3} & \cdot \\
R_{\Phi-2} & R_{\Phi-1} & R_\Phi & R_{\Phi+1} & R_{\Phi+2} & \cdot \\
R_{\Phi-3} & R_{\Phi-2} & R_{\Phi-1} & R_\Phi & R_{\Phi+1} & \cdot \\
R_{\Phi-4} & R_{\Phi-3} & R_{\Phi-2} & R_{\Phi-1} & R_\Phi & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
R_0 & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & R_0 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}.
$$

Again, we can get the state probability matrix Π, where

$$\begin{pmatrix} {\pi_0}^{(i)} \\ {\pi_1}^{(i)} \\ {\pi_2}^{(i)} \\ \cdot \end{pmatrix} = \begin{pmatrix} \sum_{\ell=\Phi}^{\infty} R_\ell & \cdot & \cdot & \cdot \\ R_{\Phi-1} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \times \begin{pmatrix} {\pi_0}^{(i-1)} \\ {\pi_1}^{(i-1)} \\ {\pi_2}^{(i-1)} \\ \cdot \end{pmatrix}.$$

If we assume that inter-arrival times of client requests are exponentially distributed, each Total Bandwidth Server can be modeled as a M/D/1 queue. The probability density function for inter-arrival time, δ, is given by

$$f_{pdf}(\delta) = \begin{cases} \lambda e^{-\lambda\delta} & \text{if } \delta \geq 0\,; \\ 0 & \text{if } \delta < 0\,, \end{cases}$$

where λ denotes an inter-arrival rate of client requests. We have the following transition probability matrix T:

$$T = \begin{pmatrix} (1+\lambda)e^{-\lambda\Phi} & (1+\lambda)e^{-\lambda(\Phi+1)} & (1+\lambda)e^{-\lambda(\Phi+2)} & (1+\lambda)e^{-\lambda(\Phi+3)} & \cdot \\ \lambda e^{-\lambda(\Phi-1)} & \lambda e^{-\lambda\Phi} & \lambda e^{-\lambda(\Phi+1)} & \lambda e^{-\lambda(\Phi+2)} & \cdot \\ \lambda e^{-\lambda(\Phi-2)} & \lambda e^{-\lambda(\Phi-1)} & \lambda e^{-\lambda\Phi} & \lambda e^{-\lambda(\Phi+1)} & \cdot \\ \lambda e^{-\lambda(\Phi-3)} & \lambda e^{-\lambda(\Phi-2)} & \lambda e^{-\lambda(\Phi-1)} & \lambda e^{-\lambda\Phi} & \cdot \\ \lambda e^{-\lambda(\Phi-4)} & \lambda e^{-\lambda(\Phi-3)} & \lambda e^{-\lambda(\Phi-2)} & \lambda e^{-\lambda(\Phi-1)} & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \lambda & \cdot & \cdot & \cdot & \cdot \\ 0 & \lambda & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

In this probabilistic approach, we can get an optimal period of a Total Bandwidth Server, Φ, which maximizes the value of ${\pi_0}^{(i)}$ with a given distribution function of inter-arrival times of client requests and the worst-case execution time of each remote

method of agile objects.

Finally, the utilization of each remote method can be defined by dividing the maximum budget by the replenishment period Φ of the Total Bandwidth Server for each remote method. Given these parameters, the Total Bandwidth Server is fully defined.

G. Experimental Evaluation

In this section, we evaluate the real-time capabilities of the extensions to Java RMI described in the previous sections.

First, the average and standard deviation of execution times of a local method are measured on five different Java Virtual Machines (VMs). This experiment illustrates the level at which each Java VM guarantees predictable execution times for local methods. We also use the same local method as a target method for upcalls requested by RMI clients throughout the experiments of this section. In this way, we can later determine the net average overhead of remote method invocations in addition to the execution time of the local method.

In a second step, we measure latency of the remote method invocations. This experiment evaluates whether or not our methodology provides predictable latency for a real-time RMI server in the presence of heavily CPU-bound tasks.

Finally, we evaluate the performance of the EDF job scheduler and the Total Bandwidth Server that ensure predictable execution times for both periodic and sporadic real-time tasks.

1. Local Method Execution Time

For comparison of local execution time, we have used TimeSys 3.1 Real-Time version for OS and five Java VMs: JDK[1] 1.3.0-classic VM [33], JDK 1.3.0-interpreted mode [33], JDK 1.3.0-mixed mode [33], TimeSys Real-Time Specification for Java Reference Implementation (RTSJ RI) [25], and TAMU RTSJ RI with Real-Time Remote Method Invocation (RT-RMI), that is, our implementation.

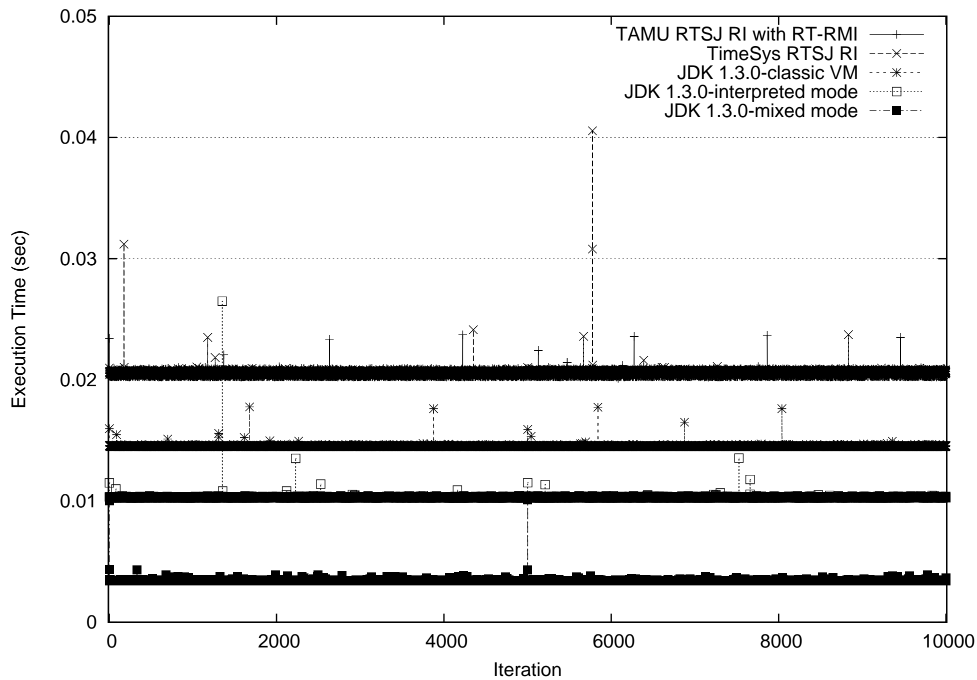

Fig. 14. Execution Time of Local Method on TimeSys 3.1-RT

Figure 14 shows the execution time distribution of a local method for each Java VM. This was measured on the server (Dell Dimension 4100 Pentium III 933 MHz with memory of 256 Megabytes). As can be seen on Figure 15, all JDK versions take

[1]Java Development Kit; the standard Java development tools provided by Sun Microsystems.

less time to execute the local method than the TimeSys RTSJ RI.

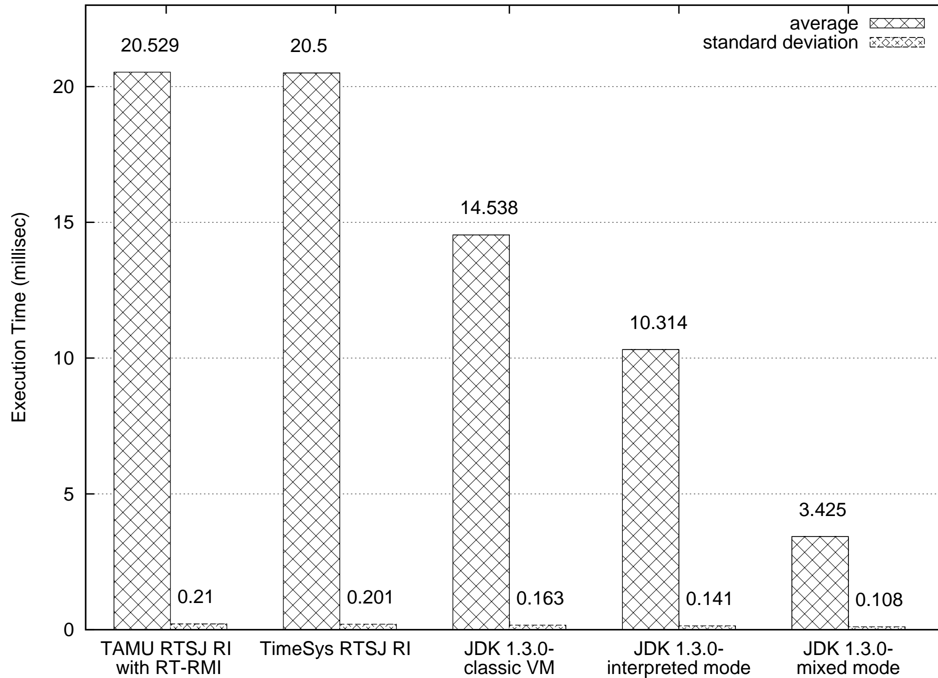

Fig. 15. Average and Standard Deviation of Local Method Execution Time on TimeSys 3.1-RT

TimeSys RTSJ RI takes approximately six times longer than JDK 1.3.0-mixed mode. This is because the TimeSys version is (a) targeted towards real-time execution, thus does not contain many optimizations that optimize performance, and (b) is a very preliminary implementation at that. Since we use the TimeSys RTSJ RI as base to implement TAMU RTSJ RI with RT-RMI, so our implementation inherits the overhead of the TimeSys version. However, as shown in Figure 15, the overhead of adding RT-RMI to our implementation is negligible.

2. Latency of Remote Method Invocation

In this experiment, the real-time Java RMI performance is measured in terms of averages of the latencies of periodic remote method invocations. In order to focus on real-time performance we use the simple configuration depicted in Figure 16, where a network analyzer is directly connected between two hosts (Dell Dimension 4100 Pentium III 933 MHz with memory of 256 Megabytes). We use a Fast Ethernet that supports 100 Mbps. We also use an Agilent Technologies Network Analyzer that has nanosecond timer resolution and Windows 2000 Pro Embedded with two CPUs.

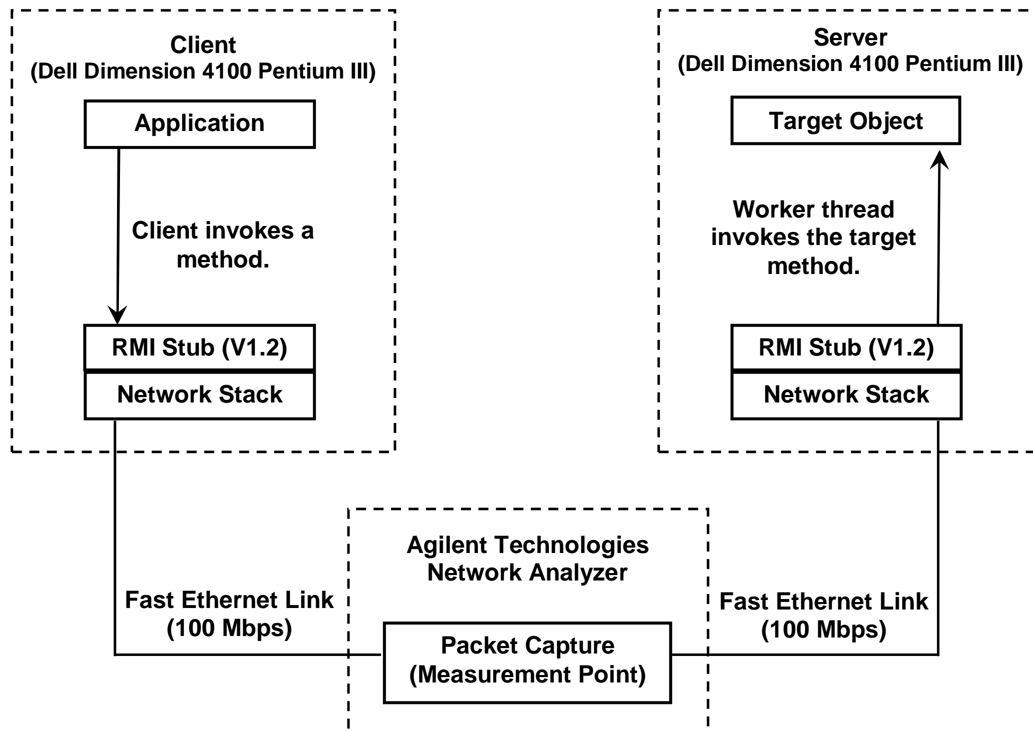

Fig. 16. The Experiment Environment

We show the latency of remote method invocation by measuring the time difference between the moment of client's sending of the first packet of the RMI request

and the moment of server's sending of the last packet of the result. The Agilent Technologies network analyzer captures all packets on the link between the server and the client. We use Ethereal [34] in order to extract timing information from the captured packets by using a refined data display for the RMI protocol. The use of the network analyzer allows our measurements not to perturb the execution of the RMI server. For periodic job arrivals multi-threaded client application generates remote method invocations to the server.

a. Java VM Running One RMI Server and High Background Load

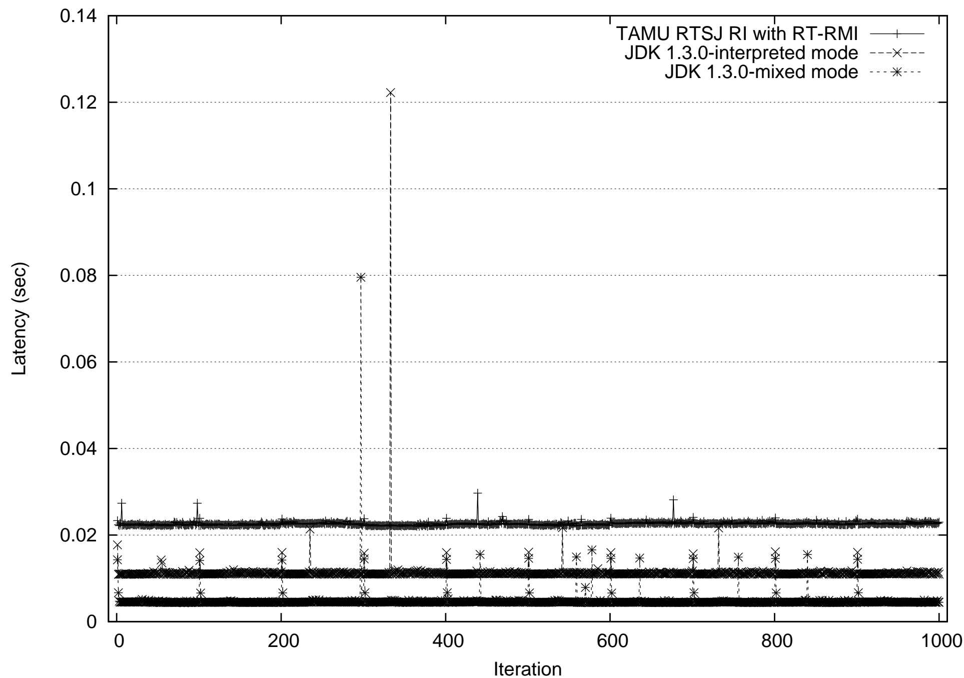

Fig. 17. RMI Latency with High Background Load

We run two non-real-time Java applications that compress big size of files on the RMI server's Java VM to generate a high background load. Figure 17 shows

the latency of remote method invocations on three Java VMs: JDK 1.3.0-interpreted mode, JDK 1.3.0-mixed mode, and TAMU RTSJ RI with Real-Time Remote Method Invocation (RT-RMI). As can be seen, lower two lines corresponding JDK 1.3.0-interpreted and mixed modes respectively, show unusual high latencies. It apparently shows sporadic long latencies of lower two lines corresponding JDK 1.3.0-interpreted and mixed modes respectively while upper one line corresponding TAMU RTSJ RI with RT-RMI shows very predictable latency.

Figure 18 shows the average latency of the same experiment with Figure 17. TAMU RTSJ RI with RT-RMI has larger average latency than those of two JDK 1.3.0 modes. However, as can be recalled from Figure 15, the average latency of TAMU RTSJ RI with RT-RMI for executing remote methods is inherited from TimeSys RTSJ RI. In addition, the overhead of the RMI protocol handling that includes the particular overhead for agile objects is 9% of the average latency of TAMU RTSJ RI with RT-RMI for executing remote methods while the overhead of RMI protocol handling in JDK 1.3.0-interpreted mode is 8% of the average latency of remote method invocations.

Figure 19 clearly shows how most of the latency variation is due to RMI protocol handling. This data also clearly demonstrates how our RT-RMI implementation (*i.e.*, addition of real-time worker thread management and Total Bandwidth Server) greatly increases RMI predictability.

This result demonstrates that TAMU version clearly supports real-time capability while the other two versions do not. In other words, the latencies of remote method invocations are predictable in TAMU version due to the EDF job scheduler and the Total Bandwidth Server.

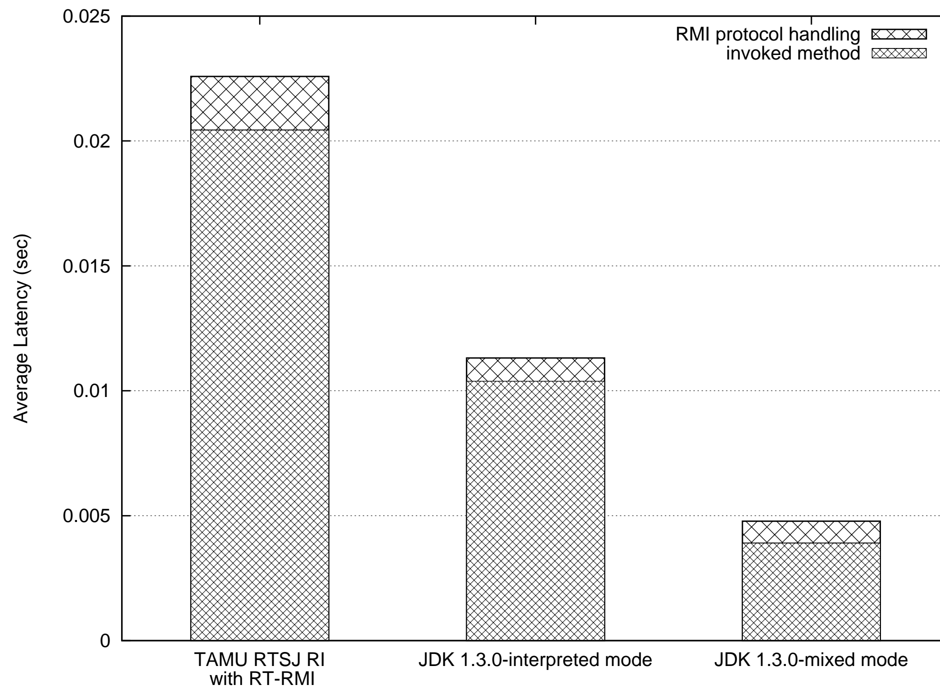

Fig. 18. Decomposition of RMI Latency

b. Java VM Running One RMI Server and Varying Amount of Background Load

Figure 20 shows the comparison of average and standard deviation of the RMI server's latencies in server execution environments where other workloads run together. The RMI server consumes 22% of CPU utilization. In Figure 20 each label in horizontal axis stands for the following.

- "None": there is no other workload in the server VM except the RMI server.
- "BG-25": one background Java thread is running in the RMI server's VM. It consumes 25% of CPU utilization. It has a Java priority of 5 and is not under control of our EDF scheduler.
- "Two RT-10s": two real-time Java threads are running in the RMI server's VM. Each real-time Java thread performs CPU-bound computations periodically and consumes 10% of CPU utilization.

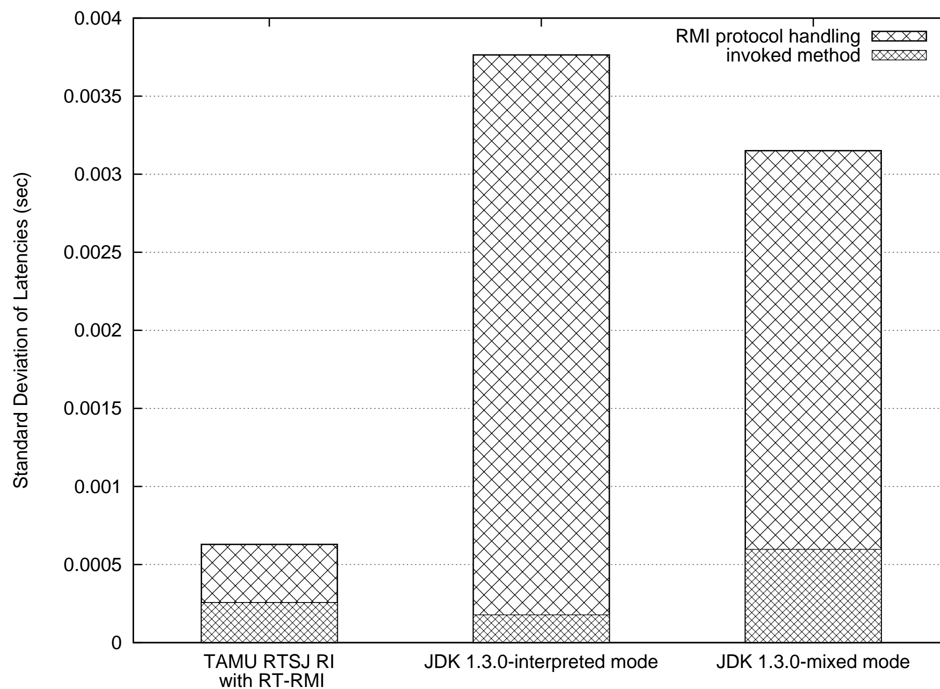

Fig. 19. Standard Deviation of RMI Latency

- "RT-10": one real-time Java thread is running in the RMI server's VM. It consumes 10% of CPU utilization.
- "RT-20": one real-time Java thread is running in the RMI server's VM. It consumes 20% of CPU utilization.
- "RT-30": one real-time Java thread is running in the RMI server's VM. It consumes 30% of CPU utilization.
- "RT-40": one real-time Java thread is running in the RMI server's VM. It consumes 40% of CPU utilization.
- "RT-50": one real-time Java thread is running in the RMI server's VM. It consumes 50% of CPU utilization.
- "RT-60": one real-time Java thread is running in the RMI server's VM. It consumes 60% of CPU utilization.

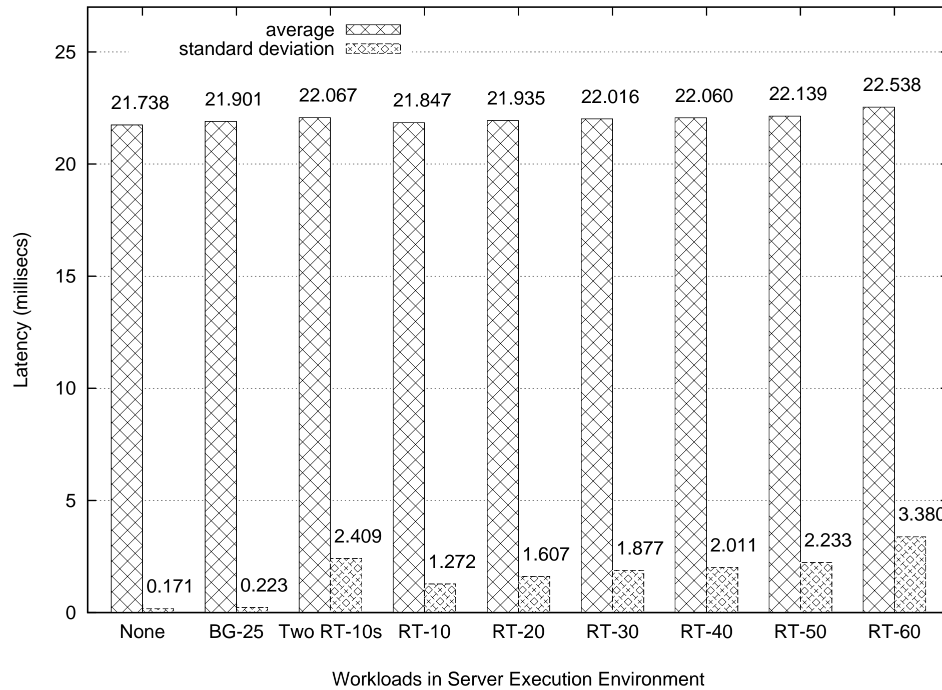

Fig. 20. RMI Latency with Varying Amount of Background Load

As can be seen, the average of the RMI server's latencies very slightly increases as the workload of a real-time Java thread increases. There is an increase of 3.68% in average latency of the RMI server when a real-time Java thread consumes 60% of CPU utilization in the RMI server's VM. However, the standard deviation of the RMI server's latencies increases up to 3.380 milliseconds.

In addition, Figure 21 shows the decomposition of the RMI server's latencies of the same experiment with Figure 20. As can be seen on Figure 21, the averages of the RMI latencies both for invoked method and RMI protocol handling vary little when the workloads of other tasks that run on the RMI server's VM change.

This result demonstrates how well TAMU RTSJ RI with RT-RMI provides predictable latency of remote method invocations even in conditions with varying amount of background load.

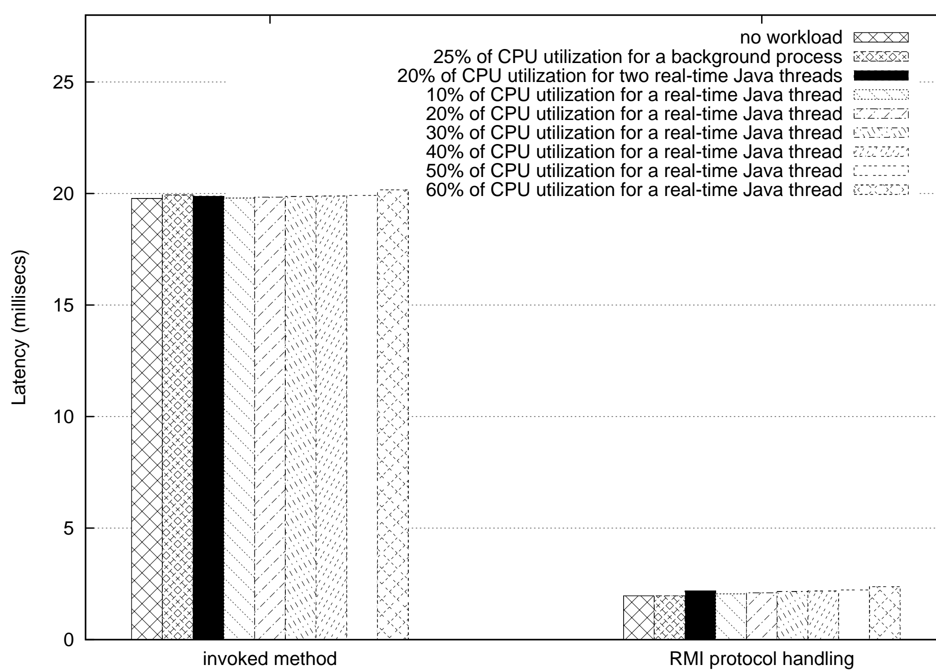

Fig. 21. Decomposition of RMI Latency with Varying Amount of Background Load

CHAPTER V

MOBILITY

A. Introduction

Dynamic process migration has been attractive for dynamic load balancing [35], fault tolerance [36], and high-throughput computing [37] in distributed systems.

By migrating processes from highly loaded nodes to lightly loaded nodes at runtime, dynamic load balancing can be achieved; in this way on-the-fly workload balancing mechanisms maximize system utilization.

When resources are geographically localized, for example in grid computing environments, process migration also leverages efficient utilization of the resources, including network bandwidth, by migrating processes closer to the nodes where the resources are located.

The Condor® Project [37] has developed software middleware for high-throughput computing environments. This middleware is based to a large extent on process migration and checkpointing mechanisms. The checkpointing mechanism captures the complete set of information for a migrating process. The given set of information allows the migrating process to continue its execution on the new location.

Process migration mechanisms transfer the memory image of running processes to the destination's execution environment. The memory image includes information of the states of both the running process and opened files. The states consist of virtual memory state, execution state, kernel state, and message channel state. As a result, the cost for process migration is proportional to the size of the process' memory image and the number of accessed files. As operating systems evolve, more complexity is involved in process migration. Typically, OS supported process migration allows a

process to migrate from one node to another *homogeneous* node. This means thus such schemes work only on homogeneous computing environments.

Alternatively, one can take advantage of language-level mechanisms, such as the Java Object Serialization protocol [21] to migrate language-level entities, such as Java objects. The sender of a Java object encapsulates the object's dynamic information in a byte stream through the Java Object Serialization protocol. On the receiver side, a copy of the original object is created, although static or transient fields are not transmitted. We distinguish passive from active objects, depending on whether they contain threads while a passive object has no thread active over any significant amount of time, active objects have such threads. For example, objects that contain listening threads are active. Examples of active objects are objects of class *java.rmi.server.UnicastRemoteObject* and its derivative classes. Since remote invocation relies on the presence of listening threads, passive objects are naturally referenced only by other objects in the same Java Virtual Machine.

In the Agile Objects System, mobility for agile objects is provided by the Java Object Serialization protocol. Agile objects are active objects due to their inheritances from class *java.rmi.server.UnicastRemoteObject*. In order to keep the migration mechanism simple, objects with executing threads cannot be migrated. Agile objects cannot be migrated in the middle of executing requested methods for clients. When all requested methods have been completed, the agile object can be migrated onto another node. The listening thread is terminated before migration and restarted after successful migration. This is possible because this thread has no state.

Our methodology satisfies the following objectives: application transparency by providing Java package *ao.migration* and package *ao.nameserver*, independence from heterogeneous execution environments by utilizing Java virtual machine, fast migration by applying both Java Object Serialization protocol and a proactive resource

discovery.

B. Related Work

We classify the migration of software component into three categories based on granularity: process, thread, and active object.

Condor [37, 38] implements UNIX process migration by using checkpointing and restart of the process. When an original process is created, Condor code linked with the original process installs a signal handler for the checkpoint signal. This signal handler allows for saving the state of a process' CPU into a checkpoint file. Because the primary concern in Condor is to make sure that the owner of a workstation does not have degraded performance after adding his or her workstation into the Condor pool of workstations, the process of the job must vacate the workstation when the owner begins to use it. The checkpoint handler causes a core dump of the process to produce a checkpoint file by combining the core file and executable modules when the running process receives a checkpoint signal. At restart time Condor restores text, data and stack segments from the checkpoint file. The high cost of Condor's process migration, however, limits the usefulness of Condor for small jobs as well as for real-time applications [39].

Ma [40] recently proposed Java Message Passing Interface (MPI) that supports transparent Java process migration for load balancing. This technique utilizes the Java Virtual Machine Debugger Interface (JVMDI) to capture the execution context and restore it at the Java bytecode level after migration. It is natural that Ma's approach improves throughput because workload allocation for conventional MPI is static whereas the Java MPI moves workload to lightly-loaded hosts transparently. However, having JVMDI on top of the Java VM causes new overhead because all

method invocations are executed under control of the JVMDI.

Zhu [41] has proposed a Distributed Java Virtual Machine (DJVM) that supports transparent Java thread migration by implementing an embedded global object space layer in the Java VM. This architecture requires global load monitor software that monitors workload on each machine and triggers migration for load balancing.

Troger [42] uses Aspect-Oriented Programming (AOP) to achieve transparent object migration in the Microsoft .NET environment. Although Troger supports a run-time entity migration, the work does not address real-time capability, our major concern.

Significant research has been performed to make run-time migration of software component as practical as possible. Although our survey is not exhaustive, we have not found one that has specifically addressed real-time capability for run-time migration of software component.

C. Our Methodology for Migration

Generally speaking, a software component is a self-contained package that holds its code and state. In our methodology, the software component is an *active object* of the Java RMI server that extends class *java.rmi.server.UnicastRemoteObject*. The migration of software component, therefore, means that Java RMI server, active object, continues to run in another host after migration because Java RMI mechanism associated with Java Object Serialization protocol automatically re-builds the Java RMI server's run-time environment in the host where the active object moved in. In this way, the migration is performed in a shortest time by utilizing Java built-in functions; dynamic class loading, the RMI mechanism, and the object serialization protocol. The migration is typically performed as follows:

1. Class Migratable declaration: For components that need to be migrated at run-time, we define their classes by extending a Java built-in class *java.rmi.server.UnicastRemoteObject*. By invoking the *exportObject()* method of class *java.rmi.server.UnicastRemoteObject* in the initialization, we make the instances of the new class available outside. These classes are also declared as interface *java.io.Serializable*.
2. Migration decision: Migration is triggered by the invocation of the method *migrate()* of the migratable RMI server. This method in turn is called by the local load balancer or by some policy module that react to an adverse event such as system malfunctions, local system policy changes, and cyber attacks.
3. State saving: We use the Java Object Serialization protocol to save the run-time state of the RMI server. Serialization is the process of saving an object's dynamic state into a byte stream.
4. Locating a destination host: In order to support low-latency migration, the destination host information needs to be readily available at the time of migration decision.
5. Transfer of Code and State Information: We use the Java's dynamic class loading and the Java Object Serialization protocol to send the code and the state of the RMI server to the destination, respectively.
6. Continuation of Execution: Upon receiving the object, the Java virtual machine in the destination host creates a listening thread for the newly migrated object. By definition, such an object needs a listening thread to receive requests from clients. If the destination host does not have the corresponding class, it first downloads the class from outside and makes the object actively run with a newly created listening thread.

Once an active object is declared by extending class *java.rmi.server.UnicastRemoteObject*, migration is performed through the RMI server mechanism, the object serialization protocol, and the Java's dynamic class loading.

1. Java Object Serialization Protocol

The Java Object Serialization protocol provides the ability to store and retrieve Java objects. The serialized byte streams represent the state of one or more Java objects, that is, sufficient data to reconstruct the Java objects.

As seen in Figure 22, Java serialization converts the memory representation of the target object in the heap into a stream of bytes. This process includes the serialization of all reachable objects from the target object. Usually the serialized stream of bytes is stored in persistent storage or transmitted to another Java virtual machine over network. On the receiver's Java virtual machine, the transmitted stream of bytes can be used to reconstruct all objects and store them into heap memory. Java objects to be stored in the byte streams should implement either interface *java.io.Serializable* or interface *java.io.Externalizable*.

Interface *java.io.Serializable* defines no methods. It serves only to identify classes whose objects may be saved into byte streams. Interface *java.io.Externalizable* defines methods *writeExternal()* and *readExternal()*. For the Externalizable classes, which implement interface *java.io.Externalizable*, the classes are responsible for saving and restoring the contents of their instances. Interface *java.io.Externalizable* also extends interface *java.io.Serializable*.

The Externalizable classes implement method *writeExternal()* to save its contents by calling the *writeObject()* method of interface *java.io.ObjectOutput* for fields of Java object or by calling the methods of interface *java.io.DataOutput*, such as methods *writeBoolean()*, *writeChar()*, *writeDouble()*, *writeFloat()*, *writeInt()*, *writeLong()*,

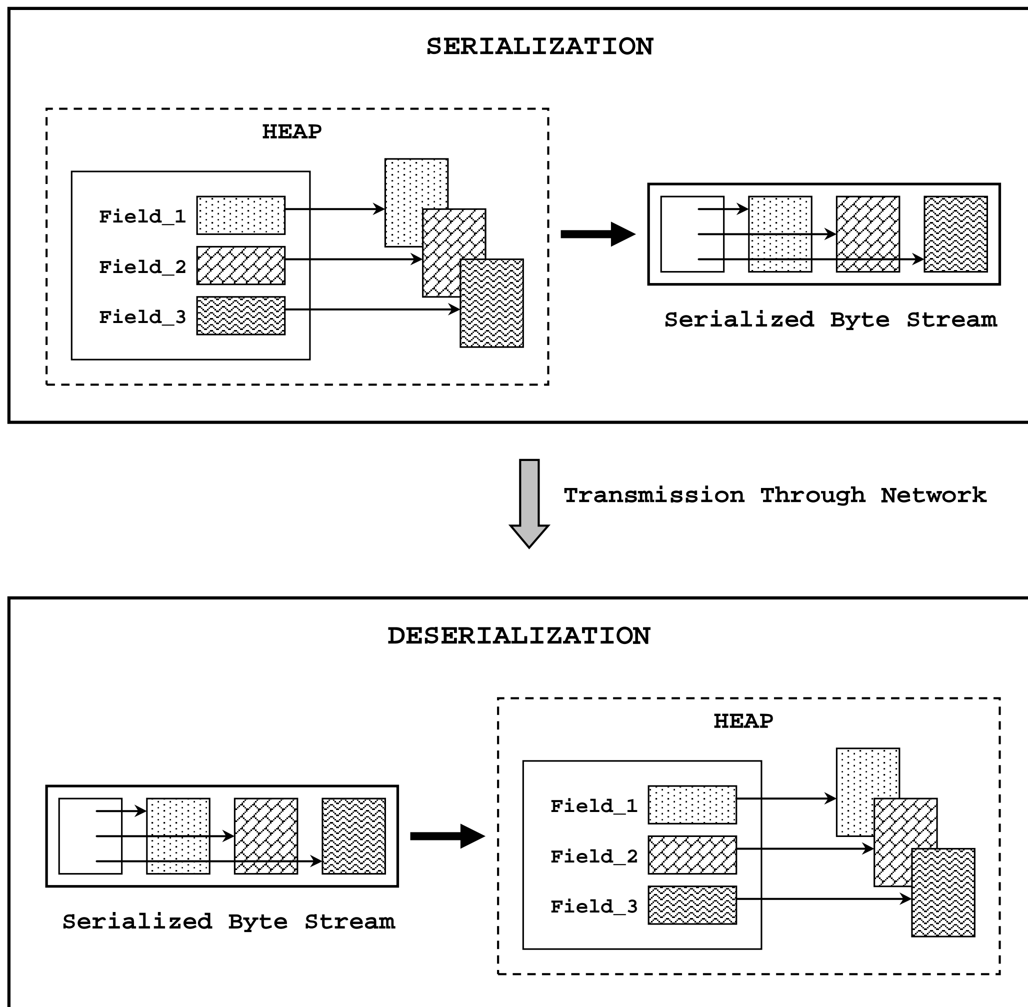

Fig. 22. Java Object Serialization Protocol

writeShort(), or *writeUTF()*, according to their fields of primitive types.

In addition, the Externalizable classes implement method *readExternal()* to restore their contents by calling the *readObject()* method of interface *java.io.ObjectInput* for fields of Java object or by calling the methods of interface *java.io.DataInput*, such as methods *readBoolean()*, *readChar()*, *readDouble()*, *readFloat()*, *readInt()*, *readLong()*, *readShort()*, or *readUTF()*, according to their fields of primitive types. Method *readExternal()* should read the values in the same order and with the same

types as were written by method *writeExternal()*.

For the Serializable classes, which implement interface *java.io.Serializable*, the byte stream includes enough information to restore the fields in the byte stream to a compatible version of the class. The name and signature of the class are included in the serialized byte stream so that the serialized byte stream must be able to identity and verify the Java class of the Java object.

Java class information is required to create a new instance from the serialized byte stream because the contents of the Java object are saved based on the Java class information. Method *writeObject()* is used to write the states of an object for its particular class so that the corresponding method *readObject()* can reconstruct it. The default method for storing instances of class *java.lang.Object* is the *defaultWriteObject()* method of class *java.io.ObjectOutputStream*. Class *java.io.ObjectOutputStream* stores graphs of class *java.lang.Object* and values of primitive data types. It also inherits from abstract class *java.io.OutputStream*.

The Java Object Serialization protocol tests each object that is to be stored to check whether or not the object implements interface *java.io.Externalizable*. If the object supports interface *java.io.Externalizable*, method *writeExternal()* is invoked to save the object. If the object does not implement interface *java.io.Externalizable* and does support interface *java.io.Serializable*, class *java.io.ObjectOutputStream* is used to store the object. Class *java.io.ObjectOutputStream* implements both interfaces *java.io.DataOutput* and *java.io.ObjectOutput*.

During deserialization, a new instance of the class of the serialized object is created, and the *readObject()* method of class *java.io.ObjectInputStream* is invoked. Method *readObject()* reads the object information from the byte stream and restores the fields of the object. Java's safe casting is used to get the desired type. The default *readObject()* method for class *java.lang.Object* is the *defaultReadObject()* method of

class *java.io.ObjectInputStream*. Class *java.io.ObjectInputStream* implements both interfaces *java.io.DataInput* and *java.io.ObjectInput* so that it can deserialize fields of both primitive data types and class *java.lang.Object*. When deserialization of an Externalizable object occurs, a new instance is created by using public no-argument constructor, then method *readExternal()* is invoked.

2. The Deserialization of Java RMI Server Objects

Usually, Java RMI server classes extend class *java.rmi.server.UnicastRemoteObject* because the latter provides its subclasses with built-in support for remote access and invocation. One of those behaviors is to create a listening thread when deserialization occurs. For this, class *java.rmi.server.UnicastRemoteObject* defines its own method *readObject()* for deserialization. As we mentioned before, the subclasses of class *java.rmi.server.UnicastRemoteObject* export themselves to outside clients by providing listening service and by supporting point-to-point active object references using TCP streams. The *exportObject()* method of class *java.rmi.server.UnicastRemoteObject* is responsible for exporting the objects of class *java.rmi.server.UnicastRemoteObject*. When any object of class *java.rmi.server.UnicastRemoteObject* is instantiated, the $\langle$*init*$\rangle$*()* method of class *java.rmi.server.UnicastRemoteObject* invokes method *exportObject()*.

Generic or default Java deserialization and clone mechanisms bypass the execution of $\langle$*init*$\rangle$*()* method of each class, instance variable initializer, for deserialized or cloned objects. The reason for this is that initialized instance values that have been resulted from the execution of each $\langle$*init*$\rangle$*()* method are soon to be replaced with the values from serialized byte streams. Therefore, the customized *readObject()* method of class *java.rmi.server.UnicastRemoteObject* invokes method *reexport()* to export deserialized object of class *java.rmi.server.UnicastRemoteObject*. Method *reexport()*

simply calls method *exportObject()* after execution of the *defaultReadObject()* method of class *java.io.ObjectInputStream*.

3. Java Classes for Agile Objects Migration Mechanism

Figure 23 shows the UML diagram of Java classes for the Agile Objects package *ao*. As seen in Figure 23, interface *ao.migration.Migratable* inherits from interface *ao.nameserver.AORemote*. Interface *ao.nameserver.AORemote* defines a method for identifying unique Agile Object IDentifiers (AOIDs). The AOID should be unique over distributed execution environments of agile objects System. Clients use this AOID to locate a migratable agile object by lookup from the Agile Object Naming server. To be a kind of the RMI server, interface *ao.nameserver.AORemote* extends interface *java.rmi.Remote*. Class *ao.nameserver.AOUnicastRemoteObject* implements interface *ao.nameserver.AORemote* and inherits from class *java.rmi.server.UnicastRemoteObject*.

a. Interface ao.migration.Migratable

Interface *ao.migration.Migratable* defines methods *migrate()*, *startMigrationTo()*, and *completeMigration()*. Agile objects should implement this interface in order to be migratable.

Method *migrate()* takes a destination as a parameter. Method *migrate()* acquires a lock from the Agile Objects Naming Server before starting the serialization of a migrating agile object. Once it has acquired the lock, it calls method *startMigrationTo()*. The reason for acquiring the lock is to make sure that the Naming Server does not give out obsolete references to migrating agile objects. The lock is released after successful migration (see below).

Method *startMigrationTo()* takes two parameters, such as destination and TCP

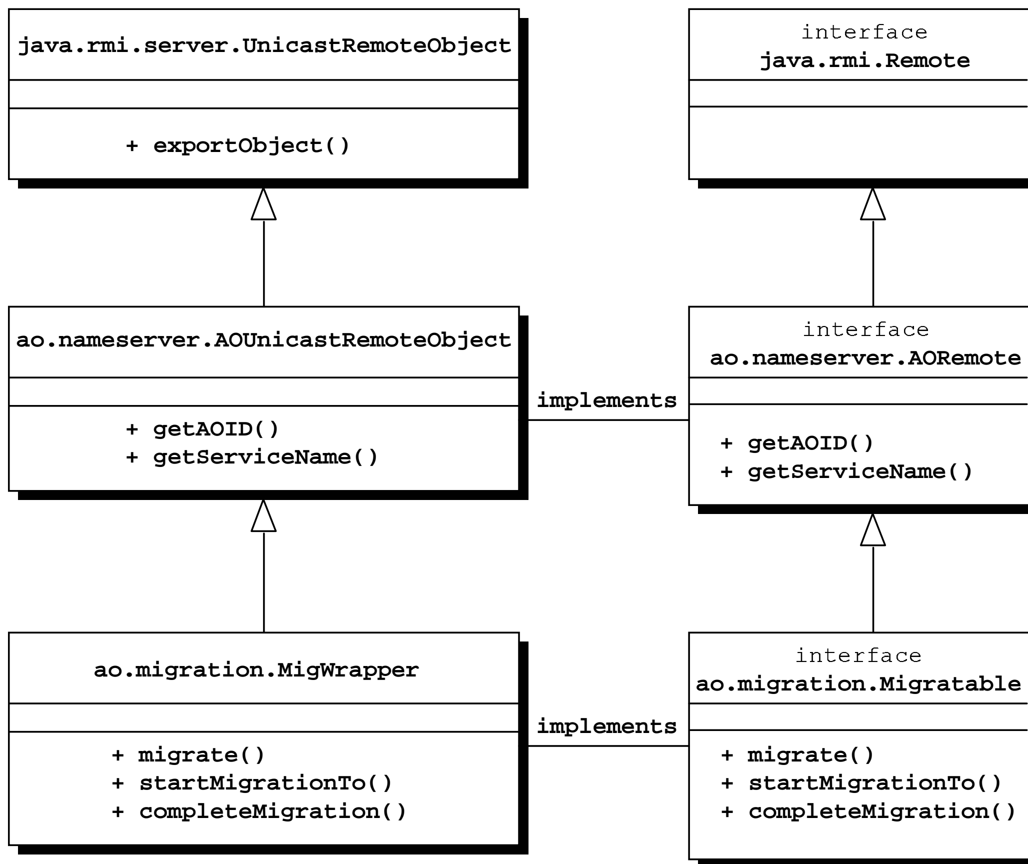

Fig. 23. UML Diagram of Java Classes for Package *ao*

port. The method makes the lookup of the Agile Objects Naming Server and requests the Agile Objects Naming Server to unbind the migrating agile object. Method *startMigrationTo()* also starts transmission of the states of the agile object to destination through Java Object Serialization protocol and class *ao.migration.DataSender*.

Method *completeMigration()* rebinds the migrating agile object associated with new location to the Agile Objects Naming Server after finishing deserialization and the execution of method *reexport()*. When the rebinding to the Agile Objects Naming Server completes, method *completeMigration()* releases the lock acquired from the execution of method *migrate()*.

b. Class ao.migration.DataSender

Migration mechanism uses this class both to serialize an agile object and to ship it to a selected destination.

c. Class ao.migration.MigWrapper

Class *ao.migration.MigWrapper* extends class *ao.nameserver.AOUnicastRemoteObject* and implements interface *ao.migration.Migratable*. Every agile object should extend this class to support migration.

Class *ao.nameserver.AOUnicastRemoteObject* extends class *java.rmi.server.UnicastRemoteObject* and implements interface *ao.nameserver.AORemote*. Interface *ao.nameserver.AORemote* extends interface *java.rmi.Remote*.

d. Class ao.migration.RTJVM

The intension of this class is to reuse the Java virtual machines hosting agile objects after they migrate. Class *ao.migration.RTJVM* is responsible for deserializing migrating agile objects on current node. After deserialization, class *ao.migration.RTJVM* rebinds the agile objects to the Agile Objects Naming Server and releases the lock acquired from the Agile Objects Naming Server by calling the agile objects' method *completeMigration()*.

D. Methodology

In order to create a worst-case scenario (*i.e.*, generating maximum latency for lookup), a strict synchronization is used for acquiring a lock from the Agile Objects Naming server. Whenever an agile object migrates, it should acquire the lock from the Agile Objects Naming server before starting its migration. Until the agile object migrates

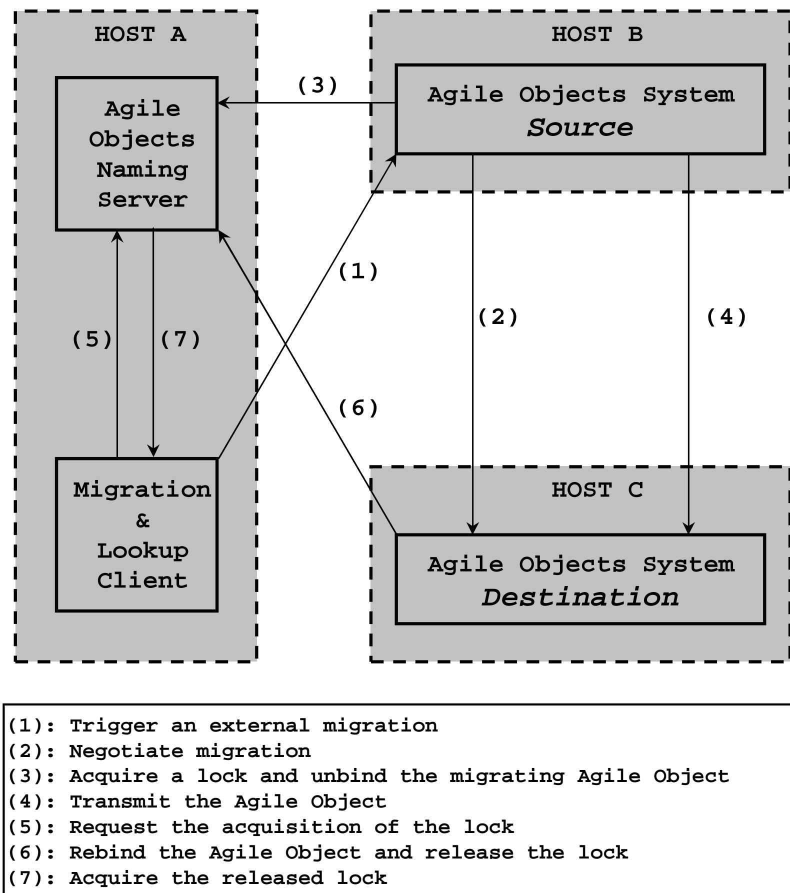

Fig. 24. Experiment Environment for the Worst-Case Latency of Lookup with a Migration

to a new location and releases the lock, no client can acquire the lock from the Agile Objects Naming server to look up the agile object. Figure 24 illustrates this worst-case lookup scenario: an external migration is triggered (1); the agile object's source host negotiates the migration with the destination host (2); before starting the migration, the agile object acquires a lock from the Agile Objects Naming server (3); Once the negotiation succeeds, Java Object Serialization mechanism is used to transmit the agile object to the destination (4); a client tries to acquire the lock from the Agile Objects Naming server, but the client should wait for acquiring the lock until the

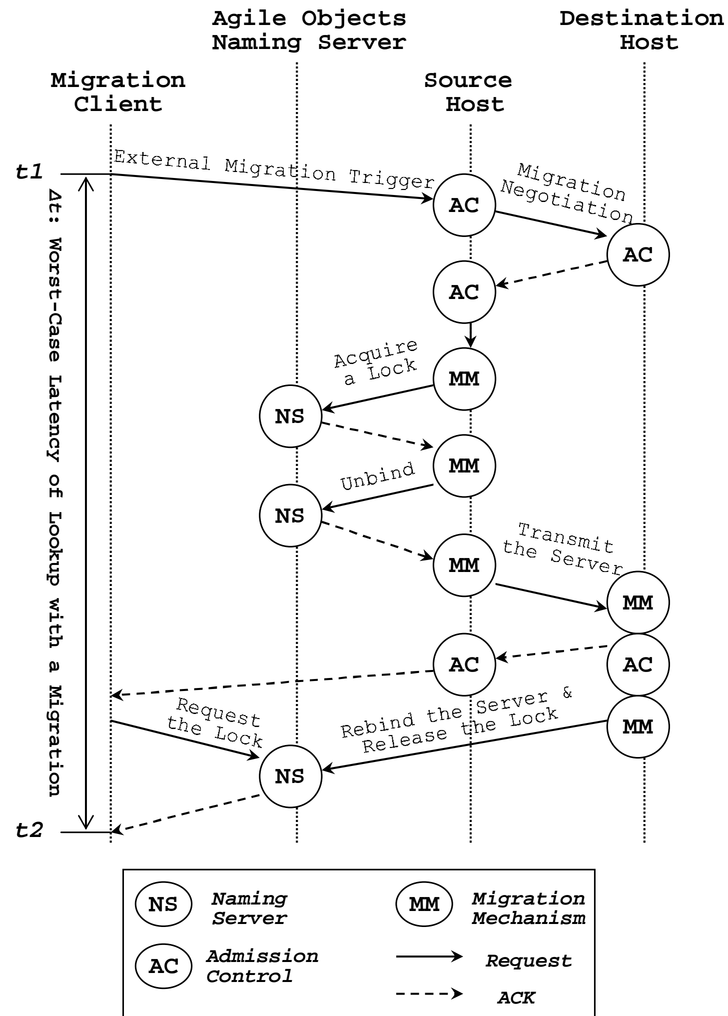

Fig. 25. The Worst-Case Latency of Lookup with a Migration

migration mechanism releases the lock (5); after finishing the deserialization of the migrating agile object, the agile object rebinds itself with the Agile Objects Naming server and releases the lock (6); the synchronization mechanism unblocks the client so that the client acquires the released lock (7).

Figure 25 illustrates the interactions between the components of Agile Objects system for the worst-case lookup scenario of Figure 24 in more detail: When the local *Admission Control* of the source host receives an external migration request,

the *Admission Control* negotiates the migration with the remote *Admission Control* of the destination host. The remote *Admission Control* checks its affordable utilization to decide whether or not to accept the migrating agile object. If the remote *Admission Control* decides to do so, it sends an acknowledgement message to the local *Admission Control*. Then the local *Admission Control* asks the local *Migration Mechanism* to acquire a lock from the Agile Objects Naming server. Once the local *Migration Mechanism* acquires the lock from the Agile Objects Naming server, it unbinds the migrating agile object from the Agile Objects Naming server. After successful unbinding, the local *Migration Mechanism* transmits the agile object to the remote *Migration Mechanism* of the destination host. If the remote *Migration Mechanism* deserializes the agile object, the remote *Migration Mechanism* sends an acknowledgement message to the local *Admission Control* for notifying successful migration. When the local *Admission Control* receives the acknowledgement message from the destination host, it sends an acknowledgement message to the migration client for responding to the external migration request. The client tries to acquire the lock from the Agile Objects Naming server, but it should be blocked until the lock is released by the remote *Migration Mechanism*. Meanwhile the remote *Admission Control* rebinds the newly accepted agile object to the Agile Objects Naming server, then the remote *Admission Control* releases the lock. After the lock is released by the remote *Admission Control*, the client can be unblocked and acquire the lock.

E. Experimental Evaluation

The latencies of lookups for migrating real-time RMI servers (agile objects) should be predictable in order to guarantee the worst-case end-to-end delays for real-time applications. Whenever migratable RMI servers move onto different hosts, clients should

get new remote object references from the Agile Objects Naming server (lookup). The latency of lookup includes the migration delay of the migratable RMI server in the worst case because clients should be blocked until the migratable RMI server rebinds itself to the Agile Objects Naming server. Therefore, we should take the worst-case latency of lookup into account for calculating the worst-case latency of a real-time remote method invocation for migratable real-time RMI server.

To evaluate the performance of the real-time migration mechanism, three hosts (Dell Dimension 4100 Pentium III 933 MHz with 256 Megabytes of memory) are used as depicted in Figure 24. All three hosts are interconnected by a router for 10/100 Mbps transmission rate. One host (Host A) is used for running both the Agile Objects Naming server and the client application, and the other hosts (Host B and Host C) are used as source and destination hosts for migration. Initially, an agile object is running on Host B. If migration is triggered by an external request, the agile object migrates to another Host C. Meanwhile, the client on Host A requests the lookup of the migrating agile object to the Agile Objects Naming server. However, the client should wait for getting the new remote object reference of the migrating agile object until the migration from Host B to Host C completes. After getting the new remote object reference of the agile object, the client invokes a remote method of the agile object. Once the remote method invocation is finished, another external migration is triggered by Host A. The second migration follows the same procedure with the first migration except that the migrating agile object migrates from Host C to Host B. We have performed this experiment 600 times with migrations between Host B and Host C.

We have measured the time difference between *t1* (issue of migration trigger) to *t2* (release of lock at Naming Server) from Figure 25 for the worst-case latency for the lookup of an object that is migrating. Our purpose is to make sure that the client

requests the lock to the Agile Objects Naming server immediately after the *Migration Mechanism* acquires the lock. This ensures that the client waits until the migration completes. In this case we have the worst-case latency for getting the new remote object reference of the migrating agile object from the Agile Objects Naming server.

F. Experimental Results

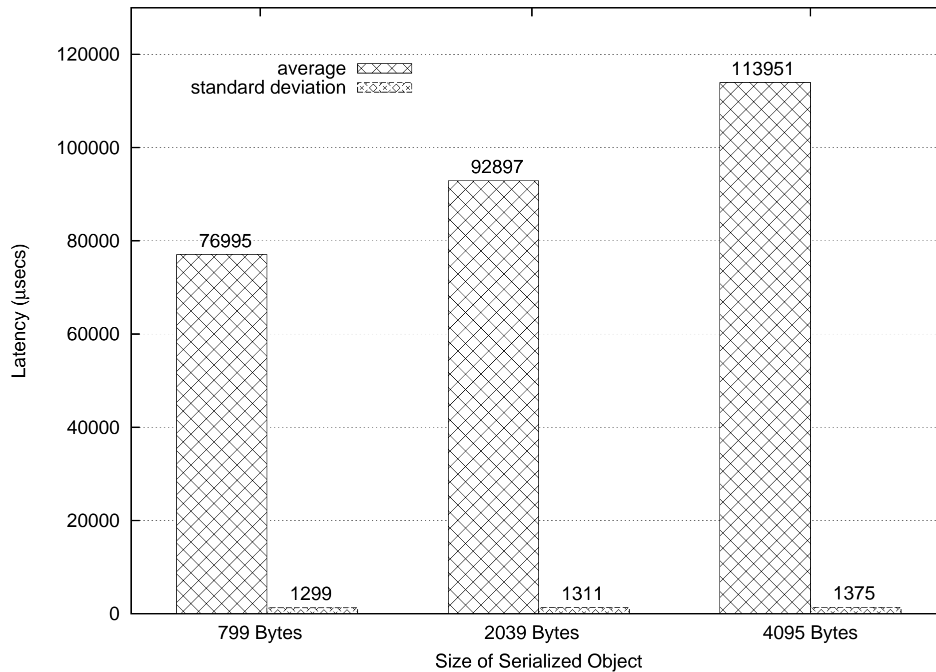

Fig. 26. Latency for a Lookup with a Migration

Figure 26 shows the average and standard deviation of the latencies of the worst-case lookup for the migrating agile object. It shows very predictable latency for the lookup, very small standard deviation of the latencies. The value of the standard deviation for the serialized object size of 799 bytes is 1.7% of the value of the average latency. The percentage of standard deviation to average latency shows that it is in

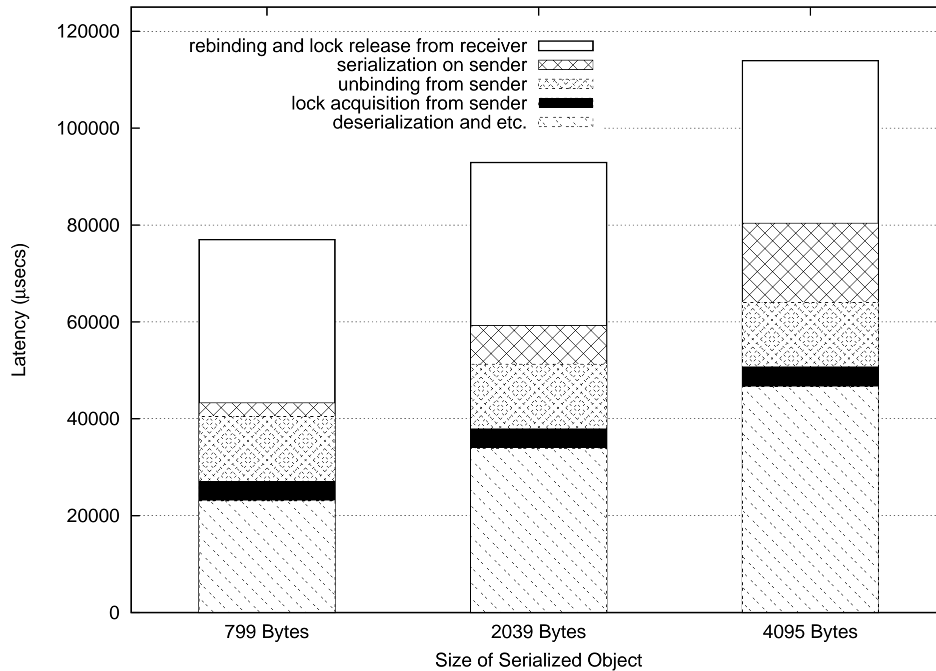

Fig. 27. Decomposition of Latency for a Lookup with a Migration

inverse proportional to the size of serialized agile object since the value of standard deviation does not change as the size of serialized agile object increases. As a result, the value of the standard deviation for the serialized object size of 4095 bytes is only 1.2% of the value of the average latency. Only one migration is allowed during the period of the Total Bandwidth Server. As can be seen from Figure 26, we have predictable latencies for the worst-case lookup with a migration for different sizes of agile objects. As the size of serialized agile object increases, the average of the latencies for the lookup increases. Each standard deviation of the latencies, however, does not increase. The latency for handling the Java Object Serialization protocol depends on the size of serialized Object. As can be seen in Figure 27 and Figure 28, only the latencies for serialization and deserialization increase as the size of serialized agile object increases. The other latencies, such as lock acquisition latency, unbinding

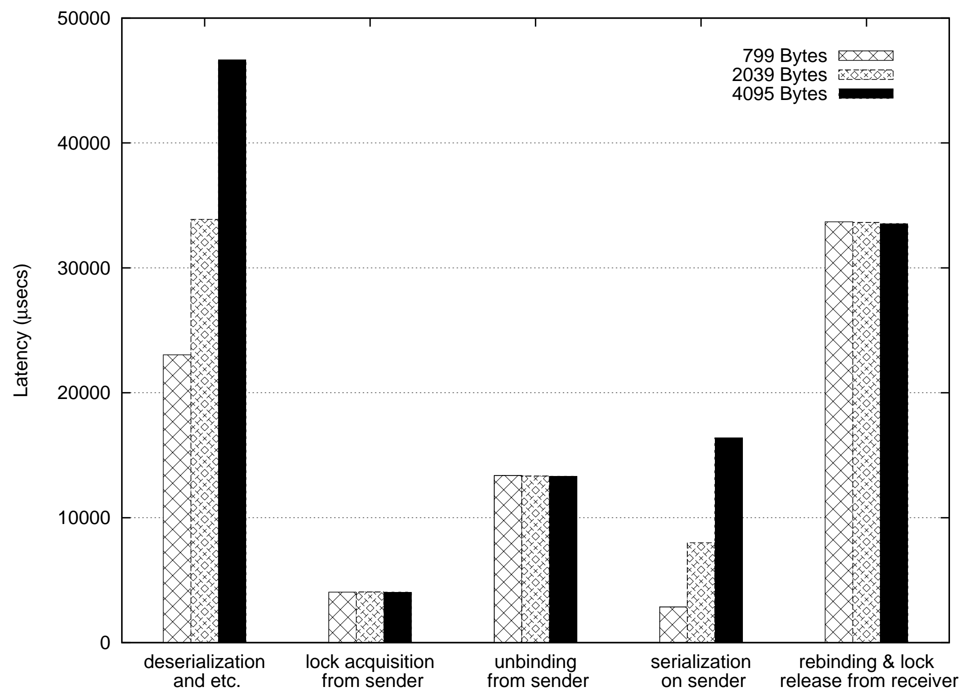

Fig. 28. Decomposition of Latency for a Lookup with a Migration in Terms of Serialized Object Size

latency, and rebinding latency, do not change as the size of serialized agile object increases. Therefore, a predictable latency for remote method invocation is achieved even with agile object's migration.

Figure 29 shows the comparison of average and standard deviation of latencies for a lookup with a migration in various execution environments. In this experiment the size of serialized agile object is 4095 bytes, and the agile object consumes 22% of CPU utilization. In Figure 29 each label in horizontal axis stands for the following.

- "None": there is no other workload both in the source and destination's VMs for migration except the agile object.
- "BG-25": one background Java thread is running in the source's VM while another background Java thread is running in the destination's VM. Each background Java thread consumes 25% of CPU utilization. They are not under

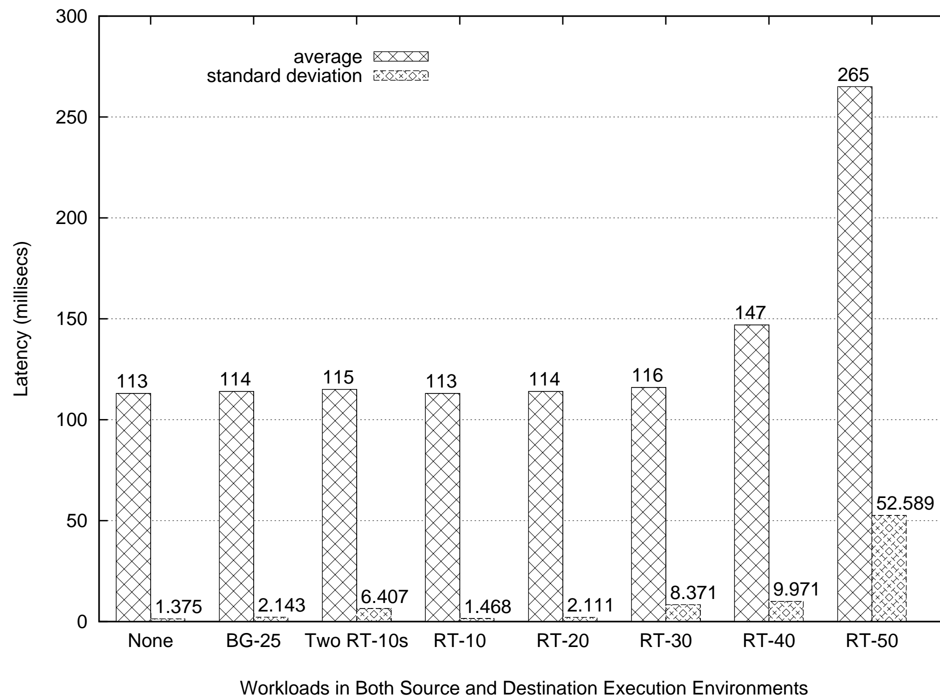

Fig. 29. Latency for a Lookup with a Migration with Varying Amount of Background Load

control of the EDF scheduler.

- “Two RT-10s”: two real-time Java threads are running in the source’s VM while two other real-time Java threads are running in the destination’s VM. Each real-time Java thread consumes 10% of CPU utilization.
- “RT-10”: one real-time Java thread is running in the source’s VM while another real-time Java thread is running in the destination’s VM. Each real-time Java thread consumes 10% of CPU utilization.
- “RT-20”: one real-time Java thread is running in the source’s VM while another real-time Java thread is running in the destination’s VM. Each real-time Java thread consumes 20% of CPU utilization.
- “RT-30”: one real-time Java thread is running in the source’s VM while another real-time Java thread is running in the destination’s VM. Each real-time Java

thread consumes 30% of CPU utilization.

- "RT-40": one real-time Java thread is running in the source's VM while another real-time Java thread is running in the destination's VM. Each real-time Java thread consumes 40% of CPU utilization.
- "RT-50": one real-time Java thread is running in the source's VM while another real-time Java thread is running in the destination's VM. Each real-time Java thread consumes 50% of CPU utilization.

As can be seen on Figure 29, the average of the latencies very slightly increases as the CPU utilization of the real-time Java thread increases up to 30%. However, there are increases of 30% and 135% in average latency for a lookup with a migration when each real-time Java thread consumes 40% and 50% of CPU utilization in both source and destination execution environments, respectively. In addition, the standard deviation of the latencies increases up to 52.589 milliseconds.

Furthermore, Figure 30 shows the decomposition of the latencies of the same experiment as Figure 29. As can be seen in Figure 30, the averages of the latencies for "lock acquisition from sender", "unbinding from sender", "serialization on sender", and "rebinding & lock release from receiver" vary little as the CPU utilization of the real-time Java thread increases up to 40%. However, the average of the latencies for "deserialization and *etc.*" contributes to the large increase of the lookup latencies when the real-time Java thread consumes 40% and 50% of CPU utilization.

G. Discussion of Experimental Results

Our experimental results show how well the Agile Objects System guarantees the predictable lookup latency for a migrating RMI server in the worst-case scenario. We also demonstrate that the lookup latency for the migrating RMI server is pre-

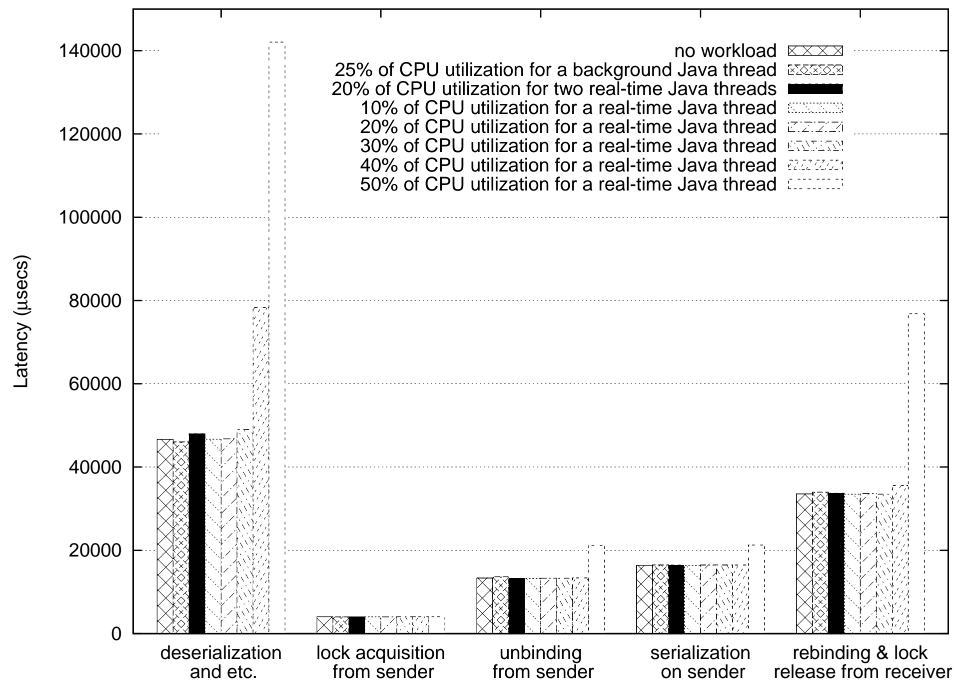

Fig. 30. Decomposition of Latency for a Lookup with a Migration with Varying Amount of Background Load

dictable even in conditions with varying amount of background load. In the support of predictable worst-case lookup latency for a migrating RMI server, we guarantee predictable worst-case execution times of the remote method invocations to the migrating RMI server.

CHAPTER VI

RESOURCE MANAGEMENT

A. Introduction

In our view, the methodology for application-survivability based on run-time software component migration should satisfy four requirements: transparency, independence from the heterogeneity of underlying systems, post migration communication capability, and fast migration. By *transparency* we mean that the application programs are not aware of migration. In other words, application programs are developed on a single processor environment and the supporting tools take care of remote invocation and run-time migration. *Independence of the heterogeneity of underlying systems* means that the migration should be able to be performed among heterogeneous machines. Heterogeneity includes not only the hardware platform but also the operating system. *Post migration communication capability* means that the migrated component, once in a new destination, should be able to communicate with other components just the same as before the migration. Otherwise, the migration will be limited to stand-alone software only. Lastly, in order to support low-latency migration, we focus on the two dominant factors: lightweight migration and proactive resource discovery. The former is to minimize the absolute amount of time required for migration and the latter is to provide a destination host information at the time of migration decision that is able to accommodate migrating components. The proactive resource discovery therefore requires a minimum amount of time in finding available host when a migration decision is made.

Many methodologies satisfy parts of these requirements; however, in our survey, we have not found a run-time software migration methodology that satisfies all four

requirements above. In our methodology, the proactive resource discovery consists of a protocol for the dissemination of demand and availability of resources (we call this the Community protocol) and associated algorithms. Based on a combination of pull-based and push-based approaches, this methodology limits resource discovery activities whenever the system is overloaded so that it can avoid dumping overwhelming resource discovery messages. It encourages such activities when hosts are available. Fast migration in this context imposes two unique challenges on resource discovery: First, the objective of resource discovery in our system is to locate *dynamic* resources like CPU bandwidth and memory space. This as opposed to discovery of static resources (servers, *etc.*), which is in principle a naming issue and can be addressed with traditional naming mechanism. Second, resource availability information needs to be collected in a *proactive* fashion. This requires in practice that a host needs to have an accurate picture of the resource availability for at least some number of hosts. This chapter of the thesis investigates the relationship between resource discovery and successful migration in distributed real-time systems. We propose REALTOR (REsource ALlocaTOR) as part of our methodology and also as a solution of proactive resource discovery in distributed real-time systems. The two ideas above, lightweight migration and proactive resource discovery, have been implemented as a middleware that also provides real-time job scheduler in Java Virtual Machine (VM), and a naming server.

Our analysis and simulation in a cluster computing environment show that the proactive resource discovery requires very low communication overhead while maintaining high effectiveness in finding available CPU resources. Our implementation and experimental measurements show that run-time component migration based on our approach takes much less time compared to similar approaches based on reactive resource discovery.

B. Related Work

We observe that resource discovery in distributed systems has evolved in three broad directions: Peer-to-peer, Grid systems, and distributed real-time systems. In peer-to-peer systems resource sharing has been rather focused on static resources, such as [43, 44]. There have been many proposals for algorithms and protocols for static resource discovery. Representative examples of peer-to-peer file sharing systems including Gnutella, Chord, CAN, PAST, Tapestry, and Freenet are compared in [45]. However, they are not directly applicable to our requirements imposed by fast migration, as static location information is not sufficient to locate the (dynamic) availability of CPU bandwidth, for example. Resource discovery in Grid systems, on the other hand, has addressed dynamic resources [46] since computation oriented scientific collaborations have been one of the major applications of such systems. We survey representative research concerning resource management for distributed computing systems.

Globus encompasses many research issues under the name of "virtual organization", which is primarily a coordinated large-scale dynamic resource sharing and problem solving system over multi-institutions. As resource discovery has clearly been identified a challenging problem, Globus has developed its own resource management architecture, Globus Architecture for Reservation and Allocation (GARA) [47]. Unlike per-session on-demand resource reservation (RSVP [48], for example) GARA focuses on advance reservations and co-allocation with which it can easily enhance end-to-end Quality of Service (QoS) [49]. In this project, a resource discovery based on the peer-to-peer model has been proposed [50], that consists of a few request-forwarding algorithms in a fully decentralized architecture accommodating heterogeneity and dynamism in resource. Legion provides another distributed computing

infrastructure in very large-scale systems. In its resource management [51], however, the prime interest was in supporting and matching user task requirements. Interfaces based on object-based design are well defined accordingly for heterogeneous resource types as well as the resource allocating procedure. Condor [52] provides resource management services that harness the capacity of very large collections of distributively owned UNIX workstations. The need for maximum computation throughput has been the driving force for the efficient utilization of distributed computational resources [53], and a metric, "Goodput" has been proposed for co-scheduling CPU and network capacity [54]. For resource discovery, a frame-work "Matchmaking" has been proposed [55], that separates matching and claiming phases of resource allocation.

None of this work has addressed the *effectiveness* of resource discovery and allocation, a measure that is of particular importance in distributed real-time systems. By effectiveness we mean the ability of the resource discovery system to find and allocate available resources in overload situations, and so avoid situations where re-configuration fails due to lack of resources. Interestingly however, many papers addressing resource discovery do not consider effectiveness when evaluating their proposals. Our methodology proposes an effective dynamic resource discovery.

In the distributed real-time systems area, distributed scheduling using bidding and focused addressing [56, 57, 58] takes real-time tasks' timing and resource requirements into account for scheduling decisions. If a task on a node cannot be locally guaranteed by its deadline, a bidding mechanism kicks in: a *bidder* on the node determines where the task should be sent. Determining such a node is a two-step process: focussed addressing followed by bidding so that it can be completed in time. In *focused addressing* the task is sent to another node based on periodically disseminated availability information from other nodes in the system. If there is no such focused node, or if the focused node may fail to guarantee the task, bidding can be executed

to guarantee the real-time properties of the task. In *bidding* the node sends out a *request-for-bid* message to other nodes. Nodes with enough resources for meeting the task's requirements respond with a bid reflecting their surplus. The task is sent to the node that offers the best bid. We will see that this bidding mechanism is similar to our methodology except that it is reactive.

C. REALTOR: REsource ALlocaTOR

The resource discovery and allocation system must satisfy a number of requirements: First, the resource availability information must be readily available at any time so that any host under attack or malfunction is able to locate a host and move the software components immediately. Any resource allocation scheme must be *proactive*, as nodes are in need for migration because of attacks, for example. Second, any resource discovery scheme for this type of systems considered here must be largely *state-less*. Nodes leave and join the system at any time, due to attacks and failures, or after recovery. Similarly, the overhead of nodes (re-)joining the system must be low. In REALTOR we rely heavily on *soft state*, which is re-freshed at low cost in order to retain an accurate view of resource availability in the system. Third, the protocols must be largely *idempotent*, so that node failures do not give rise to errors. All messages in REALTOR are refresh messages, which are idempotent in nature. Finally, given the large amount of dynamics in the system and the need to support scalability without loss of information accuracy, the resource discovery mechanisms at any node should interact only with a *small subset of other nodes*. We use the concept of *community* in REALTOR, which links a potential resource user with a community of potential resource providers. Communities are ephemeral in nature: they spontaneously appear, change over time, based on resource requirements,

resource availability at the nodes in the community, and the status of nodes in the community. We describe REALTOR along with four other resource discovery schemes for performance comparison purposes.

1. REALTOR Scheme

Each host establishes its own community for future software component migration, which is a set of nodes able to receive a migrating component. Each host is free to join as many communities as it is able to without over-allocating its spare resources. Therefore, each host usually owns one community and is a member of several other communities. The membership in a community is not static, and must be refreshed. The membership of a node in a community is valid only for the interval between two consecutive refresh messages. So, in order to maintain the membership to a community, a host needs to keep responding to all refresh messages from the organizer. When a member stops responding to refresh messages from the organizer, it de facto leaves the community. Similarly, when a community organizer stops sending refresh messages, the community will naturally disband.

a. Community Protocol

The community protocol was designed with three goals, 1) the protocol should be *effective* in finding available resources within its own community, 2) the protocol overhead should be independent of network size, and 3) the protocol should be *stateless*. Therefore, Community protocol has only two types of messages.

- *HELP:*
  - Message Format: HostID (community organizer identifier), Type (help), and Seq_No (sequence number of help).

- When a host joins the system, it begins to build its own community. A community is typically a subset of the whole system. The invitation to the new community is done by broadcasting a HELP message to the network. The interval between two consecutive HELP messages is defined by *Algorithm H* below. Networks in this context are typically application-level overlay networks. We assume that there is an authorization and join protocols for a host to join the system so that the host can be reached by HELP messages by existing hosts.

- *PLEDGE:*
  - Message Format: HostID (identifier of the pledger), Type (pledge), Seq_No (sequence number of help), and Resource availability (degree).
  - When a host receives a HELP message, it determines whether to join or not the community. Once it determined to do so, it sends a PLEDGE message to the community organizer (*i.e.*, the originator of the HELP message) whenever its resource usage status changes across a threshold level. The threshold level is determined by *Algorithm P* below at each local host.

b. Algorithm H

As can be seen in Figure 31, a host keeps sending a HELP message at every HELP_interval as long as a task arrives and its resource usage is above a threshold. The length of HELP_interval changes over time depending on the success rate in finding available resources. If it succeeds, HELP_interval is decreased by the proportional amount of *beta* as a reward, while it increases the interval by the proportional amount of *alpha* as a penalty. The idea is to avoid unnecessary resource discovery activities when the whole system is heavily loaded. Upper_limit prevents an unbounded increase of

```
Algorithm  H
Input: Time_current, Time_sent
Output: HELP message

Task A:
Whenever a task arrives   do {
        If resource usage would exceed a threshold level  {
                If ((T_current – T_sent) > HELP_interval) {
                        send HELP ;
                        set_timer;
                }
        }
}

Task B:
Timeout do {
        If ((HELP_interval + HELP_interval * alpha) < Upper_limit)
                HELP_interval +=  HELP_interval * alpha;
}

Task C:
Whenever a PLEDGE message arrives do {
        If the corresponding timer is not expired
                cancel_timer;
        Update corresponding PLEDGE list;
        If a node is found for migration and HELP_interval has not been updated for the HELP {
                If ((HELP_interval – HELP_interval * beta) > 0)
                        HELP_interval –= HELP_interval * beta;
        }
}
```

Fig. 31. Algorithm H in REALTOR

HELP_interval after a series of failure in finding available resources. The speed of expansion or shrinkage is controlled by appropriately setting alpha and beta values.

c. Algorithm P

As can be seen in Figure 32, the host replies with a PLEDGE as long as a HELP message arrives and its resource usage is below the threshold level. Also, once a host determines to be a member of a community, it replies with PLEDGE messages whenever its resource usage status changes across the threshold level. This

```
Algorithm P
Input: HELP message
Output: PLEDGE message

Task A
Whenever a HELP message arrives do {
        If the host has used its resource less than a threshold level
                Reply PLEDGE;
}

Task B
Whenever the resource availability changes across the threshold level do {
        send PLEDGE to the community organizers;
}
```

Fig. 32. Algorithm P in REALTOR

helps the organizer keep the most current information. Task B in Figure 32 sends a PLEDGE to a set of community organizers from which the host has received a HELP. When a community organizer leaves the system, a host which has kept sending a PLEDGE to the organizer will learn the leave by calculating the time lapse from the last HELP message from the organizer to the current time. If the difference is larger than *Upper_limit* (Figure 31), the host removes the organizer from the list. The calculation is performed whenever Task B is invoked. The host, therefore, is able to maintain a list of community organizers in need.

2. Other Resource Discovery Schemes

a. Pure PUSH Scheme

In this scheme, each host disseminates its own resource availability information to its neighbors unconditionally at every preset interval. In comparison to REALTOR, there is only periodic PLEDGE message without HELP.

b. Pure PULL Scheme

In this scheme, each host solicits PLEDGE from its community members whenever 1) a task arrives and 2) the resource usage level is beyond a threshold level. In comparison to REALTOR, this scheme generates HELP messages unlimitedly.

c. Adaptive PUSH Scheme

In this scheme, each host disseminates its own resource availability information to its neighbors whenever the resource usage changes across a threshold level. In comparison to REALTOR, PLEDGE is automatically generated at each major status change without solicitation (HELP).

d. Adaptive PULL Scheme

In this scheme, each host solicits PLEDGE from its community members whenever 1) a task arrives, 2) the resource usage level is beyond a threshold level, and 3) a time window has passed since the previous HELP. In comparison to REALTOR, it invokes PLEDGE exactly once for each HELP.

D. Analysis of Resource Discovery Message Overhead

In this section, we compare the message overhead of each resource discovery scheme introduced in the previous section by an analysis based on modeling of the task queue at each host because the number of message exchanges directly depends on the task queue status at each node.

For simplicity, without losing generality, we assume that: 1) with a given community of N homogeneous nodes, the task arrival at each node forms a Poisson process independently of each other, so the inter-arrival time is exponentially distributed,

2) the task execution time at node is another exponential distribution, 3) the maximum number of tasks waiting at the queue at a node to be executed is limited to K, so the tasks arriving at a node whose queue is already full will be either discarded or migrated to another node. Then we can model the queue at each node as $M/M/1/K$ [59]. Under the condition that the average task size is much smaller than the size of the task queue, the K-storage model well reflects the queuing behavior at each node.

Then the time-independent probability that k tasks are waiting in the queue to be executed is given by:

$$P_k = \begin{cases} \frac{1-\frac{\lambda}{\mu}}{1-\left(\frac{\lambda}{\mu}\right)^{K+1}} \left(\frac{\lambda}{\mu}\right)^k & \text{if } 0 \le k \le K \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

Where λ represents the task arrival rate and μ represents the task departure (service) rate. We are interested in the probability that there are more than t tasks in the queue waiting to be executed, which is given by:

$$P_h = \sum_{k=t+1}^{K} P_k = \sum_{k=t+1}^{K} \frac{1-\frac{\lambda}{\mu}}{1-\left(\frac{\lambda}{\mu}\right)^{K+1}} \left(\frac{\lambda}{\mu}\right)^k \tag{6.2}$$

Likewise, the probability that there are less than or equal to t tasks in the queue waiting to be executed is given by:

$$P_l = \sum_{k=0}^{t} P_k = \sum_{k=0}^{t} \frac{1-\frac{\lambda}{\mu}}{1-\left(\frac{\lambda}{\mu}\right)^{K+1}} \left(\frac{\lambda}{\mu}\right)^k \tag{6.3}$$

Now we analyze the message overhead in terms of number of message exchanges. First, we suppose that Δt is the minimum time interval to observe a state transition of the queue, for example, from k to k+1 or vice versa. This means that Δt is small enough, therefore, in this time interval, there can be only one state transition

at most. Second, we analyze the message overhead at a single node because the total overhead in a given community is the total sum of message overhead at each node. Lastly, we consider Pure PUSH as the benchmark because it monitors dynamic resource availability at every possible moment, which is Δt in our analysis model. We, therefore, compare the message overhead of each scheme to that of the pure PUSH. In the following B and U represent the number of broadcast and unicast messages respectively.

Pure PUSH: In a pure-PUSH approach, a node broadcasts a resource availability information message at every Δt. A node in a pure-PUSH approach therefore sends 1 broadcast message and receives $(N-1)$ unicast messages at every Δt.

Pure PULL: In this approach, HELP message is broadcast by a node when the queue is occupied by more than a threshold, for example, k tasks. The probability that there are more than k tasks at the queue for Δt is the same as Equation (6.2) since it is time independent. So, the number of HELP messages generated by a node within Δt is P_h. On the other hand, since each other node replies with PLEDGE when there are less then t tasks in the queue, the number of PLEDGE messages by the rest of nodes in the community is given by $(N-1)P_hP_l$. So, the total number of message exchanges at a node is $P_hB + P_hP_l(N-1)U$.

Adaptive PUSH: Because this approach broadcasts a resource availability message when the queue status change across a threshold in Δt, we look at the probability of the change first. Obviously, there are only two cases for the status change across the threshold: from a higher state than the threshold to a lower state or vice versa. The probability P_c of status transitions across the threshold in Δt, therefore, is the sum of the two transition cases: from $k+1$ to k or vice versa. Therefore, we have:

$$P_c = P_{t+1}P_t + P_tP_{t+1} = 2P_tP_{t+1} \tag{6.4}$$

Because the rest of the nodes in the community do the same job in the same way, the total number of message exchanges at a node is given by $P_cB + P_c(N-1)U$.

Adaptive PULL: In order to generate a HELP message for a node in this approach, there should be more than t tasks in the queue at t_0 and $t_0 + W$. W is a time window preset for an adaptive PULL. The number of HELP messages generated by a node in W is, therefore, P_hP_h for $W = \Delta t$, otherwise 0 for $W > \Delta t$. PLEDGE messages from the rest of the community nodes are then given by $P_l(N-1)$ or 0 (no PLEDGEs for no HELP). The total number of message exchanges at a node for Δt, therefore, is given by $P_hP_hB + P_hP_l(N-1)U$ for $W = \Delta t$ or 0 for $W > \Delta t$.

REALTOR: REALTOR is a combination of an adaptive PULL and adaptive PUSH. The number of HELP messages generated by a node is the same as that of the adaptive PULL: P_hP_h, for $W = \Delta t$, otherwise 0 for $W > \Delta t$. The number of PLEDGE messages from the other nodes is the same as that of the adaptive PUSH: $2(N-1)P_tP_{t+1}$. The total number of message exchanges at a node for Δt, therefore, is $(P_h)^2B + 2P_tP_{t+1}(N-1)U$ for $W = \Delta t$ or 0 for $W > \Delta t$.

Table II shows the number of message exchanges at a node in W, where $W = n\Delta t$.

Theorem 1 *The order of message overhead of the five approaches is given by:*

$$purePUSH \geq purePULL \geq REALTOR \;\;\&\;\; adaptivePUSH \geq adaptivePULL$$

Proof

Case 1: where $W = \Delta t$.

First, we consider the number of HELP messages. As can be seen in the second row of Table II, pure-PUSH is obviously larger than pure PULL, adaptive PULL, and REALTOR because both P_h and P_l are smaller than 1. So, we compare adaptive

Table II. Message Overhead ($W = n\Delta t, (n = 1, 2, 3, \ldots)$)

item	HELPs (n=1)	PLEDGEs (n=1)	HELPs ($n > 1$)	PLEDGEs ($n > 1$)
pure PUSH	1	N-1	n	$n(N-1)$
pure PULL	P_h	$(N-1)P_hP_l$	nP_h	$n(N-1)P_hP_l$
adaptive PUSH	$2P_tP_{t+1}$	$2(N-1)P_tP_{t+1}$	$2nP_tP_{t+1}$	$2n(N-1)P_tP_{t+1}$
adaptive PULL	P_hP_h	$P_l(N-1)$	P_hP_h	$P_l(N-1)$
REALTOR	P_hP_h	$2(N-1)P_tP_{t+1}$	P_hP_h	$2n(N-1)P_tP_{t+1}$

PUSH and pure PUSH because the second row does not show clearly which one is bigger ($2P_tP_{t+1}$ and 1). In order for adaptive PUSH to be bigger than 1, P_tP_{t+1} should be larger than $\frac{1}{2}$. In order for P_tP_{t+1} to be larger than $\frac{1}{2}$, in turn, the following condition should be met:

$$P_t > 0.5 \ \ and \ \ P_{t+1} > 0.5 \tag{6.5}$$

However, another condition should be met because $K > 2$ in our assumptions:

$$P_t + P_{t+1} < 1 \tag{6.6}$$

These two conditions are, obviously, contradictory, cannot be met at the same time. Therefore, adaptive PUSH cannot be larger than 1. So pure PULL has the largest number of HELP messages for $W = \Delta t$.

Second, we compare the number of PLEDGE messages. As seen in the table, it is obvious again that pure PULL, adaptive PULL, and REALTOR are all smaller than pure PUSH because neither P_h nor P_l is larger than 1. By the same argument as in HELP message comparison, adaptive PUSH cannot be larger than pure PUSH. Therefore, as long as the message overhead is concerned, pure PUSH requires the highest cost.

Case 2: where $W = n\Delta t\,(n = 2, 3, 4, \ldots)$.

As can be seen in the 4th and 5th rows of Table II, it is obvious again that pure PUSH has the biggest message overhead. Assuming that the number of PLEDGE messages is the dominant factor of the message overhead as n gets larger, adaptive PULL becomes the lowest because it does not scale with n unlike others. Since adaptive PUSH and REALTOR have the same amount of PLEDGE messages, the ordering problem is reduced to that between

$$P_h P_l \tag{6.7}$$

for pure PULL and

$$2P_t P_{t+1} \tag{6.8}$$

for adaptive PUSH and REALTOR. In order to compare the two probabilities (6.7) and (6.8) we change the form of the first one like below.

$$\begin{aligned} P_l P_h &= (P_0 + \ldots + P_t)(P_{t+1} + \ldots + P_K) &(6.9)\\ &= P_0 P_{t+1} + P_0 P_{t+2} + \ldots + P_t P_{t+1} + \ldots + P_t P_{K-1} + P_t P_K &(6.10) \end{aligned}$$

So the comparison is changed to that of Equation (6.10) and (6.8). Further, since Equation (6.10) has the term of $P_t P_{t+1}$ in it, the comparison is reduced to that of

Equation (6.11) below (Equation (6.10) - $P_t P_{t+1}$) and $P_t P_{t+1}$.

$$\begin{aligned} & P_h P_l - P_t P_{t+1} \\ = \quad & P_0 P_{t+1} + P_0 P_{t+2} + \ldots + P_{t-1} P_K + P_t P_{t+2} + \ldots + P_t P_{K-1} + P_t P_K \quad (6.11) \end{aligned}$$

Considering that the term $P_t P_{t+1}$ is only a part of Equation (6.10), it is unlikely that $P_t P_{t+1}$ alone is comparable to the rest of Equation (6.10). However, in order for $P_t P_{t+1}$ to be larger than Equation (6.11), the term should be very larger compared to the other terms in Equation (6.10). This could happen only when the average number of tasks waiting at the queue is t or $t+1$ so that the other terms in Equation (6.10) is negligible. These are the only special cases where $P_t P_{t+1}$ could be larger than or equal to Equation (6.11). Otherwise, in general, Equation (6.11) is larger than $P_t P_{t+1}$. So, pure PULL has larger overhead than both adaptive-PUSH and REALTOR. Therefore, the order of message overhead given by Theorem 1 is correct. □

E. Experimental Performance Evaluation

In this section, we evaluate the performance of REALTOR with a comparison to those of the alternative resource discovery protocols introduced previously, under increasing load, using a set of simulation experiments. We measure the performance in terms of *message overhead* and *effectiveness* in finding available resources.

For the experiments, we simulate the mesh topology displayed in Figure 33, with 25 nodes and 40 links. Each intersection represents a node. For fair comparison purposes, in this section we assume that the topology represents the limited scope of neighbors for REALTOR and all other four resource discovery schemes. In reality, there should be a mechanism in place limiting the scope of neighbors for REALTOR. we randomly generate tasks at increasing rates, and assign them randomly to a node.

The resource discovery and allocation algorithms then must migrate the tasks, when needed, to nodes with available CPU capacity. [1]

We generate tasks with exponentially distributed lengths of a mean value μ, an execution time. The generated task is given to a node randomly selected from Node 0 through Node 24. The task arrival forms a Poisson process with a rate of λ. Each node is assumed to have a single queue to process tasks. The single queue has

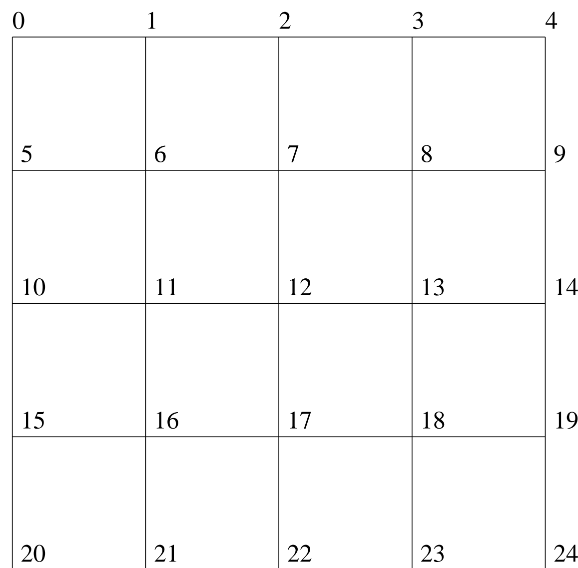

Fig. 33. The Network Topology for the Simulation

length of 100 seconds and the mean job size is 5 seconds. Tasks arriving at a node whose queue is already full are supposed to migrate to another node whose queue can still accommodate the task. The new destination the migrating task is supposed to move to, is assumed to be readily provided by the resource discovery approaches. In

[1] In this simulation, we assume a single resource - CPU. More general resource scenarios such as network bandwidth, current security level, *etc.*, would give similar results.

these experiments, in order to satisfy the requirement of *proactiveness* for immediate migration, we measure the performances of the five approaches with only a one-time migration try to the best candidate destination node provided by each approach. So, if the candidate destination node cannot accommodate the migrating task, then the task is rejected.

For this simulation, we use a simple threshold strategy for both *Algorithm H* and *P*. *Algorithm H* sends out a HELP message when the queue length exceeds a certain level. The queue level is checked whenever a new task arrives. So, the HELP messages are sent out whenever the three conditions are met: 1) a new task arrives, 2) the queue including the new task exceeds a certain level, and 3) the time window has passed. Likewise, *Algorithm P* replies HELP with PLEDGE whenever the two conditions are met: 1) the time interval between the last HELP from a node and current time is less than Upper_limit (Figure 31), and 2) the queue is occupied below a certain preset level. The total number of messages is counted as the sum of HELPs + PLEDGEs and communication for migration between admission controls. In the following figures in this section, the curve names stand for the following.

- "Pull-.9": a pure PULL approach which uses 0.9 for both Algorithm H and P.
- "Push-1": a pure PUSH which uses 1 second periodic interval for information dissemination.
- "Push-.9": an adaptive PUSH which disseminates information only when the resource usage changes across the threshold level of 90%.
- "Pull-100": an adaptive PULL which limits HELP_interval from increasing infinitely, in this case the limiting value is 100 time units (Upper_limit in Figure 31).
- REALTOR: combination of "Push-.9" and "Pull-100".

- Algorithm H 0.9: every new task arriving a queue whose length reaches more than 90% including the new task triggers broadcasting of a HELP message.
- Algorithm P 0.9: means that every HELP message arriving a node whose queue is occupied less than 90% triggers a PLEDGE message.

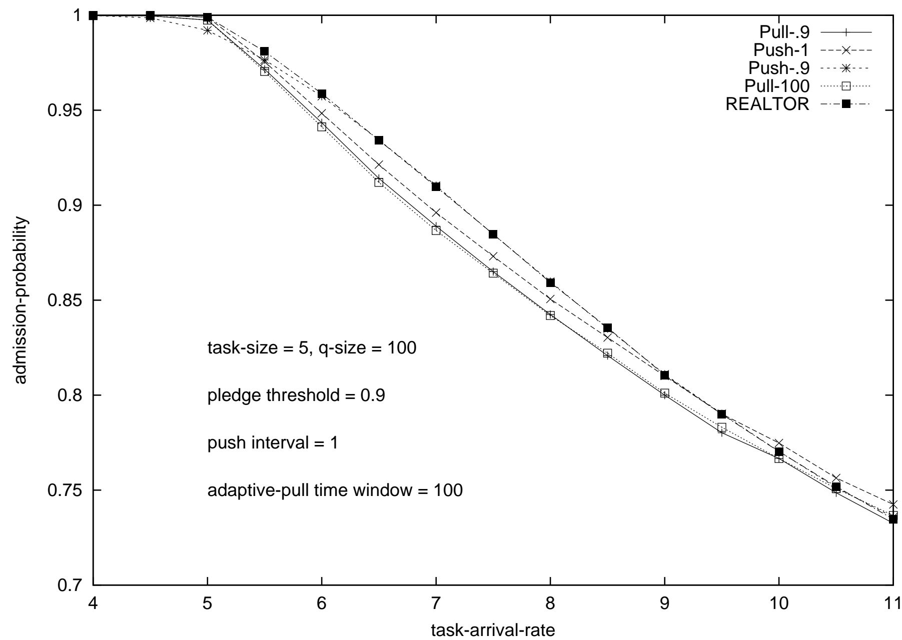

Fig. 34. Admission Probability

Figure 34 compares the task admission probability of the five approaches. The x-axis represents the task arrival rate, and y-axis shows the admission probability during the experiments. At the task arrival rate of 7, for example, more or less 90% of the tasks are admitted, so roughly 10% tasks are rejected. We limit the task arrival rate to 11 because after that a significant portion of input tasks, more than 30%, are rejected, which, we believe, not a normal situation in practice. This set of curves is obtained this way. First, we run this simulation for Push-1. After obtaining the curve

"Push-1", we repeatedly run the simulation for other approaches with different set of simulation parameters until finally we have a set of curves close enough to "Push-1". So, as seen in Figure 34, "Push-1" lies in the middle of the curves for a large portion of the rates. "REALTOR" and "Push-.9" shows the best performance for most of the range. "Pull-100" and "Pull-.9" show the worst performances. We consider that these curves are close enough to assume that they show more or less the same performance that provides the ground on that we can compare the communication overhead for the same performance.

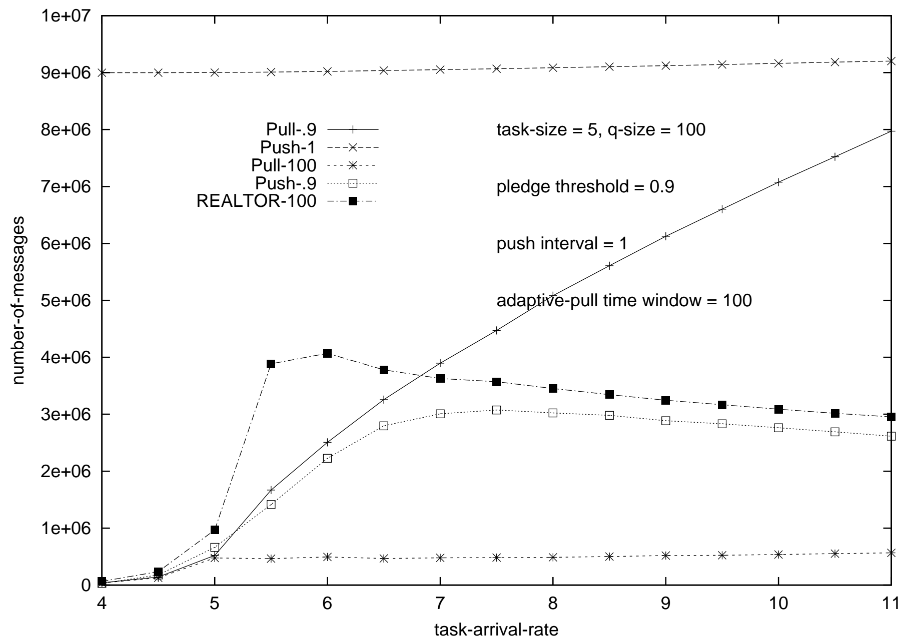

Fig. 35. Number of Messages Exchanged

Figure 35 shows the communication overhead of the five approaches. Y-axis shows the total number of message exchanges during the experiments. As we ex-

pected, "Push-1" shows the highest overhead, especially under lightly loaded conditions where it wastes too much communication bandwidth unnecessarily. "Pull-.9" (pure PULL) keeps increasing its overhead as the system gets overloaded. As it increases almost linearly, it will eventually cross "Push-1" for high rates of λ. However, "Pull-.9" is still below "Push-1" until the admission probability drops to below 0.75, after which the system will be completely overloaded. It is apparent that the pure PULL approach also wastes much communication bandwidth as the system gets overloaded. On the other hand, "Pull-100" shows the least amount of communication overhead independently from the load. However, this, in return, has a relatively lower performance curve in admission probability in Figure 34. "Push-.9" (adaptive PUSH) shows a moderate overhead and a very close performance to "Push-1" (pure PUSH). Finally REALTOR shows the best performance in admission probability with still a moderate overhead slightly higher than "Push-.9". This result is expectable because REALTOR combines the two approaches: an adaptive PUSH and an adaptive PULL, so it naturally takes advantages of both while adding a slight amount of communication overhead.

Figure 36 compares the resource discovery protocol overhead per admitted task. For example, "Push-1" costs 200 message exchanges for a single admitted task at λ = 5, while all other approaches take about less than 50. The amount of overhead in REALTOR and "Push-.9" decreases as the system becomes overloaded. This is because 1) HELP_interval is kept at the maximum (*Upper_limit* in Figure 31) due to the repeated failure of finding available resources, and 2) since the resource usage level at each host is kept above the threshold level. The reason for the peak around $\lambda = 6$ in REALTOR is that the resource usage level changes across the threshold most frequently around that point. Adding a hysteresis around the threshold would greatly diminish this effect. The admission probability at that point is about 0.95,

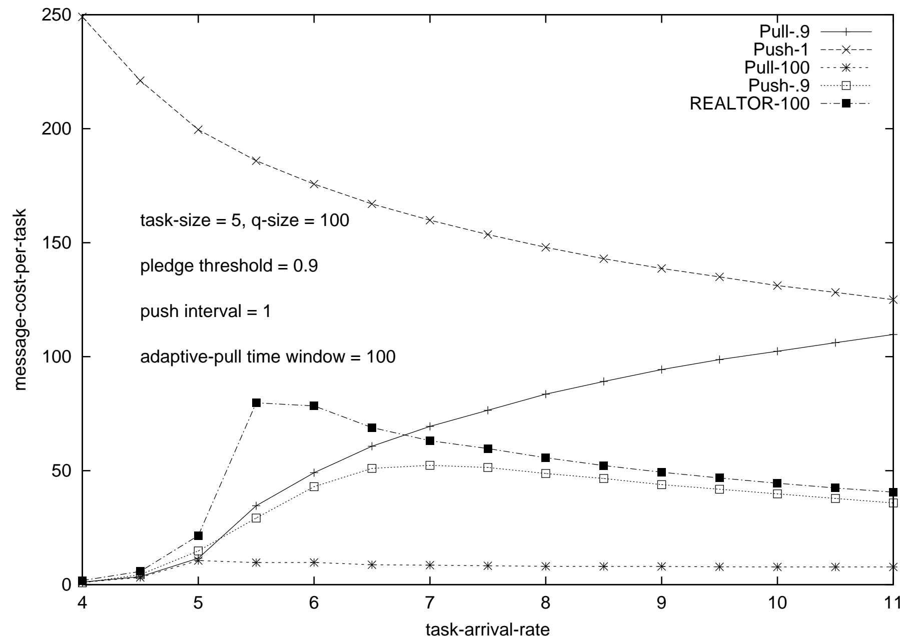

Fig. 36. Communication Cost per Admitted Task

which means there are a lot of fluctuations in usage levels, causing PLEDGE messages to be generated. This figure clearly illustrates the cost of disseminating resource information periodically regardless of load conditions.

Through the set of experimentations we confirm that: 1) pure push-based approaches waist communication resources too much during light-load conditions, 2) pure pull-based approaches also waist communication resources much in overload conditions, 3) REALTOR performs best in terms of *overhead* and *effectiveness* for any load conditions.

F. Implementation Experience

In this section, we evaluate the effectiveness of our methodology for fast migration within the Agile Objects System described in Chapter II. Here, the performance is

measured in terms of *migration time*.

In this experiment, we compare two systems: *REALTOR*-based and *REACTIVE*-based. In the REALTOR-based system, resources are discovered by REALTOR, and in the REACTIVE-based system, resources are discovered on-demand, *i.e.*, whenever a task needs to migrate. In the latter, the resource availability information is solicited in a general form of request-and-reply protocol by the host that tries to migrate a task. Both systems, however, choose a host, which replies the highest resource availability. This selection strategy is to maximize the resource utilization. Therefore, the only difference in the two systems is in the resource discovery schemes.

We used a workstation cluster of Linux machines at the Concurrent Systems Architecture Group Laboratory in the University of California San Diego, where Agile Objects System has been integrated. The cluster for this measurement consists of 25 homogeneous hosts running Redhat Linux Version 7.2 Operating System on Pentium II at 450 MHz. Each host is a single server that processes arriving tasks sequentially. Both REALTOR-based and REACTIVE-based systems use IP multicasting for HELP messages and User Datagram Protocol (UDP) for PLEDGE messages.

For the experiment, a stream of tasks is generated and distributed to hosts randomly. The arrival of the tasks forms a Poisson function with mean value λ. Each task, like in the simulation study, executes for a fixed amount of time on the server. Once a host is overloaded, it begins migrating the newly arriving tasks as long as it remains overloaded. Once the migration begins, the other hosts in the cluster accommodate the migrating tasks for a while until they are overloaded too. We implement each task as a timer waiting to expire. We have generated 4,000 tasks for each task arrival rate and measured admission probability and migration time. Like in the simulation study, the task queue size is fixed to 100 seconds at each node, and the mean job size is 5 seconds.

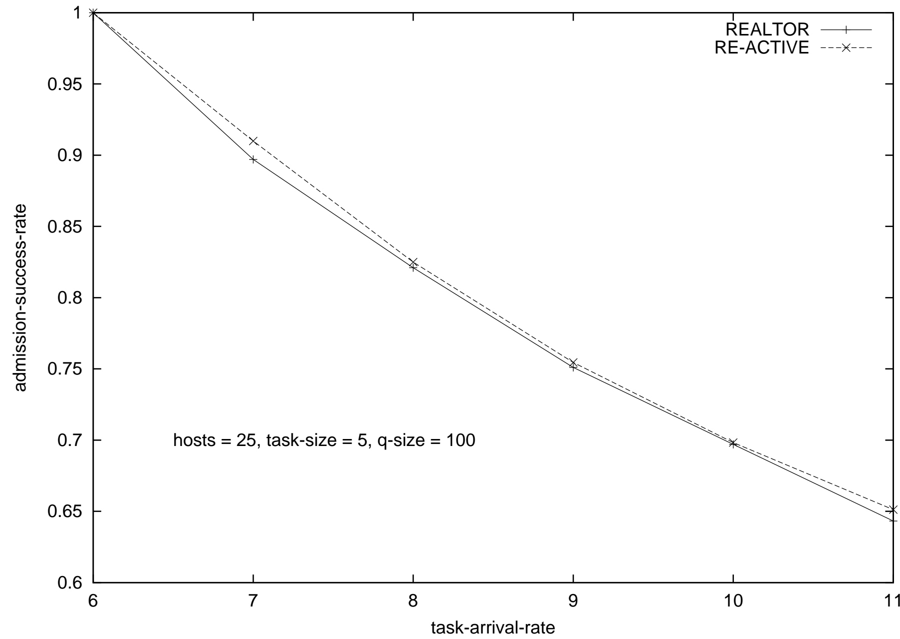

Fig. 37. Admission Success Rate

As shown in Figure 37, like in the simulation study, we first set up the experiment environment so that the admission probabilities in both systems are only slightly different.

Figure 38 shows the average migration time between two systems in this situation. The migration time of both systems is defined the time between when a host decides to migrate a task to when a best destination candidate node receives the task. As seen in Figure 38, REACTIVE takes about 12 to 13 milliseconds for a task migration while REALTOR takes about 6 to 7 milliseconds. So, the migration time is different by 5 to 6 milliseconds. In terms of percentage, however, REACTIVE is 200% larger. The migration time does not change much with different input task arrival rates since the migration time is measured only when the migration attempt is succeeded. We interpret the difference in the migration time as follows. In emergencies,

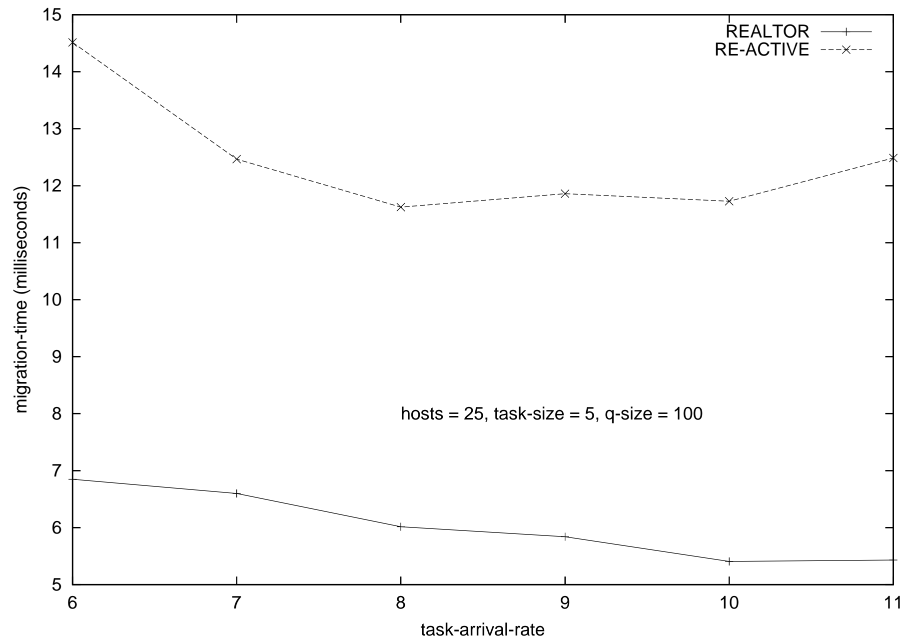

Fig. 38. Migration Time in Milliseconds

REALTOR-based systems will provide higher chances of application-survivability by migrating tasks in a minimized time. In case of system failure or external cyber attack, we conjecture that there will be very little amount of time for applications to flee. Our methodology, lightweight migration and a proactive resource discovery, requires a minimum time for migration. In this experiment, since we have used a cluster of workstations and our applications are the only tasks generating network traffic, that is a very small portion of the network bandwidth, the migration time of REACTIVE-based is believed as minimized. So, it is very likely that the migration time of REACTIVE-based system will be significantly longer in an environment where hosts are geographically widely dispersed and networks are shared by a large number of nodes. In that case, the absolute value of the difference in migration time will significantly be larger. Therefore it is highly likely that REACTIVE-based systems

will not effectively support application-survivability in real situations.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

In this thesis we proposed and evaluated our unique methodology for providing real-time capabilities to component-based reconfigurable distributed systems.

By using Java we not only largely eliminated the problem of platform heterogeneity but also simplified the migration procedure for application developers. Our methodology for providing real-time infrastructure consists of solutions to three design issues: the creation of real-time Java threads with appropriate assignment of scheduling parameters, the propagation of real-time properties between clients and servers, and modeling the patterns of client arrivals for an exported Java RMI server. As a proof of concept, the experiment results are very encouraging in that our methodology guarantees predictable latencies of remote method invocations, even in association with Java RMI server's migration. Also, using a dynamic scheduling based on exponentially distributed arrival model allows for getting optimal periods of Total Bandwidth Servers to minimize scheduling overhead and maximize utilization. As a middleware, our methodology provides a new and practical way of providing real-time capabilities. We believe that together with the DRTSJ, our methodology will play a role in supporting real-time capability for reconfigurable distributed systems. In the future, we plan to try to combine our work with the DRTSJ.

At the same time, lightweight migration and proactive resource discovery are key component in supporting application-survivability in distributed real-time systems as component migration needs to be done within a least amount of time for emergency cases. We have used Java Object Serialization protocol for lightweight migration. According to our analytical study, the communication overhead of our resource discovery protocol, REALTOR, is much less than those of pure PULL-based and pure

PUSH-based resource discovery approaches. Simulation studies show that under normal and heavy load conditions REALTOR remains very effective in finding available resources with a reasonably low communication overhead compared to a pure PUSH-based or pure PULL-based approach. Also, our implementation and measurements in a workstation cluster show that the methodology has much shorter migration time compared to a general request-and-reply reactive resource discovery protocol. We are planning to further investigate the effectiveness and limitations of this methodology in application-survivability.

REFERENCES

[1] A. Sahai, S. Singhal, R. Joshi, and V. Machiraju, "Automated policy-based resource construction in utility computing environments," Technical Report HPL-2003-176, Hewlett-Packard Laboratories, Palo Alto, CA, Aug. 2003.

[2] Object Management Group, "CORBA specifications," Available from http://-www.omg.org, Aug. 2004.

[3] Microsoft, "COM: Delivering on the promise of component technology," Available from http://www.microsoft.com/com, Aug. 2004.

[4] Sun Microsystems, "Enterprise JavaBeans technology," Available from http://-java.sun.com/products/ejb, Aug. 2004.

[5] ACE and TAO Development Team, "Real-time CORBA with TAO," Available from http://www.cs.wustl.edu/∼schmidt/TAO.html, Aug. 2004.

[6] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An aspect-based composition tool for real-time systems," in *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2003, pp. 58–69.

[7] Object Management Group, "Realtime CORBA joint revised submission," Available from http://www.echelon4.com/content%20files/1, Oct. 1998.

[8] D. C. Schmidt and F. Kuhns, "An overview of the real-time CORBA specification," *IEEE Computer Magazine*, vol. 33, no. 6, pp. 56–63, June 2000.

[9] J. W. S. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ: Prentice-Hall, 2000.

[10] X. Liu and S. Goddard, "Supporting dynamics QoS in Linux," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2004, pp. 246–254.

[11] Concurrent Systems Architecture Group, "Agile Objects Project," Available from http://www-csag.ucsd.edu/projects/agileO.html, Aug. 2004.

[12] K. Connelly and A. Chien, "Breaking the barriers: High performance security for high performance computing," in *Proceedings of the 2002 Workshop on New Security Paradigms*, Virginia Beach, VA, Sept. 2002, pp. 36–42.

[13] B. Choi, S. Rho, and R. Bettati, "Dynamic resource discovery for application survivability in distributed real-time systems," in *Workshop on Parallel and Distributed Real-Time Systems*, Nice, France, Apr. 2003, pp. 122–129.

[14] G. Bollella, J. Gosling, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java^{TM}*, 1st ed. Boston, MA: Addison-Wesley Publishing Company, 2000.

[15] Sun Microsystems, "Java^{TM} Remote Method Invocation Specification (RMI)," Available from http://java.sun.com/products/jdk/rmi/, Aug. 2004.

[16] P. Goyal and H. M. Vin, "Generalized guaranteed rate scheduling algorithms: A framework," *IEEE/ACM Transactions on Networking*, vol. 5, no. 4, pp. 561–571, Aug. 1997.

[17] R. DiGiorgio, "Java in embedded systems," Available from http://www.javaworld.com/javaworld/jw-09-1996/jw-09-javadev.html, Sept. 1996.

[18] S. Palu, "Real-time specification for Java (RTSJ)," Available from http://www.developer.com/java/article.php/10922_1367671_1, Aug. 2004.

[19] E. Jensen, "The distributed real-time specification for Java - an initial proposal," *Journal of Computer Systems Science and Engineering*, vol. 16, no. 2, pp. 65–70, Mar. 2001.

[20] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java^{TM} Language Specification*, 2nd ed. Boston, MA: Addison-Wesley Publishing Company, 2000.

[21] Sun Microsystems, "Java Object Serialization Protocol," Available from http:/-/java.sun.com/j2se/1.3/docs/guide/serialization/, Aug. 2004.

[22] C. Mangione, "Performance tests show Java as fast as C++," Available from h-ttp://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf_p.html, Feb. 1998.

[23] J. E. Moreira, S. P. Midkiff, and M. Gupta, "A comparison of Java, C/C++, and FORTRAN for numerical computing," *IEEE Antennas and Propagation Magazine*, vol. 40, no. 5, pp. 102–105, Oct. 1998.

[24] T. Lindholm and F. Yellin, *The Java^{TM} Virtual Machine Specification*, 2nd ed. Boston, MA: Addison-Wesley Publishing Company, 1999.

[25] TimeSys, "The RTSJ Reference Implementation (RI)," Available from http://-www.timesys.com, Aug. 2004.

[26] A. Corsaro and D. Schmidt, "The design and performance of the jRate real-time java implementation," in *International Symposium on Distributed Objects and Applications*, Irvine, CA, Oct. 2002, pp. 900–921.

[27] R. Clark, E. Jensen, A. Wellings, and D. Wells, "The distributed real-time specification for Java: A status report," Available from http://www.real-time.org/docs/esc02paper.pdf, Mar. 2002.

[28] A. Wellings, R. Clark, D. Jensen, and D. Wells, "A framework for integrating the real-time specification for Java and Java's remote method invocation," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Crystal City, VA, Apr. 2002, pp. 13–22.

[29] A. Borg and A. Wellings, "A real-time RMI framework for the RTSJ," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003, pp. 238–246.

[30] M. de Miguel, "Solutions to make Java-RMI time predictable," in *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Magdeburg, Germany, May 2001, pp. 379–386.

[31] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 1998, pp. 4–13.

[32] L. Abeni and G. Buttazzo, "QoS guarantee using probabilistic deadlines," in *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, York, UK, June 1999, pp. 242–249.

[33] Sun Microsystems, "java - the Java application launcher," Available from http-://java.sun.com/j2se/1.3/docs/tooldocs/linux/java.html, Aug. 2004.

[34] Ethereal Working Group, "Ethereal," Available from http://www.ethereal.co-m/, Aug. 2004.

[35] L. F. Wilson and W. Shen, "Experiments in load migration and dynamic load balancing in SPEEDES," in *Simulation Conference Proceedings*, Washington, DC, Dec. 1998, pp. 483–490.

[36] G. Lanfermann, G. Allen, T. Radke, and E. Seidel, "Nomadic migration: Fault tolerance in a disruptive grid environment," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002, pp. 280–281.

[37] Condor Team, University of Wisconsin at Madison, "Condor high throughput computing," Available from http://www.cs.wisc.edu/condor/, Aug. 2004.

[38] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpointing and migration of UNIX processes in the Condor distributed processing system," Technical Report 1346, Department of Computer Sciences, University of Wisconsin at Madison, Madison, WI, Apr. 1997.

[39] M. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel," in *Proceedings of the USENIX Winter Conference*, San Francisco, CA, Jan. 1992, pp. 283–290.

[40] R. Ma, C. Wang, and F. Lau, "M-JavaMPI: A Java-MPI binding with process migration support," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002, pp. 240–247.

[41] W. Zhu, C. Wang, and F. C. M. Lau, "JESSICA2: A distributed Java Virtual Machine with transparent thread migration support," in *Proceedings of the 2nd IEEE International Conference on Cluster Computing*, Chicago, IL, Sept. 2002, pp. 381–388.

[42] P. Troger and A. Polze, "Object and process migration in .NET," in *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, Guadalajara, Mexico, Jan. 2003, pp. 139–146.

[43] Q. Lv, P. Cao, E. Cohen, and K. Li, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of the 16th ACM International Conference on Supercomputing*, New York, NY, June 2002, pp. 84–95.

[44] Lime Wire LLC, "LimeWire," Available from http://www.limewire.com, Aug. 2004.

[45] S. Androutsellis-Theotokis, "A survey of peer-to-peer file sharing technologies," Available from http://www.eltrun.aueb.gr/whitepapers/p2p_2002.pdf, Aug. 2002.

[46] A. Iamnitchi, I. Foster, and D. C. Nurmi, "A peer-to-peer approach to resource discovery in grid environments," Technical Report TR-2002-06, University of Chicago, Chicago, IL, June 2002.

[47] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A distributed resource management architecture that supports advance reservations and co-allocation," in *International Workshop on Quality of Service*, London, UK, June 1999, pp. 27–36.

[48] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource reservation protocol," *IEEE Networks Magazine*, vol. 31, no. 9, pp. 8–18, Sept. 1993.

[49] I. Foster, A. Roy, and V. Sander, "A quality of service architecture that combines resource reservation and application adaptation," in *Proceedings of the 8th International Workshop on Quality of Service*, Pittsburgh, PA, June 2000, pp. 181–188.

[50] A. Iamnitchi and I. Foster, "On fully decentralized resource discovery in grid

environments," in *2nd International Workshop on Grid Computing*, Denver, CO, Nov. 2001, pp. 51–62.

[51] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw, "Resource management in Legion," *Journal of Future Generation Computer Systems*, vol. 15, pp. 583–594, Oct. 1999.

[52] M. Livny, J. Basney, Raman, and T. Tannenbaum, "Mechanisms for high throughput computing," *SPEEDUP Journal*, vol. 11, no. 1, pp. 36–40, June 1997.

[53] J. Basney, M. Livny, and P. Mazzanti, "Utilizing widely distributed computational resources efficiently with execution domains," *Journal of Computer Physics Communications*, vol. 140, pp. 246–252, Oct. 2001.

[54] J. Basney and M. Livny, "Improving goodput by co-scheduling CPU and network capacity," *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 220–230, Fall 1999.

[55] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management for high throughput computing," in *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998, pp. 140–146.

[56] W. Zhao and K. Ramamritham, "Distributed scheduling using bidding and focused addressing," in *Proceedings of the IEEE Real-Time Systems Symposium*, San Diego, CA, Dec. 1985, pp. 103–111.

[57] K. Ramamritham, J. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers*,

vol. 38, no. 8, pp. 1110–1123, Aug. 1989.

[58] T. Cheng, J. Chung, and K. Lin, "Dynamic load balancing algorithms in loosely-coupled real-time systems," in *Proceedings of the 16th International Computer Software and Application Conference*, Chicago, IL, Sept. 1992, pp. 143–148.

[59] L. Kleinrock, *Queueing Systems, Volume I: Theory*, 1st ed. New York, NY: John Wiley & Sons, 1975.

VITA

Name:	Sangig Rho
Permanent Address:	565 Shindaebang Woosung APT 16-805 Seoul, 156-011, Republic of Korea
Educational Background:	Bachelor of Science, Electronic Engineering Yonsei University, 1989
	Master of Science, Electronic Engineering Yonsei University, 1991
	Doctor of Philosophy, Computer Engineering Texas A&M University, 2004