While You're Away: A System for Load-Balancing and Resource Sharing based on Mobile Agents

Niranjan Suri, Paul T. Groth, and Jeffrey M. Bradshaw
Institute for Human & Machine Cognition, University of West Florida
{nsuri,pgroth,jbradshaw}@ai.uwf.edu

Abstract

While You're Away (WYA) is a distributed system that aggregates the computational power of individual computer systems. WYA introduces the notion of Roaming Computations – *Java-based programs that move around the network utilizing the resources of idle workstations. WYA provides architectural independence and addresses issues of convenience, security, and incentive for owners of workstations.*

WYA is based on the NOMADS mobile agent system, which uses the Aroma Virtual Machine (VM) to provide strong mobility, resource control, and resource accounting. WYA currently runs on Win32 and UNIX workstations but is being extended to work on other computational devices such as television set-top boxes, video game consoles, and Internet appliances.

1. Introduction

One of the goals of distributed systems is to aggregate the computational power of several individual computer systems. This aggregate computational power can match or exceed the processing power of supercomputers while at the same time lowering the cost/performance ratio significantly. Systems may be classified into two kinds: those that use dedicated groups of systems (such as computational clusters) and those that use idle workstations.

While You're Away (WYA) is a distributed system that leverages idle workstations to perform computations. WYA introduces the notion of roaming computations, which are extensions of mobile agents that move to and out of workstations as the workstations become free and busy. One advantage of using idle workstations is that existing systems can be reused without incurring the cost of dedicated clusters. However, WYA trivially extends to support computational clusters by treating them as workstations.

WYA relies on the NOMADS mobile agent system [1], which provides strong mobility for Java-based agents. Unlike weak mobility, which requires that a mobile agent restart execution after a move, strong mobility moves an agent's execution state with the agent. NOMADS also supports forced mobility, where external events can forcibly move agents from one system to another. Since the execution state of agents is also moved, agents resume execution on the new system without any knowledge of the forced migration. Secondly, NOMADS provides dynamic resource control and resource accounting mechanisms. Resource control allows limits to be placed on the resources consumed by any agent and can be used to protect hosts against malicious or buggy agents as well as to prioritize the execution of agents. Resource accounting allows the system to measure and keep track of the resources consumed by agents on a host.

Important features of WYA include:

Architecture Independence: Roaming computations can be moved across systems of different architectures. Therefore, it is no longer necessary to have identical workstations for load-balancing.

Workstation Availability: When an end-user wishes to use his or her personal workstation, any roaming computations on that workstation are immediately moved out using the forced migration capabilities of NOMADS. This is critical to make sure that users do not view the roaming computations as bothersome.

Workstation Security: Using the resource control mechanisms of NOMADS, WYA guarantees that the resources of user workstations are not abused.

User Incentive: WYA uses the resource accounting mechanisms to keep track of the resources used by roaming computations. As a user's workstation gets used, the user accumulates points that could conceivably be used for a variety of purposes.

0-7695-1010-8/01 $10.00 © 2001 IEEE

The rest of this paper is organized as follows. Section two briefly describes the NOMADS system upon which WYA is based. Section three presents an overview of the design on WYA. Section four the WYA programming API and the implementation of the server. Section five discusses transparent resource redirection. Finally, section six concludes by discussing future work.

2. Overview of NOMADS

The NOMADS mobile agent system [1] was developed to overcome limitations of current Java-based systems. NOMADS offers two key capabilities: strong mobility (i.e., the ability to capture and move the execution state of the agent along with the agent at the demand of the agent, the system, or a user) and strong security (i.e., the ability to securely execute an agent while accounting for and controlling the agent's access to system resources).

NOMADS introduces two forms of strong mobility – synchronous and asynchronous. In the first case, an agent is free to request a move operation at any point in its execution. NOMADS will capture the execution state of the agent, move the state to the destination system, and continue the execution of the agent. The agent resumes execution at the very next statement after the request to move. NOMADS refer to this form of strong mobility as *anytime* mobility. It should be noted that an agent that uses anytime mobility could always be rewritten to work with weak mobility (i.e., systems that do not move execution state, but restart the execution of the agent on the destination system). However, the burden would then lie on the agent developer to program make the agent always maintain its own execution state. Therefore, anytime mobility greatly simplifies the task of writing an agent.

The second form of strong mobility is referred to as *forced* mobility. In this case, an agent may be moved from one system to another due to an asynchronous external event that may be generated by the agent system or a user or administrator. The agent's execution state is transparently migrated to the destination platform where it resumes execution. Note that the migration can be completely transparent to the agent, which simply continues executing the next bytecode instruction on the new platform. An agent can remain completely oblivious to the fact that it was moved to a new location (much like a process is oblivious to the fact that an operating system may be constantly saving and restoring the state of the process during multitasking). While it may be possible to simulate forced mobility in agent systems that do not move execution state, doing so requires constant polling which results in an inefficient system.

The second key capability of NOMADS is resource accounting and control. NOMADS allows the resources consumed by any agent to be measured and queried. Moreover, various limits may be placed on the resources that can be consumed by an agent. Limits include both rate limits (e.g., percentage of CPU usage, disk and network read/write rates) and quantity limits (e.g., disk spaced used). These resource limits may be dynamically adjusted at a fine level of granularity and are enforced transparently to the agent running within NOMADS.

The core component of NOMADS is the Aroma Virtual Machine (VM) [2]. Aroma is designed to support state capture, resource control, and resource accounting. Unlike other systems that use a modified Sun Java VM, the clean-room nature of the implementation of Aroma allows the VM (and consequently NOMADS and WYA) to be distributed without any licensing constraints.

3. Design of WYA

The current design of WYA is optimized for a Local Area Network (LAN) environment. The system includes a coordinator (called the WYA Server) and one or more workstations. Users submit roaming computations to the coordinator, which then distributes them to idle workstations. When workstations become busy, the computations are sent back to the coordinator. The coordinator also keeps track of the resource consumption matrix, which determines the points earned by end-user workstations and points consumed by jobs.

WYA uses two types of workstations. Idle workstations are systems where a user is logged in but is currently inactive. Free workstations are systems that do not have a user logged into the console. Workstations that have a console user rely on a special screen saver to detect idle time. The screen saver also shows status information such as the computations currently running, resource consumption meters, and total points accumulated. For workstations that are free, WYA uses the CPU load to decide whether to send computations to the system.

Figure 1 shows the steps that normally occur in WYA. The first step involves a user using the WYA launcher to submit a job to the coordinator. The coordinator maintains a job queue (which is currently FIFO). When a workstation becomes idle and the WYA screen saver is invoked, the screen saver notifies the coordinator, which then sends a computation to the workstation. If the user wishes to resume using the workstation, the screen saver,

before disappearing, will cause the computations to move back to the coordinator. Note that a computation may move back and forth between idle workstations and the coordinator many times before completing. When the computation finishes, the computation returns to the original user's workstation to report any results.

Also note that computations may move to free workstations. In the case of free workstations, WYA detects CPU idle time to decide to move computations to a workstation.

4. Implementation

This section describes the API used by end-users when developing roaming computations and also the implementation of the WYA server.

4.1 Roaming Computation API

Figure 2 shows a typical roaming computation. Users must extend the class RoamingComputation and define three methods: init(), compute(), and report(). WYA guarantees that the init() function will be executed on the user's workstation. Hence, the init() function should be used by computations to perform such tasks as reading data files or otherwise accessing resources that are only available on the user's workstation. After execution of init() is completed, the computation is queued on the WYA server until a workstation is available. The computation is executed on any available workstation until the compute() function returns. Then, the computation is moved back to the user's workstation and the final report() method is executed. Computations can write data files or carry out other tasks to convey the results back to the user in the report() method.

Note that while the compute() method is executing, the state of the computation may be captured, moved to the WYA server, moved back to a (possibly different) workstation, and restarted. This process could take place many times before the compute() method completes and the computation is returned to the original user's workstation.

4.2 WYA Server

The WYA server maintains a queue of roaming computations that are ready to execute. If a workstation becomes available, the server is notified either by the screen saver on the workstation or by the NOMADS service (if no user is logged in and the CPU usage is below a pre-configured threshold). The server then dequeues the first roaming computation and sends the computation to the idle workstation. When jobs are sent back to the server, they are placed at the end of the queue (a simple round-robin algorithm).

After every iteration of a computation going to a workstation and returning, the server queries the workstation about the resources consumed by the computation during the last iteration. The server uses this information to build up a matrix of resources consumed by a computation (for billing purposes) and resources provided by a workstation (for compensation purposes).

5. Transparent Resource Access

Since the compute() method of the roaming computation may execute on many arbitrary systems, the

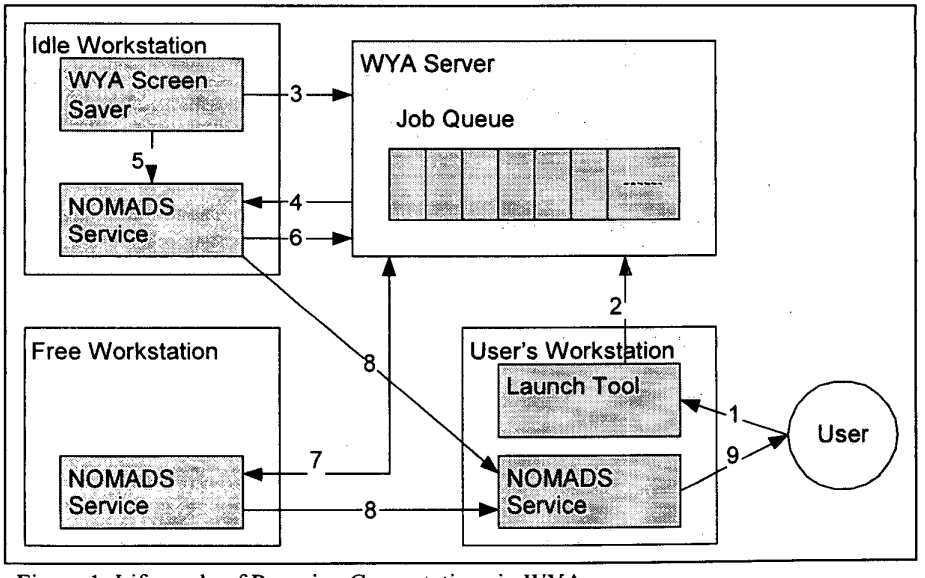

Figure 1: Life-cycle of Roaming Computations in WYA

compute() method cannot access any resources on the user's workstation. Moreover, since the computation may be moved from one system to another at any point in the execution of the compute() method, the method cannot even depend on accessing resources on the currently running host.

However, WYA does allow support for network endpoints to be used within the compute() method, In particular, WYA relies on the mockets (mobile sockets) [3] mechanism in NOMADS to allow roaming computations to create and use a mocket (which is similar to a socket) even though the computation may be moving from workstation to workstation. Therefore, roaming computations can read and write over the network without having to reestablish any connections even though they are being moved between systems.

6. Future Work

One key requirement that has not been addressed is the notion of sub-computations. We will to provide a mechanism for a computation to spawn sub-computations and to wait for them to complete if necessary. These computations will also be allowed to roam across workstations as needed.

We are currently working on a mechanism to transparently access files regardless of the location and migration of the roaming computation. Coupled with the existing Mockets service, transparent file redirection would allow roaming computations to access both disk and network resources.

We are also interested in expanding the WYA system to work with other computational devices such as video game consoles, television set-top boxes, and other Internet appliances.

Finally, we plan to develop a Just-In-Time (JIT) compiler for the Aroma VM. A JIT is key to providing acceptable performance for roaming computations within a VM environment.

7. References

[1] Suri, N., Bradshaw, J.M., Breedy, M.R., Groth, P.T., Hill, G.A., and Jeffers, R. Strong Mobility and Fine-Grained Resource Control in NOMADS. Proceedings of the 2nd International Symposium on Agents Systems and Applications and the 4th International Symposium on Mobile Agents (ASA/MA 2000). Springer-Verlag.

[2] Suri, N. Bradshaw, J.M., Breedy, M.R., Ford, K.M., Groth, P.T., Hill, G.A., Saavedra, R. State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine. Sumbitted for Publication.

[3] Mitrovich, T.S., Ford, K.M., and Suri, N. Transparent Redirection of Network Sockets. OOPSLA Workshop on Experiences with Autonomous Mobile Objects and Agent-based Systems. On-Line Reference - http://www-users.cs.umn.edu/~tripathi/Workshop.html.

```
import edu.uwf.nomads.wya.RoamingComputation;

public class MyComputation extends RoamingComputation
{
    public void init (String args[])
    {
        // Perform any initialization required here
    }

    public void compute()
    {
        // Actual computations go here
    }

    public void reportResults()
    {
        // Report results back to the user here
    }
}
```

Figure 2: Sample Roaming Computation Demonstrating the API