A Market-based Protocol with Leasing Support for Globally Distributed Computing

George Kakarontzas* and Spyros Lalis**

*Computer Engineering, Telecommunications and Networks Department
University of Thessaly
Volos, Greece
{gkakar,lalis}@inf.uth.gr

**Institute of Computer Science, Foundation for Research and Technology - Hellas
Heraklion, Greece
lalis@ics.forth.gr

Abstract. We have developed JaWS, a Java-based web computing system, which enables users to effortlessly export their machines in a global market of processing capacity to host remote computations [6]. Leases are used to promote dynamic task placement as well as fair compensation for the host providers. In this paper, we present an updated protocol used by hosts and applications to interact with the JaWS market, through which resource allocation takes place. Although this work is carried out in the context of JaWS, it can also be applied to other market-based resource allocation frameworks.

1. Introduction

The large growth of the Internet, both in the number of personal connected devices as well as in bandwidth, constitutes large scale distribution of computations over the Internet very appealing. However, this poses several problems due to the widely heterogeneous, error prone and uncontrolled environment at hand.

One of the main characteristics of globally distributed computing is that hosts, most likely being personal computers, are not owned by the agents that employ them to do a computation. This is in contrast to cluster computing environments where users have access (i.e. an account) to the corresponding machines. Moreover, host machines are typically provided on a voluntary basis and there is no ‘professional’ administrator responsible for installing and maintaining the corresponding runtime system. Thus awkward and time-consuming administrative tasks must be eliminated, allowing people to casually make their computing devices a host of the system.

Perhaps more important, one should offer incentives for people to become host providers. A credit-based approach is an important step towards achieving this goal (even if credit is not translated into real money). It is thus desirable to award providers depending on the amount of work performed on their machines. The value of a host could even be determined according to the laws of supply vs demand, with hosts being assigned to the applications that make the best offer. This leads to the concept of leasing, i.e. periodically negotiated host usage –as opposed to using a host until a computation finishes.

In this paper we present a protocol with built-in support for leasing, dynamic negotiation, and computation migration. The described protocol is an updated version of the protocol used in JaWS [6], currently under development. The rest of the paper is organized as follows. Section 2 gives an overview of the market-based operation of JaWS. Section 3 discusses how support for Task groups reflects on the market mechanism while Section 4 describes issues that arise due to the introduction of leasing. Section 5 introduces the market protocol and Section 6 gives indicative scenarios. Section 7 gives an overview of related work. Finally, in Section 8, we give future research directions.

2. Market-based Allocation in JaWS

JaWS, is a system for globally distributed computing using off-the-shelf Java technology.
Machines connected to the Internet can be registered with the system as hosts via a web page. Client applications can be submitted to JaWS from any machine connected to the Internet with Java installed on it. The main entities of JaWS are shown in Figure 1. Core system entities are depicted in gray, while application-dependent entities are depicted in white.

Client applications are submitted to JaWS in the form of a Scheduler and several Task objects. The Scheduler interacts with the system in order to allocate hosts to be used for Task execution.

Hosts interact with JaWS through a Java Applet, the HostAgent, which is installed on the local Java Virtual Machine of the host machine as soon as the user registers with the system. The Host Agent receives Tasks to be executed on behalf of client applications.

Allocation of hosts to applications occurs through the JaWS Market. From an interaction point of view, the Market serves as a meeting place for ‘trading’ processing capacity, with hosts acting as providers and applications acting as consumers. Allocation is performed based on orders submitted to the Market by hosts (i.e. Host Agents) and applications (i.e. Schedulers). Orders include a machine profile, consisting of performance benchmark figures and the host abort ratio, a ‘reliability’ indicator computed as the ratio of computations forcefully terminated versus the computations initiated on a host. Orders also include a usage duration, indicating the amount of time a Task can occupy a host before renegotiating the contract of use.

A credit based [1] mechanism is used for charging. Credit can be translated into anything that makes sense in the context where the system is deployed. Within a non-profit environment, it may be used to represent time units to facilitate quotas, to introduce priorities, or to give credit to host donors. Service-oriented organizations could charge clients for using hosts by converting credit to actual currency. A double auction mechanism [5] is employed to match the orders issued by hosts and applications. This essentially guarantees a fair settlement if both hosts and applications submit their true valuations, thereby eliminating the need for a series of negotiations to set the price of use.

For each match, a lease is produced, which is a contract between a host and an application containing their respective orders. A lease entitles the client to utilize the host for a certain amount of time (which corresponds to the expiration time of the respective orders) and for a certain amount of credit. This amount is transferred from the application to the host when the lease expires.

3. Support for Task Groups

In JaWS, we plan on providing extended support for a number of programming paradigms in the form of paradigm-specific schedulers. For instance, besides trivially parallel applications, JaWS should support Tasks that communicate with each other. In the context of this discussion, we show how the notion of a Task group is explicitly supported by the market mechanism. The UML [11] diagram in Figure 2 depicts the relationship between a Scheduler, its Task objects, and the orders submitted to the Market.

The Scheduler may allocate hosts by submitting to the Market Single Orders or Group Orders. A Single Order is placed to allocate a single host for a single Task. A Group Order is placed in order to allocate several hosts for an equal number of Tasks. By using a Group Order, the Scheduler does not have to place (and keep track of) different orders for a group of Tasks that are to be executed simultaneously.

A typical case where this is useful is when several Tasks depend on each other and where...
having only a part of them active is not particularly meaningful. Through a Group Order, the Market can then be instructed to produce a lease for an entire group ‘atomically’, i.e. either a lease for each Task of the group or none at all. Another benefit of this grouping concept is that the Market can attempt to allocate neighboring hosts (e.g. part of the same LAN) to accommodate a Task group. Hence, a Task group has a good chance of being ‘naturally’ placed on a homogeneous computer environment with acceptable communication delays between the various hosts.

Task groups can also be dynamic. In other words, a Task could start as part of a group but then evolve into an isolated Task that does not interact with the rest of the group. From the Market point of view, this requires splitting a Group Order into a new Group Order (for the rest of the group’s Tasks) and a Single Order (for the isolated Task). Conversely, if desired, an isolated Task could become a member of a group. In this case, the corresponding Single Order should be integrated into the Group Order. The Scheduler can perform these changes in a straightforward way.

4. Support for Leases

Leasing is clearly desirable for host providers. It guarantees that the compensation (in terms of credits) for making their processing capacity available will be determined according to supply vs. demand. However, several issues must be addressed in order for a system to support leasing in a way that is efficient and simplifies application programming.

Since host usage is subject to negotiation each time the lease expires, applications cannot occupy hosts arbitrarily long. In order for an application to use a host for a period of time that is greater than the lease duration, it must be possible for the application to renew the lease. This may have to be done several consecutive times until the computation finishes. Moreover, this must be achieved in a way that avoids unnecessary suspension and restart of the Task executing on the host, in case the application manages to renew the lease.

The success of a lease renewal attempt cannot be guaranteed though. In a competitive situation, a host occupied by an application may have to be re-allocated to another application that makes a better offer for this resource. This interruption of service must be communicated to the application in a twofold way.

The application Task that executes on the host must be notified in order to checkpoint its state before it is suspended. This information will be used at a later point in time to restart the Task when a new host is eventually allocated to the application. This ‘soft’ way of stopping Task avoids various inconsistencies that may occur in a distributed computation if Tasks are killed forcefully (notably, in the current versions of Java, it is impossible to really kill a thread).

The application Scheduler must also be notified of such an event in order to take corrective actions and to handle the checkpoint state of the Task that was suspended.

5. A Protocol Supporting Dynamic Lease Renewal and Task Migration

Having considered these issues, we are developing a revised market protocol, which we intend to use in the next version of JaWS. The proposed protocol is described as a set of method calls (in Java-like syntax) between the various entities of the system. The most relevant calls are briefly discussed in the following.

\[\text{HostAgent} \rightarrow \text{Task}\]
- \(\text{void Start(TData)}\): Instructs the Task to start execution using the supplied initialization data.
- \(\text{TData Suspend()}\): Instructs the Task to suspend execution, checkpoint and return its state.
- \(\text{void Abort()}\): Instructs the Task to abort execution without saving its state.

\[\text{HostAgent} \rightarrow \text{Market}\]
- \(\text{void Available(Order)}\): Inserts an order in the Market, indicating that the machine can be used as a host for Task execution. The order will be considered in future matching attempts. If the HostAgent has already inserted an order, the previous order is overridden by this order.
- \(\text{void Unavailable()}\): Cancels the order that has been previously inserted into the Market. The respective order is removed from the Market and will not be considered in future matching attempts. However, if a Task already resides on the HostAgent, it continues execution.

\[\text{Market} \rightarrow \text{HostAgent}\]
- \(\text{void Start(TCode,TData)}\): Instructs the HostAgent to start execution of a Task;
this leads to the creation of a Task object and invocation of the corresponding method.

- **TData Suspend():** Instructs the HostAgent to suspend the currently executing Task and to return its checkpoint state; this leads to the invocation of the corresponding Task method.
- **void Abort():** Instructs the HostAgent to abort the currently executing Task, if any; this leads to the invocation of the corresponding Task method.

**Scheduler -> Market**
- **void Alloc(Orders):** Inserts a list of orders in the Market, indicating the machines to be allocated in order to host the Tasks of this computation. These orders will be considered in future matching attempts and are added to any previously inserted orders.
- **void Cancel(Orders):** Cancels a list of orders that have been previously inserted into the Market. The respective orders are removed from the Market and will not be considered in future matching attempts. Leases obtained through these orders remain valid and the corresponding Tasks continue execution. Any other previously inserted orders remain valid.
- **void Start(Lease,TCode,TData):** Instructs the Market to start execution of a Task on the host for which the lease was obtained. The respective order remains in the Market.
- **TData Suspend(Lease):** Instructs the Market to suspend execution of a Task on the host for which the lease was obtained and return the checkpoint state. The respective order remains in the Market.
- **void Abort(Lease):** Instructs the Market to abort execution of a Task on the host for which the lease was obtained. The respective order remains in the Market.

**Market -> Scheduler**
- **void Confirm(Leases):** Informs the Scheduler of leases corresponding to matched orders in order for them to be confirmed. A lease is confirmed by starting a Task on the corresponding host. The respective order remains in the Market.
- **void Expired(Lease,TData):** Informs the Scheduler of a lease expiration and failure of renewal. The checkpoint state of the Task residing on the corresponding host is also provided, should it be desirable to restart the Task on another host at a later point in time. The respective order remains in the Market.

**Market -> Timer**
- **void SetTimer(Event,millis):** Registers a timer event to be generated after a certain amount of time elapses.
- **void ResetTimer(Event):** Removes a timer event that has been inserted previously.

**Timer -> Market**
- **void Event(Event):** Informs the Market of a timer event. The respective event is removed from the timer.

It is important to notice that once an order is inserted in the Market, it remains in the Market unless explicitly removed. Furthermore, the Market automatically attempts to renew leases that expire. The HostAgent and Scheduler are notified only in two cases: when a lease is first generated (in order to start Task execution) and when the lease expires and the renewal attempt fails (in order to handle the checkpoint state of the Task). Also, since lease expiration events are handled exclusively through the Market, race conditions that can easily occur in an Internet-based distributed system are avoided.

### 6. Scenarios of Operation

In this section we show a few characteristic scenarios of how this protocol works. For each scenario we describe the objects and the method calls involved, illustrating the whole execution sequence via a corresponding message diagram.

**Scenario A – Continuous operation**

In this scenario (see figure 3) host H1 inserts an order in the Market. The Market attempts matching but since there is currently no demand for hosts, no lease is produced. At a later point in time, Scheduler S1 inserts an order. The Market attempts matching again and this time a lease is produced for host H1 and Scheduler S1. This lease is communicated to S1. S1 confirms the lease by requesting the Market to start Task T1 on H1, which in turn forwards this request to H1. After successful initiation of T1 on H1, the Market registers a timer event, for the expiration
of the lease. When the timer expires, the Market attempts matching and (in this scenario) produces a new lease H1 and S1. Since the new lease is a renewal of the old lease, H1 and S1 are not bothered. The Market registers a new timer event to be notified when the new lease expires.

**Scenario B – Interrupted operation**

In this scenario (figure 4) we bring into the picture an additional Scheduler and an additional host. After Task T1 from scheduler S1 is started on H1 (until then this scenario is identical to the previous one), a new Scheduler, S2, places an order in the market. This triggers matching which does not produce a lease because H1 has already been assigned to S1 and the corresponding lease has not yet expired.

After the expiration of the H1-S1 lease, however, the Market attempts matching and at this point H1 is allocated to S2 (assuming that S2 made a better offer than S1). Hence, the Market will suspend Task T1 on H1. It will then ask S2 to confirm the lease, and as a consequence S2 will request for task T2 to be started on H1. The Market forwards this request to H1, and once it is executed successfully, it registers a timer event for the expiration of the H1-S2 lease. As a last action, the Market informs S1 of the fact that the S1-H1 lease was not renewed and that T1 has been suspended. The checkpoint of T1 is handed over to S1.

At a later point in time, while S2 holds H1, a new host, H2, is made available to the system. The Market attempts matching and produces a lease for S1 and H2 that is communicated to S1. In turn, S1 confirms the lease by restarting Task T1 on H2, using the checkpoint that was created as a result of suspending T1. Finally the Market registers a timer event for the expiration of the S1-H2 lease.

From the above scenarios it becomes evident that this protocol is intended for coarse-grained parallelism and for extended periods of leasing. Else the context switch overhead due to task suspensions and migrations caused by frequent host reallocation (during periods of contention) would make the system inefficient.

**7. Related Work**

In [3] a market is described where hosts receive cyber money or other resources for providing their services to clients. The need for a motive so that hosts will provide their computational resources to agents is also recognized in [2]. Price advertisement is regarded as a simple mechanism for agent coordination since high prices will discourage agents from using certain hosts and even make them defer their execution for a later time. This view is supported in [3], where it is suggested that market mechanisms can be extended to achieve scalability and load balancing.
A similar project to JaWS is the Popcorn project [10]. Popcorn also provides an incentive for the sellers of computer power who are credited for the usage of their machines with popcoins an electronic monetary unit. The language used is Java and the researchers of Popcorn developed an API available for download from the Internet (source code as well) [9]. This API can be used for the development of Popcorn distributed applications, which are composed from “Computlets”, the Popcorn analog to JaWS tasks.

Javelin++ [7] is an evolution of Javelin [4] where the efforts of the developers where mainly in the scalability and performance areas. There is a shift from Java Applets to a Javelin daemon, running as a screen-saver on the host when it is idle, to take advantage of advanced security options offered in JDK1.2. Users can download and install the screen-saver on their machines, instead of pointing to a web page. Javelin++ provides better scalability than its predecessor with the introduction of many brokers. There is a master broker and there are a number of secondary brokers assigned to it. The scalability of work is achieved through the technique of work stealing. Hosts steal work from other hosts when they become idle. Fault tolerance is achieved through distributed eager scheduling where tasks are reassigned to hosts when there are results still outstanding.

A commercial application of Internet computing is Parabon’s Frontier[8]. Frontier is the product name of the server that does the distribution of the tasks to clients. Another product that comes with Frontier is Pioneer, which is a Java desktop application that executes the tasks on the client side. Frontier also comes with a development suite that helps clients to build Java applications. This development suite includes a Java API that is used for programming Frontier applications and also includes libraries of templates that can be used in programming distributed applications.

8. Future Directions

A system that supports leasing can be a platform for future experimentation and research. For instance, we plan to examine the conditions under which a leasing approach is better than a non-interrupting approach, where a task assigned to a host will migrate to another host, only in the presence of network failure.

We would also like to look at a number of different design aspects of the system in the near future. One important goal is to classify the types of applications that the system supports and provide clear guidance and an extensible framework to develop such applications. This will simplify programming and provide a basic source of information to support applications more effectively on the server. Moreover, we would like to focus on the concept of task groups and support various different types with support for different communication patterns.

9. References

8. Parabon’s Frontier: www.parabon.com