A General Resource Reservation Framework for Scientific Computing*

Ravi Ramamoorthi, Adam Rifkin, Boris Dimitrov, and K. Mani Chandy

California Institute of Technology

Abstract. We describe three contributions for distributed resource allocation in scientific applications. First, we present an *abstract model* in which different resources are represented as tokens of different colors; processes acquire resources by acquiring these tokens. Second, we present *distributed scheduling algorithms* that allow multiple resource managers to determine custom policies to control allocation of the tokens representing their particular resources. These algorithms allow multiple resource managers, each with its own resource management policy, to collaborate in providing resources for the whole system. Third, we present an implementation of a distributed resource scheduling algorithm framework using our abstract model. This implementation uses Infospheres, which are Internet communication packages written in Java, and shows the benefits of distributing the task of resource allocation to multiple resource managers.

1 Introduction

A user often needs access to several distributed heterogeneous resources. For instance, a scientist may conduct a distributed experiment [3] requiring a supercomputer, a visualization unit, and a special high quality printer all in different locations. All three resources are essential to the experiment so the scientist needs to *synchronously* lock and use all three *distributed resources* for the same time period to complete the computing task. The distributed heterogeneous resources together form a *networked virtual supercomputer* or *metacomputer* [1]. The scientist also wants resources to be scheduled automatically as a service of the appropriate software, with or without the inclusion of specific supplemental information such as the times the user is available to perform the experiment.

Traditional metacomputing resource allocation [6, 9] uses a central authority for scheduling, usually for efficiency. For example, the IBM SP2 uses a scheduling algorithm [8] that reduces the wait time of jobs requiring only a few nodes, if these can be scheduled without delaying more computationally intensive jobs.

* This work was supported in part under the Caltech Infospheres Project, by the Air Force Office of Scientific Research under grant AFOSR F49620-94-1-0244, by the CISE directorate of the NSF under Problem Solving Environments grant CCR-9527130, and by the NSF Center for Research on Parallel Computation under cooperative agreement CCR-9120008. We thank Doug Lea for his helpful comments.

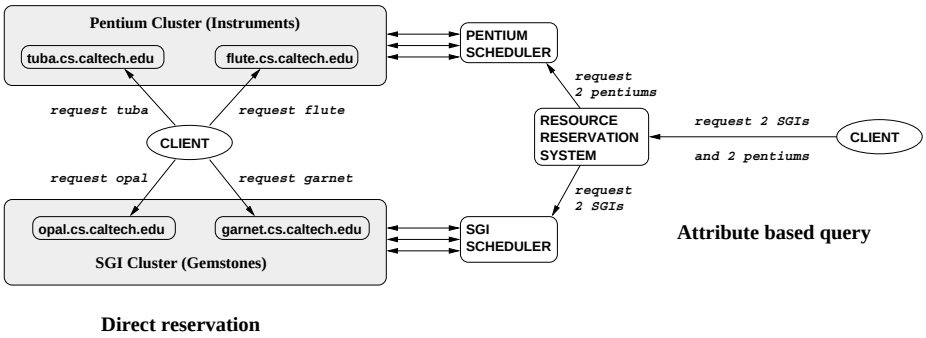


Fig. 1. Two models are given for resource reservation. On the left, the client simply asks for specific machines. On the right is a more advanced request, in which the client asks the Resource Reservation System (RRS) for two SGIs and two Pentiums. The RRS connects to separate resource managers that schedule time on the two clusters (using, for example, our calendar-based algorithm).

By contrast, consider the computational needs of users requiring resources managed by different groups in different places. Scheduling is more complicated because it is impractical for individual sites to "know" global information that would help them to do more efficient scheduling [6].

The owner of a set of resources may have resource management policies that are different from those of owners of other resource sets. Our challenge is two-fold: (i) to establish methods of cooperation so that the collection of owners offers system-wide resources to users, and (ii) to make the algorithms scalable so that new resource providers can enter the common resource pool quickly and semi-autonomously.

An infrastructure for reserving resources in a distributed system is required by many applications. Our research deals with designs and implementations of distributed resource management schemes that coordinate different policies for different sets of resources. Though this paper addresses resources used in meta-computing, our research deals with resources in many distributed applications.

A convenient abstraction for such applications represents each indivisible resource by an indivisible *token* of some color [4]; different types of resources have different colors. For instance, a node of an IBM SP2 can be represented as a token of the IBM SP color. Likewise, a room in a hotel can be represented by a token of the hotel color.

Our model deals with time explicitly. So, a reservation can be made for 64 nodes of an IBM SP2 for 10 contiguous hours, or a hotel for seven nights.

The centralized IBM SP2 scheduling algorithm relies on knowledge of how many nodes each process needs to "promote" less computationally-intensive tasks as necessary. On the other hand, as illustrated in Figure 1, if each node in a supercomputer were to be scheduled independently in a distributed way, efficient scheduling would become much more difficult. As metacomputing applications use distributed heterogenous systems, they will need algorithms for efficient dis-

tributed resource scheduling. In addition, negotiation protocols might need to leverage the notion of resources as economic currency, perhaps using electronic commerce protocols.

This paper presents a general framework for heterogeneous resource reservation. Within this framework, we present a simple Java implementation using Infospheres [2]. Specific contributions include: (i) an abstraction for distributed resource management problems that fits many, but not all, applications; (ii) a distributed implementation that coordinates multiple resource managers, each with its own policy; and (iii) efficient processing of user preferences by sending Java applets to resource managers to perform resource scheduling.

In Section 2, we discuss some simple attempts at distributed resource allocation algorithms, describe how they fail, and introduce *calendars*, which are useful for efficient resource allocation. In Section 3, we describe how the calendar metaphor builds on our resources-as-tokens metaphor. A simple application to safe metacomputer scheduling across distributed resource managers is presented in Section 4, after which we discuss efficient scheduling when resources are specified by attribute. We conclude with some observations in Section 5.

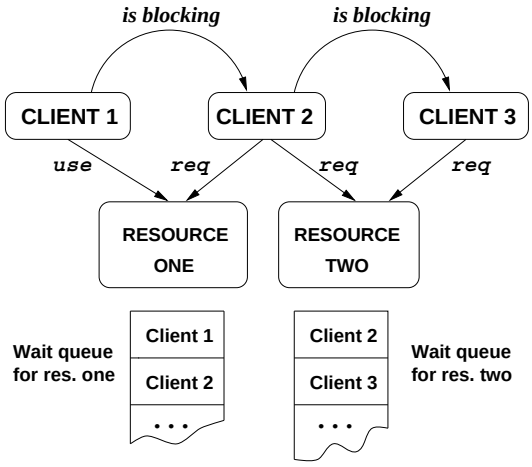
2 Distributed Resource Reservation Algorithms

The problem of distributed resource reservation has several simple solutions, including local clocks and central server, that are correct but may be inefficient. We discuss how calendars provide a more scalable solution.

One approach to resource reservation is to try to lock all of the resources the application wants. If an application is unable to lock a resource, it enters a queue waiting for it based on the priority of a logical *local clock* timestamp [4]. If an application with lower priority has the resource but is not yet using it, that application must relinquish the resource (or token), deferring to the higher-priority application. This method is robust and fairly scalable, but can be inefficient. For example, as illustrated in Figure 2, client 1 can be using resource 1, while client 2 is waiting to use it. Client 2 has locked resource 2 and is not using it, but still prevents client 3 from using 2 (which 3 could use since it requires no other resources to run its task).

As discussed in Section 1, we could improve efficiency by using a *central scheduling* algorithm, as used by the IBM SP2. However, this is clearly impractical from a scalability standpoint. Our goal is to recover some or all of the efficiency of a worldwide central server while maintaining the scalability features of distributed resource-management servers.

Calendars allow a nice tradeoff between scalability of resource managers and efficient utilization of resources. Allowing an application to "make appointments" in a calendar for resource reservation, *a resource cannot be blocked from use while sitting idle*. If a resource is unused, no application has an appointment for it at that time. Thus, efficient resource allocation is possible without global information. This calendar model is easily extensible to general resource reservation.



Inefficient local clocks solution.

Fig. 2. Local clocks can fail to use available resources. Client 1 holds resource 1, and client 2 is next in line for both resources 1 and 2. Because client 2 is blocking it, client 3 must wait for client 2 to finish (2 has higher priority), and hence client 1 to finish, even though client 3 does not even use resource 1.

3 Our Model

Individual resources use a calendar metaphor for arranging their schedule; the basic calendar functionality our implementation provides includes the concepts of time slots and access lists.

A *time slot* consists of a time interval with a particular time unit grain. Every time slot can be in one of three states: *locked*, *held* or *available*. A *Locked* slot appears when a client commits to using a resource during that slot; as a result, locked slots can be read but not written. *Held* slots are slots that a particular client is considering locking, but has not yet committed to locking. Only that client can write to these slots, thereby locking them; they are read-only for other clients. However, unlike a locked slot, a held slot can be released, reverting to *available*. Available slots may be read or written and converted to *held* or *locked* status. Slots correspond to the tokens discussed in Section 1; so, resource reservation is tantamount to collecting the proper tokens.

Each slot has an associated *access list* that keeps track of which processes can obtain a lock on that slot. For instance, a resource manager may provide access to an authorized user from the Center for Research on Parallel Computation, but not grant access to anyone else. Thus, some slots may be available to only one set of users, while others can be available to other sets of users. This approach differs from traditional "whiteboard scheduling" models.

The *reservation* of a set of resources is determined when all of the resource managers (or servers) agree to lock the slots that correspond to the same time.

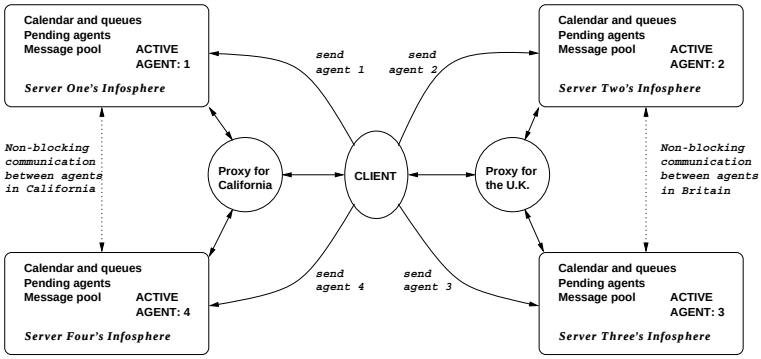


Fig. 3. Our model for scheduling a meeting starts when a client sends agents to the various resources it desires. Agents communicate with the client and with resources. For efficiency, groups of nearby agents can coordinate to avoid excessive message-passing to clients, who may be geographically distant. The resource managers or servers can also send agents to clients to request back the slots that the clients hold.

We implement reservations atomically using a two-phase commit protocol [7]. The action starting with resource-request initiation and ending with resource-reservation commitment corresponds to an Infosphere session [5].

Reserving Resources. Our paradigm for resource reservation, using client requests and brokering agents, provides a test bed on which effective algorithms can be developed for specific tasks; see Figure 3. Resource reservation begins with a client application making a request, the only interaction a user needs to have with the system. A resource can be represented by a boolean function over all possible Cartesian products of resources and meeting times, with additional weights given to represent hints. For example, requiring one Pentium and one SGI on Monday at 10AM is a request that assigns boolean `true` to all combinations of resources that include the desired Pentium and SGI. In addition, hints can help the system choose more appropriate scheduling policies. Although the general framework is too complex to implement directly in some applications, for any particular application a suitable subset can be implemented.

Like ambassadors to foreign countries, the client system can send a small set of instructions in Java as *agents* [10] to any resource manager to request computing time. Several efficiency improvements make agent communication attractive. Agents can include user preferences for efficient filtering of available times at the server end. The filtered set can then be returned to the client, thereby avoiding heavy message passing in congested or high latency networks. Since nearby agents can designate a common agent to efficiently set up a coordinated reservation time among these agents, hierarchical solutions can be used to obtain lists of available slots. By varying the programs that the agents define, different algorithms can easily be tested without major modifications in the system.

Not only can clients send agents to servers, but servers can send agents to clients to request back slots that clients had on hold, upon request from a client that has higher priority. Our system requires that the agent recipient must lock the slot withing a time period, or the slot will be automatically returned to the resource manager.

Agent Primitives. Scheduling agents communicate with resource managers on servers using query, lock, release, and wait messages (Figure 4).

Queries. A query is the first communication an agent makes when setting up a meeting. When a server receives a query, it gives the client's agent complete (access-dependent) information about which slots are available, held, and locked. The agent relays (a possibly filtered version of) this information back to the client. However, it also executes quickly on the server, filtering this information to reserve some vacant slots and wait for its client to decide what to do with them. The server may impose restrictions on how many slots the agent can reserve at any one time. We could dispense with agents and allow the server to pick the slots it reserves for the client; although this is our implementation default, the agent innovation allows the client to encode some preference information and have it honored without the lag of message-passing.

Locks and Releases. The server allows authorized clients to lock slots they hold or release uncommitted slots. It sends released slots to the highest-priority client on the waiting list if one exists.

Waits. The server can receive requests to be placed on the waiting list for specific slots. If the slot is held, but not committed, the server will honor the request and, if the requester has higher priority than the current holder, the server will request the current holder to return the slot. The holder must lock the slot within a certain time period, or it will be returned automatically.

4 Applications

Two applications illustrate our framework: scheduling specific resources controlled by more than one resource manager, and scheduling by attribute.

Multiple Resource Managers. Consider scheduling two or more resources, each controlled by a different manager. One solution is to use local clocks (discussed in Section 2) to place on hold each resource's calendar before scheduling computing time by locking the appropriate slots. That has the efficiency problem discussed in Section 2, but it will be smaller since we are using the algorithm only to schedule calendars, not resource use. Thus, just introducing the calendar metaphor provides substantial savings.

In this algorithm, when one user is reserving time on a given resource, all other users are excluded, while in reality we need mutual exclusion only on individual slots for safety. We can therefore improve this algorithm's efficiency: a client can use finer-grained adaptive control to place on hold only a small part of the resource manager's calendar at a time.

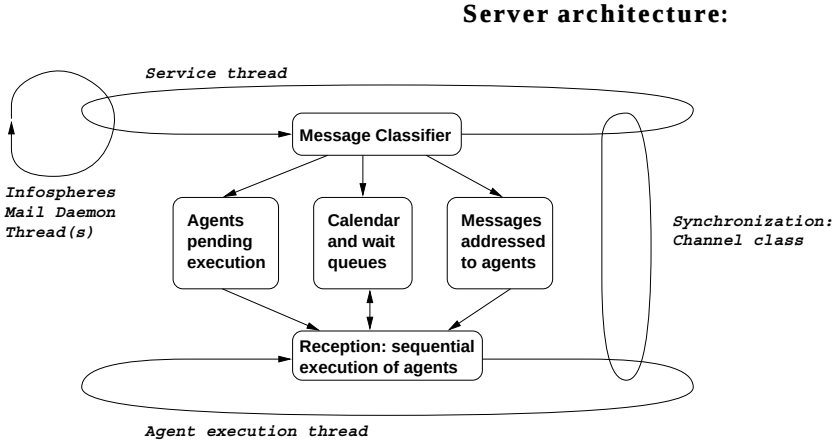


Fig. 4. Agents execute atomically and communicate with their server and the outside world via a receptionist class which provides only non-blocking send and receive methods. Though agents run in the address space of the server, privacy is possible via the Java security manager.

Query. The client indicates interest in scheduling a resource by sending agents to the various servers. The agent executes the program given to it by the client, and after communicating with the resource manager returns to the client's system a list of slots that it is currently holding, and information (which in case of free slots may not be up to date) on whether the remaining slots are free, on hold, or locked. Note that the server pipes available slots through the agent giving the client a small number of desirable held slots. The agent is an efficient way to encode client preferences cheaply.

Schedule if possible. The client can then schedule computing time if at least one slot returned to it from all of the resources matches. It writes to that slot, committing to using all of the resources at that time, makes any necessary payments for use of those resources, and releases the remaining slots.

Negotiate. If the client cannot immediately schedule all required resources, it negotiates instead of just giving up and retrying. Specifically, it releases all held slots that were locked by other users in at least one desired resource, since there is no chance of getting all desired resources for that time slot. It then enters a queue on other slots, where there is still a chance of acquiring all desired resources. Using logical clocks, the algorithm continues until resources are reserved, or no reservation is possible. The negotiation phase is usually unnecessary.

Change the agent. Based on the type of negotiation required, the client can keep evolving its agent to better meet its changing needs for placing holds as well as locking and releasing slots.

Resource Reservation by Attribute. Offering reservation by attribute (for example, a request for "3 SGIs and 2 Pentiums") is easily integrated into our

existing framework. The "and" clause defines resources that must be reserved together, so these can be treated as specific resources themselves. This reduces to a scheduling problem such as "get 3 SGIs out of the 30 known to my system." We want n out of k homogeneous resources where $(k > n)$. We can use the algorithm for multiple resource managers if we send agents to p out of the k resources, but then pick the "best" (or earliest) time at which n out of the p polled resources are available. Our problem then reduces to choosing p; choosing $(p = k)$ may not always be the best solution due to message passing delay. For this reason, we have developed a simple mathematical model for choosing the optimal p. In our model, the expected *delay* in scheduling a job is computed for each p using the probability q that a given slot is unavailable. We then plot a graph of cost versus p to find the p that minimizes the delay.

5 Conclusions

We have investigated generalizable resource allocation algorithms for which desired resources can be specified by attribute only, and for which different resource managers can coordinate synchronously. Our model builds on the concept of *resources as tokens* and the metaphor of *calendars for scheduling*. To improve efficiency under high network latency, our implementation passes small Java programs as agents for coordination. Our design represents the first step toward the development of a robust scheduling infrastructure, layered above conventional schedulers currently available, for the next generation of virtual supercomputers constructed from heterogeneous resources distributed over the Internet.

References

1. C. Catlett and L. Smarr. Metacomputing. *Comm. of the ACM*, 35:44–52, 1992.
2. K.M. Chandy, J. Kiniry, A. Rifkin, and D. Zimmerman. A framework for structured distributed object computing. *Parallel Computing*, 1997. Submitted.
3. K.M. Chandy, J. Kiniry, A. Rifkin, and D. Zimmerman. Webs of archived distributed computations for collaboration. *Journal of Supercomputing*, 11(1), 1997.
4. K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.
5. K.M. Chandy and A. Rifkin. Systematic composition of objects in distributed internet applications: Processes and sessions. *Proceedings of the Thirtieth Hawaii International Conf. on System Sciences*, pages 395–404, January 1997.
6. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997.
7. J. Gray and A. Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.
8. D.A. Lifka, M.W. Henderson, and K. Rayl. Users guide to the argonne sp scheduling system. Technical Report ANL/MCS-TM-201, Argonne, May 1995.
9. M. Litzkow, M. Livney, and M. Mutka. Condor – a hunter of idle workstations. In 8^{th} *International Conf. on Distributed Computing Systems*, pages 104–111, 1988.
10. P. Maes. Agents that reduce work. *Comm. of the ACM*, 37(7):31–40, July 1994.