Using Announce-Listen with Global Events to Develop Distributed Control Systems*

K. Mani Chandy, Adam Rifkin, and Eve Schooler
Caltech Department of Computer Science 256-80, Pasadena, CA 91125
{mani,adam,schooler}@cs.caltech.edu

February 6, 1998

Abstract

We specify an abstract model for dynamic distributed control systems in which the component objects make local decisions based on system-wide constraints and approximate global state. We focus on the issue of distributed resource management, exploring a solution that is both compositional and scalable because it builds global events into the Java infrastructure by exploiting its multicast facilities.

1 Introduction

A *distributed control system* consists of interacting component objects, each of which have persistent local state and one or more threads of control. Abstractly, a simple specification of a distributed control system consists of the descriptions of several components: *state*, *computation*, *events*, and *constraints*.

State. The state of a distributed system is defined in terms of its components. In our context, the components are the system's participating objects, as well as the communication infrastructure itself. The objects are persistent, communicate with other objects, and have one or more local threads of control. Each object has access to its own local state — the values of its variables, its pending inputs and outputs, and its threads of control. However, no object can directly access the local states of other objects, nor can it directly access the entire state of the distributed system itself.

Computation. The computation of a distributed system specifies the state of the components at a given time. In control terms, we consider the computation to be the trajectory of the system's state.

Events. A distributed system may receive inputs from its component objects or from the uncontrollable external environment. Likewise, its component objects may generate output that is directed either internally or externally. Communications may have arbitrary delays.

Constraints. A system's computation is determined both by its inputs and by the constraints of its objective function, which determine its state transitions. An example of a constraint is that a high-priority request is always serviced before a low-priority request for the same set of resources, if both requests are made at the same time and place. An example of an objective function is to minimize the average response time for a request.

*This work was supported under the Caltech Infospheres Project is sponsored by the Air Force Office of Scientific Research under grant AFOSR F49620-94-1-0244, by the CISE directorate of the National Science Foundation under Problem Solving Environments grant CCR-9527130, by the NSF Center for Research on Parallel Computation under Cooperative Agreement Number CCR-9120008, by a Microsoft Graduate Fellowship, and by Parasoft and Novell. More information about Infospheres is available at http://www.infospheres.caltech.edu/ on the Web.

Design Challenges. Several aspects of distributed control systems make the problem of designing the control challenging:

1. Because of their distributed nature, component objects do not have direct access to the state of the entire system, so they know neither the states of other objects, nor the status of communications in transit.
2. New objects enter and leave the system dynamically, so an object may not know which other objects are in the system at any given point in a computation.
3. Each object makes its own local decisions. By the constraints and objective function, however, we require the local decisions to result in an approximately optimal system-wide outcome.

The distributed control model fits a wide range of applications from flight control of a flock of uninhabited autonomous vehicles to determining the proper scope for a multicast session-invitation application. The common problem is that each object has to make an autonomous decision based on old (and possibly partial) information about other parts of the system. In this paper, we restrict attention to resource management systems.

2 Distributed Resource Management

Consider the general problem of distributed resource management: *consumers* request *resources* from a finite pool of *providers*.

2.1 Problem Specification

Specifying this problem involves choosing among several axes:

- *Consumers* — collaborating (e.g., in metacomputing) or competing (e.g., for airline tickets).
- *Resources* — indivisible tokens to represent individual resources (e.g., a particular time slot for scheduling) or slidebars to represent ranges of resource collections (e.g., a slice of bandwidth required).
- *Providers* — collaborating (e.g., to piece together less expensive service packages) or competing (e.g., to keep sensitive information hidden).

An example of distributed resource management is arranging a ski trip. With the proper compositional structures in a distributed vacation-control system, you might announce your requirements (for example, ski trip price, kind of slopes, and calibre of hotels) and have providers respond with competing proposals. An alternative approach involves *middlemen* objects that act on behalf of the consumers to search for service providers, putting together complete packages by filtering, aggregating, and collating information.

2.2 Solution Categories

Solutions to distributed resource management can be constructed along several axes:

- *State management* — soft state (i.e., approximations or probabilities) or hard state (i.e., invariants).
- *Communication mechanisms* — point-to-point casting (i.e, multiparty communication built on individual connections), broadcasting, or multicasting (i.e., efficient broadcast using announce groups).
- *Compositional structures* — announcing, listening, or mediating (e.g., forwarding) requests and responses.

In this paper, we investigate multicast algorithms for distributed resource management. We describe the concept of soft state, and use it to build a *consumer announce, provider listen* model in which the consumers compete for resources (represented as tokens) from collaborating providers.

3 A Framework for Distributed Resource Management

Any distributed resource management algorithm requires the location, reservation, and scheduling of resources. Our algorithms to perform these tasks are based on *soft state* approaches — relying on approximate rather than absolute global snapshots. Each distributed object periodically *announces* its local state information and *listens* for updates from other objects. In this section, we outline our messaging implementation, and describe the resource management components layered above it.

3.1 Soft State

An object estimates the current state of a distributed system based on information received earlier from other objects; thus the estimated soft state is based on old information. For instance, if an object B is listening on a multicast address at time t, then we may know with high probability that B will continue to listen for some more time. If at time $(t + t')$, object A receives a message from B, timestamped t, stating that B is listening, then A has the "soft" information that B is listening at time $(t + t')$; and the state softness is quantified in terms of probability [4].

By contrast, "hard" state is a property that follows from an invariant of the system. For instance, consider the invariant: If variable x in object A has value 1, then variable y in object B has value greater than or equal to 1. In this case, if x in object A has value 1, then object A "knows" that y in object B has value at least 1, and this knowledge is hard [2].

Whether soft state can be used depends on the nature of the problem, the relative frequency of announcements to state changes, and the flexibility of constraints.

3.2 Global Events

The announce-listen paradigm is used at the messaging layer to assist in resource location, reservation, and scheduling. We have implemented this messaging facility in Java as *global events*.

Java Beans provide *local events* as a mechanism by which a component informs other components that something interesting has happened. These events can be thought of as active messages; for example, a button is pressed at a source, and channeled through an event listener, to trigger a method in an event observer automatically. An event propogates from an *event source* through an *event notifier* to one or more *event observers* that respond to the events as they arrive. The notifier routes the event to the observers using a control list, and observers can ask the notifier to be added or removed from this list without notifying the event source.

Our global event structuring mechanism is identical to the local event model of Java Beans, except that instead of Java Beans' referencing an object within a single Java Virtual Machine, we use a global name for the object, employing the Web's URL convention. Furthermore, because the components of the global event system are distributed, multicast can be used for efficient group communication, instead of Java Beans' local event point-to-point casting.

Using global events, an event is *announced* by a source object in one virtual machine, and notifiers for that event in other virtual machines anywhere on the Internet *listen* for the event and forward it to the appropriate (distributed) observers. Unlike the group communication in *virtual synchrony* [1], it is not necessary for the event sources to know at any point who the event observers will be. Our global event model is useful not only to distribute events and the objects that use them, but also to compose event notifiers, to filter using predicates, and to provide security using access control lists at the event notifier level.

There are several advantages to using global events and soft state. The announce-listen paradigm is fault-resilient [7]; that is, if a resource provider goes away, the system adapts dynamically to continue to meet the requests of the consumers. Furthermore, systems constructed using global events and multicast are compositional and scale; providers and consumers can add or remove themselves at any point dynamically. Unfortunately, such systems also have the potential for oscillation; that is, if state changes faster than the communication updates, then soft state may give a bad estimate of the current system state. We are currently exploring the tradeoffs through simulation and implementation.

3.3 An Algorithm for Consumer Announce, Provider Listen

Consumers use global events to announce their requests for resources to providers, including estimates of how long they will need those resources. Multicast allows us to limit the scope of announcements, as illustrated in figure 1. Additionally, multicast is useful for progressively searching regions with wider scopes.

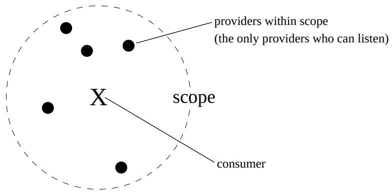

Figure 1: Only providers within the scope of the consumer announcements can hear the announced events.

When a listening provider can give a partial or total basket of tokens fulfilling a given request, it uses global events to announce to the other providers what tokens it can commit to that consumer. Since providers collaborate to satisfy each consumer's request, the listening providers can ante up additional resources (or keep from doing so) based on previous information announced by providers and consumers.

As illustrated in figure 2, multicast provides a scalable bus abstraction that allows any number of objects to participate in group communications; however, unlike virtual synchrony [1], multicast does not guarantee reliable delivery. Not all listening providers will receive all consumers' requests, but consumers can increase announcement scopes if they receive no viable responses to their requests.

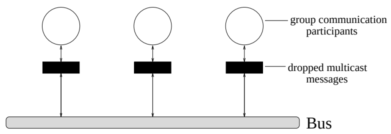

Figure 2: Multicast provides a scalable bus abstraction, but messages are not guaranteed reliable delivery.

Deadlock Avoidance. This algorithm could present a problem if middlemen are used, because deadlock may occur if no consumer obtains all of the resources it needs. To solve this problem, we stipulate that consumers must give up a provider's tokens, if asked by that provider, even if all of the requested tokens have not been received yet. This scheme works using *global priorities*: the priority of a request event is based on the local timestamp of that request, given by the requestor's local logical clock, with ties broken by the alphabetic comparison of requestors' globally unique names. Global priorities represent the use of an invariant in an algorithm. We can assert that if a request R has a higher priority than another request R' at one global object, then R has a higher priority than R' at *all* global objects. All objects share "common knowledge" about the global priorities because they listen on the same multicast bus. The combination of soft state messaging with hard state invariants results in an algorithm that scales but is free from deadlock.

Oscillation Avoidance. Using global priorities, a provider demands tokens back from its lowest priority requestor, who must then return them. Because requests and responses are shared among all participants (consumers, providers, or even middlemen) on the multicast bus, providers are able to make good estimates of which consumers need which resources, and thereby avoid "oscillation" of requests for and returns of tokens. Here, system components use soft state to approximate the true current state of the system, which works well, provided the system state does not change rapidly compared with announcement frequency.

Location Services. Because our architecture is based on multicast, introducing and removing system components is a simple operation. For example, a new consumer can easily locate middlemen and providers after obtaining a small set of multicast addresses on which to announce requests and to listen for responses. The result is an automatically-configurable directory infrastructure for locating objects and resources. To prune the search space, announce-listen is also used to disseminate and to cache information closer to requestors.

Flexible scoping is used to make our location algorithms more efficient. When a consumer requests resources, listening providers can announce back the rough estimates of when their resources will be available. As illustrated in figure 3, a consumer that does not want to wait then either increases its scope and re-announces its request, or sends its request to proxying agents outside the scope that announce the request elsewhere.

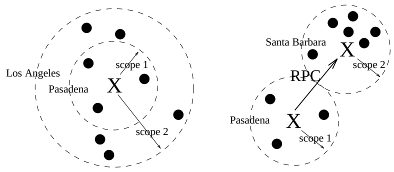

Figure 3: To continue a search, a consumer increases its scope or uses a proxy to search in another scope.

Reservation and Scheduling. Providers employ a calendar metaphor with time slots for arranging the usage schedules of their resources. Each slot for each resource is in one of three states: *available* for use by a consumer, *reserved* for a particular consumer but not yet locked, or *scheduled*. Each slot has an associated *access list* that keeps track of which consumers can obtain that lock.

The reservation of a set of resources is determined when all of the resource managers agree to lock the slots that correspond to the same time. Consumers use a two-phase commit protocol to reserve and then to schedule desired resources atomically [5, 6]. The action beginning with resource-request initiation and ending with resource-reservation commitment corresponds functionally to an Infosphere session [3].

4 Status and Future Work

The overall goal of any distributed resource management system is the efficient matching of resource providers and requestors. The Infospheres Infrastructure 2.0 includes several packages to assist with this task. In this paper, we have focused on building global events into Java, and exploiting its multicast facilities, to develop distributed control announce-listen algorithms that are both scalable and fault-tolerant. Presently, we are investigating the tradeoffs between soft state and hard invariants, between pushing and pulling resource requests and responses, and between a hierarchy of middlemen and a flat requestor-provider structure.

References

[1] K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos, California, 1994.

[2] K. M. Chandy and J. Misra, 'How Processes Learn', *Journal of Distributed Computing*, Volume 1, Number 1, Pages 40–52, 1986.

[3] K. M. Chandy and A. Rifkin, 'Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions', *Computer Journal*, Oxford University Press, October 1997.

[4] K. M. Chandy and E. M. Schooler, 'Designing Directories in Distributed Systems: A Systematic Framework', *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Pages 318–328, Syracuse, New York, August 1996.

[5] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[6] R. Ramamoorthi, A. Rifkin, B. Dimitrov, and K. M. Chandy, 'A General Resource Reservation Framework for Scientific Computing', *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, Volume 1343 of Springer-Verlag's Lecture Notes in Computer Science, Pages 283–290, Marina del Rey, California, December 1997.

[7] E. M. Schooler, 'A Multicast User Directory Service for Synchronous Rendezvous', Technical Report CS-TR-96-18, Department of Computer Science, California Institute of Technology, Pasadena, California, 1996.