Active Middleware Services in a Decision Support System for Managing Highly Available Distributed Resources

Sameh A. Fakhouri[1], William F. Jerome[1], Vijay K. Naik[1], Ajay Raina[2], and Pradeep Varma[3]

[1] IBM T. J. Watson Research Center, Hawthorne, NY 10532
{sameh,wfj,vkn}@us.ibm.com
[2] IBM Global Services, Bangalore, India
rajay@in.ibm.com
[3] IBM India Research Laboratory, New Delhi, India
pvarma@in.ibm.com

Abstract. We describe a decision support system called Mounties that is designed for managing applications and resources using rule-based constraints in scalable mission-critical clustering environments. Mounties consists of four active service components: (1) a repository of resource proxy objects for modeling and manipulating the cluster configuration; (2) an event notification mechanism for monitoring and controlling interdependent and distributed resources; (3) a rule evaluation and decision processing mechanism; and (4) a global optimization service for providing decision making capabilities. The focus of this paper is on the design of the first three services that together connect and coordinate the distributed resources with the decision making component. We discuss the overall architecture and design of these services. We describe in some detail the asynchronous, concurrent, and pipelined nature of their interactions and the fault tolerance designed in the system. We also describe a general programming paradigm that we have followed in designing these services.

1 Introduction

A cluster is a collection of resources (such as nodes, disks, adapters, databases, etc.) that collectively provide scalable services to end users and to their applications while maintaining a consistent, uniform, and single system view of the cluster services. By design, a cluster is supposed to provide a single point of control for cluster administrators and at the same time it is supposed to facilitate addition, removal, or replacement of individual resources without significantly affecting the services provided by the entire system. On one side, a cluster has a set of distributed, heterogeneous physical resources and, on the other side, it projects a seamless set of services that are supposed to have a look and feel (in terms of scheduling, fault tolerance, etc.) of services provided by a single large virtual resource. Obviously, this implies some form of continuous coordination and mapping of the physical distributed resources and their services onto a set of virtual resources and their services.

Typically, such coordination and mappings are handled by the resource management facilities, with the bulk of the work done manually by the cluster administrators. Despite the advances in distributed operating systems and middleware technology, the cluster management is highly human administrator bound (and hence expensive, error-prone, and non scalable beyond a certain cluster size). Primary reasons for such a state-of-the-art is that existing resource management systems adopt a static resource-centric view where the physical resources in the cluster are considered to be static entities, that are either available or not available and are managed using predetermined strategies. These strategies are applied to provide reliable system-wide services, in the presence of highly dynamic conditions such as variable load, faults, application failures, and so on. The coordination and mapping using such an approach is too complex and tedious to make it amenable to any form of automation.

To overcome these difficulties, we take an approach that is different from the traditional resource management approach. In this approach, resources are considered as services whose availability and quality-of-service depends on the availability and the quality-of-service provided by one or more other services in the cluster. For this, to state it informally, the cluster and its resources are represented by two dimensions. The first dimension captures the semi-static nature of each resource; e.g., the type and quality of the supporting services needed to enable its services. Typically, these requirements are defined (explicitly or implicitly) by the designers of the resource or the application. These may be further qualified by the cluster administrators. These are formalized as simple rules that can be dynamically and programatically evaluated, taking into account the current state of the cluster. The second dimension is the dynamic state of the various services provided by the cluster. The dynamic changes are captured by events. Finally, all the coordination and mapping is done at a logically centralized place, where the events are funneled in and the rules are evaluated. This helps in isolating and localizing all the heterogeneity and associated complexity. By separating the dynamic part (the events) from the semi-static parts (the rules), and combining these in a systematic manner only when needed, the desired level of automation in the coordination and mapping of resources and services can be achieved.

While the general principles outlined above are fairly straightforward, there is a nontrivial amount of complexity in managing the choreography. To show the proof of concept, we have designed and implemented a system called Mounties based on the above described general principles. The Mounties architecture itself is composed of multiple components, a primary component being the modeling and decision making engine. The remaining components together form an active and efficient resource management layer between the actual cluster resources and the decision-making component. This layer continuously transports the state information to the decision maker and commands from the decision maker to the cluster resources, back-and-forth in a fault-tolerant manner. In this paper, we describe in detail the architecture and design of the services that form this middleware.

The remainder of the paper is organized as follows. First we define some terms and cluster concepts and then, in Sect. 3, briefly describe the overall Mounties approach. Following that, in Sect. 3.3, we present a small example to illustrate some of the key concepts. An overview of the Mounties architecture and its design is described in Sect. 4. Described in Sect. 5 are the salient features of the three main services that coordinate the actions between the cluster resources and the decision making component. In Sect. 6, we describe the programming paradigm that we have followed in designing these services. Finally, we conclude the paper after reviewing the related work, in Sects. 7 and 8, respectively.

2 Definitions and Basic Cluster Concepts

In a cluster managed by Mounties, hardware components such as nodes, adapters, memory, disks, and software components such as applications, database servers, web servers are all treated as cluster *resources*. When there is no ambiguity, in this paper, we use the terms resource and the service it provides, interchangeably. A *location* is a unique place in the cluster where a resource or service physically resides and makes its service available. Typically it is identified by the node (or the processing element), but it could be any uniquely identifiable location (such as an URL). To provide its intended services, a resource may need services provided by one or more other resources. These are referred to as the *dependencies*. In addition to the dependencies, a resource may have other limitations and restrictions such as capacity (defined in the following) or location in the cluster where it can provide its services. Some of these may be because of the physical limitations of the resource while others may be imposed by the cluster administrators. The dependencies and the specified limitations together form a set of *constraints* that must be satisfied for making a service available. Usually the cluster administrator satisfies these constraints by *allocating* appropriate resources. Typically, a cluster is expected to support multiple services. To achieve this, constraints for multiple resources must be satisfied simultaneously, by judiciously allocating lower level supporting resources and services. This hierarchical allocation of resources (i.e., one level of resources supporting the next level of resources) gives rise to a particular *cluster configuration* where dependency relations are defined among cluster resources. Note that there may be more than one possible cluster configuration to provide the same set of services. When there are only a limited number of resources or when the constraints among resources are complex, there may only be a small number of ways in which cluster can be configured to satisfy all the constraints. Determining such unique configurations is a hard problem.

Resources have attributes that distinguish them from one another. These include Name, Type, Capacity, Priority, and State. Each resource has a unique *Name* and resources are classified into multiple *Types* based on the functionality they provide. *Capacity* of a resource is the number of dependent resources that it can serve simultaneously. The capacity may be inherent in the design of a resource or it may be imposed by cluster administrators for performance or testing purposes. All allocations of a resource must ensure that its capacity constraints are not violated. *Priority* denotes the relative importance of a resource or a service. In Mounties, the Priority is a number (on a scale of 1 to 10, 1 being the lowest) to indicate its relative value. It is used in more than one way. For example, if two resources depend on a resource that can only support one of them, then one way to resolve the conflict is to allocate the scarce resource to the resource with higher priority. Similarly, in a cluster there may be more than one resource of a certain type and a resource or service that depends that type of resource may have a choice in satisfying that dependency. Here Priority of the supporting resources may be used to make the choice. The Priority field can also be used in stating the goals or objectives for cluster operation; e.g., resources may be allocated such that the sum of the Priorities of all services made available is maximized. The *State* of a resource indicates the readiness of its availability. In Mounties, the State of a resources is abstracted as ONLINE, OFFLINE, or FAILED. An ONLINE resource is ready and is available for immediate allocation, provided its capacity is not exhausted; An OFFLINE resource could be made ONLINE after its constraints are satisfied. A FAILED resource cannot be made available just by satisfying its constraints. The FAILED state is indicative of either a failure because of an error condition or unavailability because of administrative servicing requirements.

Finally, we note that throughout the paper we use the term *end users* to mean the cluster administrators, the applications that use the cluster services, or the end users in the conventional sense. In practice, cluster administrators and high level applications tend to be the real users of the services provided by Mounties.

3 The Mounties Approach

As described in the introductory section, Mounties introduces a constraint-based methodology for the cluster configuration, startup and recovery of applications and other higher level resources. The constraints are used to build relationships among supporting and dependent resources/services. Under this approach, the heterogeneity and nonuniformity of the physical cluster are replaced by the consistent and single-system like service views. This is further enhanced by providing higher-level abstractions that allow end users to express requirements and objectives that are tailored to a particular cluster and the organization using the cluster.

3.1 Basic Rules and Abstractions

In a cluster, certain services are expected to be normally available. In Mounties, this is expressed by means of a resource attribute called the *NominalState*. The NominalState acts as a constraint for one or more resources in the cluster and this information becomes a part of the cluster definition. To indicate the normal availability of the services of a resource, the NominalState of that resource is set to ONLINE. This constraint is satisfied when the State of that resource is ONLINE. Furthermore, the ONLINE NominalState implies that every effort must be made to keep that service ONLINE. Similarly, a NominalState of OFFLINE is sometimes desirable; e.g., for servicing a resource or when the cost of keeping a resource on-line all the time is too high.

When a resource or service has an ONLINE NominalState, the cluster management system needs to be informed about how the resource or service can be brought on-line. Typically, most services or applications depend on other lower level services or resources. Mounties provides two main abstractions for expressing the inter-resource dependencies: the *DependsOn* relationship and the *CollocatedWith* relationship. Resource A DependsOn B if services of Resource B are needed for the liveliness of A. Note that a resource or an application may require services of more than one type of other resources. Generally these services may be available anywhere in the cluster. In certain cases, only the services provided by local resources can be used. To express such a location specific constraint a CollocatedWith relationship is used. For example, Resource A CollocatedWith B means Resource A must have the same location as that of B; i.e., they must reside on the same node. Note that services of B may be available at more than one location. In that case, there is a choice and a decision has to be made about the location that is to be picked. Similarly, sometimes it is desirable not to locate two resources on the same node. This is expressed by the Anti-CollocatedWith constraint.

Mounties provides a new resource abstraction called an Equivalency. Informally, an equivalency is a set of resources with similar functionality, but possibly with different performance characteristics. It has a run-time semantics of "choose one of these". Since the selection of the most appropriate resource from an equivalency depends on the cluster-state, the concept of equivalencies provides Mounties with a strong and flexible method to meet the service goals of the cluster. With this abstraction, the end-user is freed from making ad-hoc decisions and allows Mounties to choose the most appropriate resource based on the conditions at run-time. An equivalency can also be associated with a weighting function, called a policy. A policy can guide, but not force, the decision- making mechanism within Mounties towards a particular selection based on end-user preferences or advanced knowledge about the system. Since an equivalency can be treated as a resource, it maintains uniformity in specifying constraints and at the same time allows specification of multiple options that can be utilized at run-time.

Finally, Mounties provides abstractions for defining *business objectives* or goals of how the resources in the cluster are to be managed and configured. These objectives typically consist of maintaining availability of cluster services and of individual resources in a prioritized manner, allocation of resources so as to balance the load or services, or delivering a level of service within a specified range, and so on.

3.2 Management and Coordination of Resources

At the lowest levels, all resources are manipulated in a programmable manner or from the command line. Mounties divides the work such that the decision making and resource allocation processes (which require global knowledge about the cluster) are distinct from the resource monitoring, controlling, and manipulating processes (which require resource specific information) such as the resource managers. This encapsulation of resource manipulation gives flexibility and requires no special programming in order to add an application into the cluster once its resource manager is available. For the purpose of this paper, we will not focus on the topic of resource managers.

Mounties gathers and maintains information about the cluster configuration and the dependency information for each resource at cluster startup or whenever a new resource or application is introduced in the cluster. A continuous event notification and heartbeat mechanisms are also needed for monitoring cluster-wide activities. Using these mechanisms, Mounties continuously monitors the cluster-wide events and compares the current cluster-state with the desired state. Whenever there are discrepancies between the two, the best possible realignment of resources is sought after taking into account the conditions existing in the cluster and the desired cluster-wide objectives. If a new realignment of resources can lead to a better configuration, commands are issued to the resources to bring about the desired changes.

We now illustrate these concepts using a simple, but realistic example.

3.3 An Example

Our example involves a cluster of three nodes shown in Fig. 1. Both Node 0 and Node 1 have disk adapters that connect them to a shared disk which holds a database. Each node has a network adapter which connects it to the network. The services of this cluster are used by a Web Server as shown in Fig. 2.

The hardware and software components shown the Fig. 1 are defined to Mounties along with their attributes and are treated as resources. For example, the disk adapter 0 has the following attributes:

```
Disk Adapter 0 Attributes
     {
        Capacity = 1
        Priority = 2.0
     }
```

The nodes and other adapters in the system are defined to Mounties in a similar manner. Using these basic resources, a set of equivalencies are defined. As explained earlier, an equivalency is a grouping of the same type of resources and is treated as an abstract resource. In our example, Equivalency 1 groups the two disk adapters into one new resource. Similarly, Equivalency 2 groups the three network adapters into one new resource.

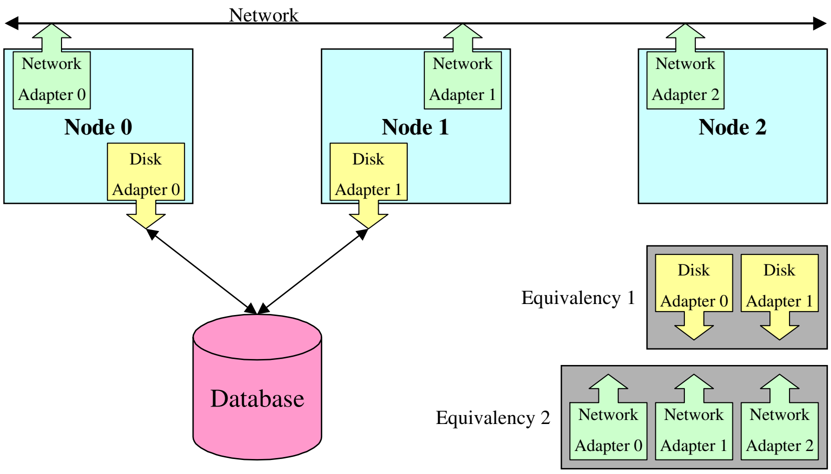

Fig. 1. An example cluster configuration managed by Mounties

The database itself has two engines that can be brought on-line only on the nodes with both disk and network adapters. Figure 2 shows the dependencies for the two database management engines. Database engine 0 has the following attributes:

```
Database 0 Attributes
     {
        NominalState = ONLINE
        Priority      = 8.0
        DependsOn     = Equivalency 1, Equivalency 2
        CollocatedWith = Equivalency 1,Equivalency 2
     }
```

Database engine 1 is defined in the same manner. Aside from having a relatively high priority of 8, both engines have a NominalState of ONLINE. This indicates to Mounties that it should try an keep them both ONLINE at all times. In addition, the database engines have dependencies and collocation constraints on both Equivalency 1 and 2. Both constraints are represented in Fig. 2 by the bi-directional arrows linking the Database engines to the Equivalencies.

Mounties represents these constraints as follows: For each Database engine to be online we need a Disk Adapter, a Network Adapter and they must be located on the same node as the Database engine. So, if Mounties were to pick Disk Adapter 0 from Equivalency 1 to satisfy the requirements of Database 1 for a disk adapter, the collocation constraint will force it to also pick Network Adapter 0 from the Equivalency 2. So, to make Database 1 ONLINE, Mounties would perform the following allocations:

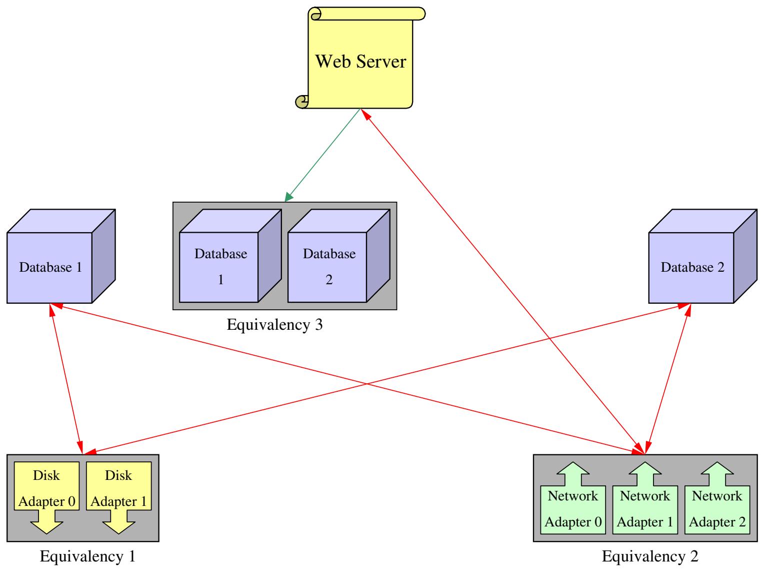

Fig. 2. Dependencies for a Web Server supported by the example cluster of Fig. 1

```
Database 1
    {
        From Equivalency 1 = Disk Adapter 0
        From Equivalency 2 = Network Adapter 0
        Node Assignment    = Node 0
    }
```

These allocations satisfy all the constraints of Database 1, therefore it can be brought ONLINE. When allocating resources for Database 2, neither Disk Adapter 0 nor Network Adapter 0 are eligible because their capacity is exhausted. Mounties cannot allocate Network Adapter 2 from Equivalency 2, since there is no Disk Adapter on Node 2 that would satisfy the collocation constraint. The only choice then is the following allocations for Database 2:

```
Database 2
    {
        From Equivalency 1 = Disk Adapter 1
        From Equivalency 2 = Network Adapter 1
        Node Assignment    = Node 1
    }
```

These allocations satisfy all the constraints of Database 2, therefore it can be brought ONLINE.

Figure 2 also shows Equivalency 3, which contains both Database engines. Shown also is a new resource, Web Server which has the following attributes:

```
Web Server Attributes
     {
        Nominal State = ONLINE
        Priority      = 6.0
        DependsOn     = Equivalency 2, Equivalency 3
        CollocatedWith = Equivalency 2
     }
```

The dependency and collocation constraints are shown with the bi-directional arrows linking the Web Server to Equivalency 2. The dependency is shown with the uni-directional arrow linking the Web Server to Equivalency 3.

Given the previous assignments that Mounties made to bring the Database engines up (i.e., make their State ONLINE), the only available Network Adapter from Equivalency 2 is Network Adapter 2. To satisfy the Web Server's dependency on Equivalency 3, Mounties could pick Database 1. So, to bring the Web Server to the ONLINE state, Mounties would perform the following allocations:

```
Web Server
     {
        From Equivalency 2 = Network Adapter 2
        From Equivalency 3 = Database 1
        Node Assignment    = Node 2
     }
```

This completes the resource allocations necessary to bring all resources to the ONLINE state. While running, if Database 1 should fail for any reason, Mounties would switch the Web Server over to Database 2 and thus keep it ONLINE.

We note here that in the above, we have described the decision making process in an intuitive manner. In Mounties, this process is formalized by modeling the problem as an optimization problem with specific objective functions defined by cluster administrators. The optimization problem encapsulates all the relevant constraints for the cluster resources along with desired cluster objective. Good solution techniques invariable involve performing global optimization.

4 Mounties Design Overview

In previous section, we have discussed the resource management concepts used in Mounties. We now describe the Mounties architecture and its design in some detail, and provide rationale for our design decisions where appropriate.

A cluster is a dynamically evolving system and is constantly subject to changes in its state because of the spontaneous and concurrent behavior of the cluster resources, random and unpredictable nature of the demands on the services, and the interactions with end users. At the same time, a cluster is expected to respond in a well-defined manner to events that seek to change the cluster-state. Some of these events are:

1. Individual resource related events such as: resource is currently unavailable; unavailable resource has become available; a new resource has joined the cluster; a resource has (permanently) left the cluster.
2. Feedback response to a cluster manager command: successful execution of a command such as go online or go offline; failure to execute such a command.
3. End user interactions and directives: cluster startup and shutdown; resource isolation and shutdown; manual overrides for cluster configurations; movement of individual and/or a group of resources; changes in dependency definitions and constraint definitions among resources; updates to business objectives; requests leading to what-if type of analysis, and status queries.
4. Resource groups related events, or virtual events, which arise from a combination of events/feedback related to individual resources.
5. Alerts and alarms from service and load monitors.

With these dynamic changes taking place in the background, a cluster manager such as Mounties is required to make resource allocation and other changes such that the predefined global objectives are met in the best possible manner, while resource specific constraints are obeyed. The resource specific constraints usually limit the number of ways in which the resources in the cluster can be configured. These constraints include capacity constraints, dependency constraints, location constraints, and so on. The objectives and the constraints lead to a solution of a global optimization problem that must be solved in soft real-time. This requires an efficient decision making component and a set of services that form an efficient middleware connecting the resources with the decision making component. Before describing how these components can be designed, first we describe the overall clustering environment in which a system like Mounties operates.

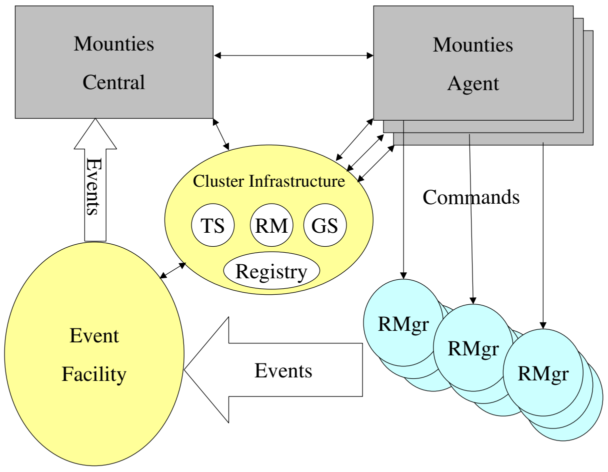

Fig. 3. Mounties design and its relationship cluster services for high availability

4.1 Cluster Infrastructure

The Mounties system as described here can be used as an application/resource management system or as a subsystem for guaranteeing high availability and quality-of-service for other components in the cluster. When used an application/resource management system, the Mounties system described here can basically be used in a stand-alone mode. When used as a guarantor of dependable services, a few other cluster services are required. In Fig. 3, we illustrate a conceptual design of Mounties on the top of basic high availability services. Using these services, Mounties can then be used as an intelligent mechanism for guaranteeing high availability. Note that the basic cluster services that Mounties would depend on are provided as standard services in state-of-the-art clusters such as IBM's SP-2 System [6,7]. As shown in Fig. 3, four additional cluster services are needed to ensure high availability: (1) a persistent Cluster Registry (CR) to store and retrieve the configuration of the resources; (2) a mechanism called Topology Services (TS) for detecting node and communication adapter failures; (3) a mechanism for Reliable Messaging (RM) for important communication between Mounties Central and all the other Mounties Agents; and (4) a Group Services (GS) facility for electing a leader (i.e., Mounties Central) at cluster initialization and whenever an existing leader is unable to provide its services (because of a node failure, for example). We note here that the Mounties Repository and the Event Notification services (described in the next section) can be embellished to incorporate the functions provided by Cluster Registry and Reliable Messaging. Similarly, a customized version of Group Services can be designed into the Mounties architecture to monitor and elect Mounties Central.

4.2 Internals of Mounties Design

Overview and the Ideal. In brief terms, designing the internals of the manager described thus far is an exercise in coming up with software that can coordinate the following choreography: Events arise asynchronously, throughout the cluster. They are delivered to the coordinator (such as an ideal version of Mounties) using pipelined communication channels. The coordinator is programmed to respond to events in the context of a semi-static definition of the cluster, that consists of dependencies, constraints, objective functions etc. The coordinator's decision-making component, basically an optimizer, has to combine the dynamic events with the semi-static definition in order to arrive at a response to events. The response has to translate into simple commands to resources such as go ONLINE and go OFFLINE. The coordinator sends its commands to resources at the same time as when various events arise and traverse the cluster. The commands are also sent using pipelined communication channels. Thus there is a basic dichotomy in the activity of coordinating the choreography. At the one end there is the cluster of resources and the events it generates. At the other end there is the decision-making optimizer. In between the two is middleware that along one path, collects, transports, and fine-tunes events for the decision-maker, and on the reverse path, decomposes the decisions of the decision-maker into commands that are then transported to the individual cluster resources.

Ideally, the coordinator reacts to the events instantaneously. It is able to account for faults in command execution–not all commands may succeed–along with being able to respond to events and command feedback in a real-time manner. Suppose the ideal coordinator is an infinitely fast computation engine. In this case, the choreography becomes a seamless movement of events, commands, and commands feedback in a pipelined/systolic manner throughout the cluster. Events and feedback upon arrival at the coordinator get transformed instantaneously into commands that in turn get placed on channels to various resources. The coordinator is able to ensure that globally-optimal solutions get deployed in the cluster in response to cluster events.

In Mounties, the ideal coordinator as described above is approximated by one active Mounties Central that resides on one node to which all events and command feedback get directed. Mounties Central can change or migrate in response to say node failure. However, at one time, only one Mounties Central is active.

Command Execution Model. The next definition we add in deriving our practical system from the ideal alluded to above is a command execution model. The model builds fault tolerance and simplicity in the execution of commands by sacrificing pipelining. It uses the following protocol: A command contains all the state needed for its execution by a resource manager. A command is only a simple directive to a resource manager; e.g., “go ONLINE using X, Y, Z resources”, or “go OFFLINE”, and no more. A resource manager does not need a computation engine to handle conditional behavior or context evaluation at its site. To achieve this, no new command is sent out until Mounties is aware of the positive outcome of the commands that the execution of the new command depends on. It is up to Mounties Central to make the best use of the command feedback it receives in order to minimize command failure. So for example, after receiving an “go ONLINE” command, a resource manager need not find out whether its supporting resources are actually up. The resource manager should simply assume that to be the case. In general, the more effective Mounties is in managing such assumptions, more efficient is the overall resource coordination. Clearly, one of the things Mounties Central has to do is to issue the commands in the partial order given by dependencies. Thus, in order for a resource to be asked to go on-line, its planned supporting resources have to be brought up first. Only after that the resource is to be asked to go on-line using the specific supporting resources. Similarly, before bringing down a resource, all the resources dependent on that resource must be brought down first. The existing and the planned dependencies in the cluster thus enforce a *dataflow* or partial order on the execution of the commands.

The above command execution model imposes minimal requirements on resource managers. This allows our system to coordinate heterogeneous and variously-sourced resources without requiring unnecessary standardization on the implementation of resource managers. The command execution proceeds in a dataflow or frontier-by-frontier manner. Within a frontier, commands do not depend on one another, and thus can proceed concurrently. A preceding frontier comprises of commands whose execution results are needed for the succeeding frontier. For bringing up resources, the frontiers are arranged bottom up, from the leaves to root(s), while for bringing down resources, the order is reversed. For example, in shutting down the cluster in the example of Sect. 3.3, the first the web server has to be brought down. The next frontier comprises of the two databases and either can be brought down before the other. On the other hand, in bringing up the same cluster, the order of the frontiers is reversed and the web server is the last entity on which an up command gets executed. Note that ordering of the frontiers does not imply synchronized execution. Individual commands in a frontier are issued as soon as the corresponding commands in the preceding frontiers are executed successfully. Although commands across frontiers are not pipelined, no artificial serialization is introduced either. The system remains as asynchronous and concurrent as it can within the bounds of the commands model described above.

Realizable Decision Making. An infinitely-fast or zero-time computation engine is not realizable. Since the optimization decisions involve solution of NP-hard problems [9], even an attempt at approximating zero time, or say hard real time, for solving the optimization problem is not possible. The approach we follow embraces global heuristic solutions that can be arrived at in soft real time. The computationally intensive nature of the decision making component predisposes us towards persisting with a previously derived global solution even when there are a limited number of command failures. It is not computationally-efficient to chart a totally new global course every time there is a command failure. So for example, when a resource refuses to go ONLINE, Mounties looks for an auxiliary solution from within the proposed solution that can substitute for the failed resource. For example, a lightly-loaded resource can (and does) replace a failed resource in case the two belong to the same equivalency. Auxiliary solutions are local in nature. If the finally deployed solution turns out to have too many auxiliary solutions, then the quality of the solution is expected to suffer. To avoid the configuration to deviate too far from the globally optimal solution, Mounties recomputes a global solution whenever the objective value of the deployed solution is below a certain value as compared to the proposed solution. This is done by feeding back an artificially-generated event that forces recomputing the global solution. In summary, Mounties does not attempt to maintain a globally-optimal cluster configuration at all times. Instead, Mounties looks for global approximations for the same. The obvious tradeoff here is using a suboptimal solution versus keeping one or more cluster services unavailable while the optimal solution is being computed. The tradeoff could be unfavorable for Mounties in a relatively uneventful and simple clusters where resources take relatively long time to execute “go ONLINE” and “go OFFLINE” commands as compared to the time spent in determining optimal solution. For such clusters, it would be of merit to recompute a globally optimal cluster configuration.

Computing a globally optimal solution based on the constraints and the current state of cluster, is a significant function of Mounties. The resulting optimization problem can be cast as an abstract optimization problem that can be solved using many well known techniques such as combinatorial optimization methods, mathematical programming and genetic/evolutionary methods. For that reason and to bring modularity to the design, in Mounties, we treat that as a separate module and is called, the Global Optimizer or simply, the Optimizer. It is designed with a purely functional interface to the rest of the system. A detailed discussion of the Optimizer is beyond the purview of this paper and is discussed elsewhere [9,10]. The interface to the Optimizer module completely isolates it from effects of concurrent cluster events on its input. A *snapshot* of the current cluster-state, which incorporates all events that have been recorded till the time of the snapshot, is created and handed over to the Optimizer. The metaphor *snapshot* is meaningful since once taken, the snapshot does not change even if new events occur in the cluster. The snapshot is thus referentially transparent, i.e., purely functional and non-imperative, and references to a particular snapshot return the same data time after time. Given a snapshot, the Optimizer proceeds with its work of proposing an approximately optimal cluster configuration that takes into account the current context and the long-term objectives defined for the cluster.

Just as the Optimizer is not invoked whenever a new cluster event arrives, it may not be interrupted if a new event arrives while it is computing a new global solution. This is primarily to maintain simplicity in the design and implementation. Thus, when the Optimizer returns a solution, the state of the cluster, as perceived by Mounties, may not be the same as the state at the time the optimizer is invoked and that the results produced may be stale. Our system however does try to make up for exclusion of newer events by aligning the solutions proposed by the optimizer with any events that may have arrived during the time the solutions were being created. Such an alignment however, is local in nature. Over longer time intervals, the effects of newer events get reflected in the global solutions computed subsequently.

Because of the nature of the problem, simple rule-based heuristics can be used to make local optimization decisions prior to invoking the Optimizer. Such preprocessing can significantly reduce the turnaround time in responding to events. The preprocessing step is also necessary for isolating the Optimizer from the on going changes in the system. This is referred to as the Preprocessor. Specifically, the Preprocessor waits on a queue of incoming events and then processes an eligible event all by itself or hands down a preprocessed version of the problem to the Optimizer. The decisions from the Optimizer or the Preprocessor are directed to a module called the Postprocessor, which is the center of the command generation and execution machinery. Figure 4 shows the interactions among the Preprocessor, the Optimizer, the Postprocessor, and other modules. These modules are discussed in detail next.

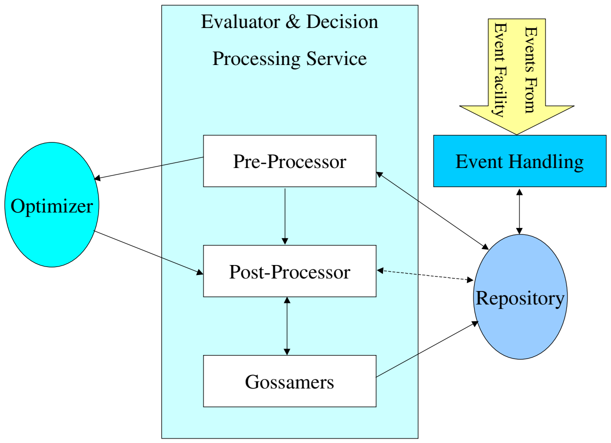

Fig. 4. Mounties Central: internal design

5 Main Services

5.1 The Resource Repository

The Repository of resource objects provides a local, somewhat minimal, and abstract representation of the cluster. The repository cache is coherent with the actual cluster to the extent that cluster events are successfully generated and reported to Mounties. Mounties does safe/conservative cluster management without any assumptions of: (a) completeness of the set of events received by it; (b) correctness of any of the events received by it; and (c) (firm) significance of the temporal ordering of the events received by it. Generally, the effectiveness and efficiency of management depends upon the completeness, correctness, and speed with which events are reported to Mounties, but Mounties does not become unsafe even if event reporting degrades. Within the above event-reporting context, Mounties does assume ownership of the management process, so resources are not expected to configure themselves independently of Mounties. If the context requires say human intervention and direct configuration of resources, then either this can be routed through Mounties, or the semantics of the events reported to Mounties modified so that Mounties remains conservative in its actions.

Regardless of its current state, the repository is updated with an event before the preprocessor is informed. The updating of the repository is an atomic act: readers of the repository either see the update fully, or not at all. The repository is partitioned, and individual resource objects can be accessed individually, so the synchronization requirements of such updating are limited. Partitioning of the repository serves many purposes, including permitting higher concurrent access and better memory use and reduced traversal and searching costs.

Resource objects in the repository contain only a few fields representing necessary information such as current status, desired status, and the current supports of the resource, etc. Snapshot related information (e.g., a time-stamp when the last snapshot was taken and is the object now ready for another snapshot) as well information on the planned actions to be taken are also stored in the resource objects. Since the repository is read and modified concurrently, it is mandatory to reason about all possible combinations of concurrent actions that can take place in the repository so that no erroneous combination slips through. This is carried out by (a) restricting the concurrent access and modifications to only a small set of states in the resource objects, and (b) establishing/identifying invariants and other useful properties of these fields such as monotonicity. For example, we know that cluster events can only change the state of a resource from on-line to off-line or failed and not from failed to on-line since the change to on-line from any state requires a Mounties command.

5.2 The Evaluator and Decision Processing Mechanisms

The Preprocessor. As shown in Fig. 4, events arrive from the cluster and are recorded in the repository module. If an event needs attention by the Preprocessor, then the event is also placed in the input queue of the Preprocessor after it has been recorded in the repository. When there are one or more events in its input queue, the Preprocessor creates a snapshot of the relevant cluster-state by identifying and making a copy of the affected part of the repository. While the repository is constantly updated by new events, the snapshot remains unaffected. Any further processing, in response to the event, takes place using the information encapsulated in the snapshot. Note that the snapshot may capture some of the events that are yet to show up in the Preprocessor queue. Since the repository is more up-to-date, the Preprocessor treats the snapshot as representative of all the events received so far. Note also that because of the atomic nature of the updates to the repository, a snapshot captures an atomic event entirely, or leaves it out completely. For identifying the part of the repository affected by an event, the Preprocessor partitions the cluster resources into disjoint components, called islands, by using the constraint graphs formed by the resource dependencies and collocation constraints. Clearly, an event cannot directly, or indirectly affect resources outside its own island. Such partitioning also serves the purpose as an optimization step prior to applying the global optimization step, by creating multiple smaller size problems, which are less expensive to solve. This is especially beneficial at cluster startup time, when each island can be processed as a small cluster.

Preprocessing includes many more activities: excluding ineligible events (an event can be ineligible for reasons like Mounties is busy with processing a previous snapshot comprising the event's related resources, and thus processing the same resources in another snapshot may lead to divergent action plans which cannot be reconciled); clubbing multiple events (in conjunction with the repository's predisposition) into a larger event; optimizing the snapshot associated with one or more events so that either the event can be handled directly by the Preprocessor, or can be posed as an optimization problem to the Optimizer. A somewhat advanced, but optional treatment of the Preprocessor is to partially evaluate an event using a basic set of rules so as to reduce the amount of processing done by the Optimizer. In general, this can lead to globally non-optimal solutions, but in many instances simple rules can be constructed and embedded in the Preprocessor so as to keep the solutions globally optimal while reducing the load on the Optimizer.

5.3 The Postprocessor

Using the cluster status contained in a snapshot, a new cluster configuration is created by either the preprocessor alone, or by the preprocessor and the optimizer jointly . The configuration primarily indicates the supporting resources to be used in on-lining the resources in the snapshot. The solution is in the form of a graph, outlining the choices to be made in bringing up the resources in the snapshot. Note that, in the cluster, some of these resources may be yet to be configured; some other resources may already be configured and up, as desired by the solution, while the remaining resources may be configured differently and may require alterations. The postprocessor takes this into account and partitions this solution graph into one or more disjoint components that are then handled by simple finite-automaton like machines called the *up-* and *down-gossamers*. Commands within a disjoint region are executed in a pipelined or concurrent manner, as discussed earlier. Across disjoint regions these can be carried out concurrently.

When the Postprocessor picks up a solution to translate into commands and control machinery (one or more gossamers), the Postprocessor notes into the repository the availability of the resources comprising the solution for new analysis. This makes events related to these resources eligible for preprocessing (see above). For Mounties Central supported by a single-processor node, a convenient task size for the Postprocessor is from picking up a solution to the creation of gossamers related to the solution. The Postprocessor can make auxiliary solutions available to a gossamer as the following. If a resource cannot come up because of a failure of one or more issued commands and a suitable alternative resource exists (with spare capacity to support another dependent resource) then that alternative is treated as an auxiliary solution.

The Gossamers. Each gossamer is a simple finite-automaton like machine, which is responsible for changing the state of its set of resources to ONLINE or OFFLINE and follows the dataflow order. Simultaneous execution by multiple gossamers brings a high-degree of concurrency to the execution process. The simplicity in their design allows these entities to be spawned just like auxiliary devices while the more interesting and "thinking" work is kept within the other modules (e.g., the Postprocessor). A gossamer executes its commands by "wiring up" the relevant part of the repository with the solution-set assigned to it. Mounties attempts to bring down a resource only after it has confirmed that all resources dependent on such a resource are currently down. A "go ONLINE" command for a resource is dispatched only after receiving positive acknowledgements for all the supporting resources, and checking that the supporting resources have enough capacity for the upcoming resource (i.e. all necessary resource downs have occurred). This naturally leads to the execution of the commands in a dataflow manner.

The process of on-lining and off-lining of resources in unrelated parts of a solution can proceed simultaneously in a distributed manner. If a resource fails to come up after being asked to do so, the related gossamer asks (the Postprocessor) for auxiliary solutions for the same resource in trying to bring dependent resources of the same up, upon their individual turns.

5.4 Other Services

The Event Notification and Event Handler Mechanisms. Mounties Central and Mounties Agents are associated with a component of the Event Handler. We use Java RMI layer as the event notification mechanism. The central handler gets requests from the agents, which are serialized automatically by Java RMI and communicates back with the agents, again using Java RMI. Because we use the standard services provided by Java RMI, we do not describe those in detail here. We note here that the more reliable event notification mechanisms can replace the RMI-based event notification layer, in a straightforward manner. All resource managers in the cluster, various Mounties agents, and Mounties Central, as well as Mounties GUI all are glued together by the event notification mechanism. We describe the GUI component in some details here.

Mounties GUI. The GUI displays various graphical views of the cluster to the end user, in response to the submitted queries and commands. These requests are routed through the Event Notification mechanism. Java's EventDespatcher thread writes the request in the form of an event in an input queue of the EventHandler. The EventHandler then requests for the required data from Mounties Central. When the necessary information is received, the EventHandler communicates the same to the Mounties agent that local to the node where the initial request came from. The actual rendering is then done by the GUI. The two-way communication between the local Mounties agent and the Mounties Central is done over a layer of Java RMI. Using the GUI, the user can view many of the important characteristics of the resources being managed.

6 Structuring Mounties Implementation

Implementation of Mounties architecture and design imposes a challenging requirement for the software developer–the challenge being how to ensure that the software developed is correct, robust, extensible, maintainable, and efficient enough to meet soft real-time constraints. In this section, we describe a programming paradigm that is well suited to meet these requirements.

A concurrent specification is naturally suited to Mounties and is more likely to yield a verifiably correct and robust implementation of the system. A simple and concurrent implementation of Mounties would comprise of a CSP-style process [5] for each functional block described earlier. Each such process would then communicate with other processes via communication channels, and the entire operation would then proceed in a pipelined manner. Such a specification however can suffer from two problems: (a) complexities associated with managing parallelism including state sharing and synchronization, and (b) inefficiency of fine-grained parallelism. Both of these problems can be addressed by using a different approach than the CSP approach, as described in the following. The approach described here enables a variable-concurrency specification of Mounties and is consistent with the overall operational semantics of Mounties described previously. The paradigm also provides a few additional benefits such as: efficiency and ease in performance tuning; simple extensions to simulate events using cloned copies of the repository; flexibility and amenability to changes in functionality (e.g., adding more Preprocessor smarts).

6.1 Efficient and Flexible Concurrent Programming

The paradigm comprises of an approach of defining relatively short lived, dynamic, concurrent tasks wherein the tasks can be in-lined. In the limit of this approach, all of the tasks can be in-lined, resulting in a sequential implementation of the system. The key issue in this approach is not to compromise on the natural concurrency in the description of the system while defining the dynamic, concurrent tasks, and task in-lining.

In this paradigm, computations are broken into a set of atomic tasks. Tasks are defined such that (a) each task is computationally significant as compared to the bookkeeping costs of managing parallelism; and (b) each task forms a natural unit of computation so that its specification is natural and straightforward. In initial prototyping, (b) can overrule (a), so that correctness considerations of initial work can override performance considerations. Each atomic computation described in a detailed Mounties semantics has to be contained in a task from this set of atomic tasks. Although this is an optimization and not a requirement, for reducing context-switching costs, the computation of a task should proceed with thread-preemption/task-preemption disabled.

Under this paradigm, the operations within Mounties can proceed as follows. Each event from the event handler results in the creation of one or more tasks, to be picked by the one or more threads implementing Mounties. The tasks wait in an appropriate queue prior to being picked. In processing a task, the thread/processor will compute it to completion, without switching to another task. The task execution can result in one or more new tasks getting created, which the thread will compute as and when it gets around to dealing with them. So for example, say an event arises, that creates a Preprocessor-task. The Preprocessor-task can end up creating an Optimizer-task, and a Postprocessor-task. The Postprocessor-task can create gossamer-related tasks, and so on. Allowing for performance tuning and also for later extensions, it may be desirable for the Preprocessor to inline the Postprocessor task within itself and to create the gossamer-related tasks directly, which can be done straightforwardly in this paradigm since tasks are explicit and not tied to the executing threads.

In this programming paradigm, computation and communication are merged. Generally a task is a procedure call, with its arguments representing the communicated, inter-process, channel data from the CSP model. In general inter-module communication is carried out by task queues connecting the modules, wherein, the scheduler is given the charge of executing a task for a module by causing a thread to pick it up from the module's incoming queue. Since in this paradigm, just one thread can implement all the modules, it becomes possible to continue thinking in terms of a purely sequential computation, and to avoid concurrency complexity such as synchronization and locks. If this sequential exercise using this paradigm is carried out in consistence with the Mounties choreography described earlier, then a straightforward extension of the work to multi-threaded implementation with thread safety is guaranteed. The accompanying complexity of lock management and synchronization is straightforward.

7 Related Work

The Mounties system described here is of relevance to both the commercial state-of-the-art products as well as to academic research in this area. First we describe and compare the Mounties System with three important systems that can be considered as the state-of-the-art: IBM's HA/CMP, Microsoft's MSCS, Tivoli's AMS system, and Sun's Jini technology.

Application management middleware has traditionally been used for products that provide high availability such as IBM's HA/CMP and Microsoft's Cluster Services (MSCS). HA/CMP's application management requires cluster resource configuration. Custom recovery scripts that are programmed separately for each cluster installation are needed. Making changes to the recovery scheme or to basic set of resource in the cluster requires these scripts to be re-programmed. Finally, HA/CMP recovery programs are stored and executed synchronously on all nodes of the cluster. MSCS provides a GUI-driven application manager across a two-node cluster with a single shared resource: a shared disk [11]. These two nodes are configured as a primary node and a backup node; the backup node is used normally pure backup node and no service-oriented processing is performed on it. Configuration and resource management is simplified with MSCS: there is only one resource to manage with limited management capabilities.

Tivoli offers an Application Management Specification (AMS) mechanism, which provides an ability to define and configure applications using the Tivoli Application Response Measurement (ARM) API layer [12]. These applications are referred to as instrumented applications. The information gathered from the instrumented applications can be used to drive scripts by channeling the information through the Tivoli Event Console (TEC). The TEC can be configured to respond to specific application notification and initiate subsequent actions upon application feedback. The current version of ARM application monitoring is from a single system's perspective. Future versions may include correlating events among multiple systems.

Over the last few years several new efforts towards coordinating and managing services provided by heterogeneous set of resources in dynamically changing environments. The examples of these include TSpaces [14] and the Jini Technology [3]. The TSpaces technology provides messaging and database style repository services that can be used by other higher level services to manage and coordinate resources in a distributed environment. Jini, on the other hand is a collection of services for dynamically acquiring and relinquishing services of other resources, for notifying availability of services, and for providing a uniform means for interacting among a heterogeneous set of resources. Both TSpaces and Jini technologies are complimentary to Mounties in the sense that they both lack any systematic decision making and decision execution component. However, the services provided by the Repository and Event Notification mechanisms in Mounties do overlap in functionality with the similar services provided in TSpaces and Jini. Finally, there are several resource management systems for distributed environments with decision-making capabilities. Darwin is an example of such a system that performs resource allocations taking into account application requirements [1]. Although there are similarities between Darwin and Mounties, Mounties provides a much richer set of abstractions for expressing complex dependency information among resources. Also, the Mounties system is geared towards optimizing the allocation of services such that overall objectives are met; in Darwin the goal seems to be more geared towards optimizing the requirements of an application or of a service.

The Mounties services described here have some similarities with the *Workflow management systems* that are typically used in automating and coordinating business processes such as customer order processing, product support, etc. As in Mounties, workflow systems also involve coordination and monitoring of multiple tasks that interact with one another in a complex manner [4]. Thus, the task and data choreography can have similar implementation features. However, workflow systems typically do not involve any type of global decision making component, much less solution of an optimization problem resulting in commands for the components of the system.

At the implementation level, Mounties software structuring approach or programming paradigm provides a contrast with approaches such as CSP [5], and Linda [2,13]. Briefly, in comparison to CSP, instead of defining static, concurrent tasks, our paradigm works with relatively short lived, dynamic, atomic tasks that can be inlined. Since tasks in our approach are delinked from threads, our approach has the advantage of allowing greater flexibility and control in software development including variable and controlled concurrency, and a finer level of control over task priority and data priority. In contrast to CSP, the Linda approach and futures [8] provide a handle on dynamic threads, [8] provides a method of dynamic thread in-lining, and Linda in particular provides a handle on a coordination structure, a tuplespace, that can straightforwardly emulate and provide the equivalent of CSP channels for data communication. Our paradigm is different from all these programming language approaches in that it is an informal framework wherein implementation issues/idioms relevant to Mounties-like systems find a convenient, and top-down expression, beyond what these generic language approaches with their compiler/run-time support provide. We leave a formalization of our paradigm as a language/framework for say building domain-specific compilers as an exercise for the future.

8 Conclusions

In this paper, we have described the Mounties system that is designed to support a diverse set of objectives including support for global cluster startup, resource failure and recovery, guarantees for quality-of-service, load-balance, application farm management, plug-and-configure style of management for the cluster resources, and so on. The system itself is composed of multiple services and we describe the design of the key services. The services described here are designed to be general purpose and scalable. This modularity allows for substitution, at run-time, by alternate services including alternate decision making components. Moreover, the system is flexible enough to operate in a full auto pilot mode or a human operator can control it partially or fully. The three services described here (the repository services, the evaluation and execution services, and the event notification services) are adaptable to changes in the system. New resources, constraints, and even new rules or policies can be defined and the system adjusts the cluster-state around these changes. In that sense, these services are active and dynamic components of the middleware. A fourth component of the system, the Optimizer, is also capable of adjusting to such changes in the system. The Optimizer, which is not described here, will be a topic of a separate publication.

Finally, we note here that the decision making capabilities and associated support services are general enough to be applied in other scenarios including in environments that are much more loosely coupled than clusters and that are highly distributed such those encountered in mobile and pervasive computing environments. In such environments, multiple independent decision support systems can co-exist in a cooperative and/or hierarchical manner. This is an area we intend to explore in the future.

Acknowledgements

Many individuals have contributed to the concepts that lead to the Mounties system as described in this paper. In particular the authors would like to thank Peter Badovinatz, Tushar Chandra, and John Pershing, Jr. for many insightful discussions. Many thanks to Rob Strom for his help in improving the style of the paper.

References

1. P. Chandra, A. Fisher, C. Kosak, E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, *Darwin: Customizable Resource Management for Value-Added Network Services,* Proceedings of 6th International Conference on Network Protocols, pp. 177–188, Oct. 1998. 369
2. N. Carriero, and D. Gelernter, *Linda in Context,* Communications of the ACM, vol. 32, pp. 444–458, April 1989. 369
3. K. Edwards, Core JINI, The Sun Microsystems Press Java Series, 1999. 369
4. J. Halliday, S. Shrivastava, and S. Wheater, *Implementing Support for Work Activity Coordination within a Distributed Workflow System,* Proceedings of 3rd IEEE/OMG International Enterprise Distributed Object Computing Conference, pp. 116–123, September, 1999. 369
5. C. Hoare, *Communicating Sequential Processes,* Prentice Hall International (U.K.) Ltd., 1985. 367, 369
6. IBM Corp., RS/6000 SP High Availability Infrastructure, IBM Publication SG24–4838, 1996. 359
7. IBM Corp., RS/6000 SP Monitoring: Keeping It Alive, IBM Publication SG24–4873, 1997. 359
8. D. Kranz, R. Halstead, and E. Mohr, *Mul-T: A High Performance Parallel Lisp,* Proceedings of the ACM Symposium on Programming Language Design and Implementation, pages 81–91, June 1989. 369
9. K. Krishna and V. Naik, *Application of Evolutionary Algorithms in Controlling Semi-autonomous Mission-Critical Distributed Systems* Proceedings of the Workshop on Frontiers in Evolutionary Algorithms (FEA200), Feb, 2000. 361, 362
10. V. Kumar and V. Naik, *Modeling the Global Optimization Problem in Highly Available Cluster Environments* Submitted for publication, 2000. 362
11. M. Sportack, Windows NT Clustering BluePrints, SAMS Publishing, Indianapolis, IN 46290, 1997. 368
12. Tivoli Corp., Tivoli and Application Management, http://www.tivoli.com/products/documents/whitepapers/body_map_wp.html, 1999. 368
13. P. Varma, Compile-time analyses and run-time support for a higher order, distributed data-structures based, parallel language, University Microfilms International, Ann Arbor, Michigan, 1995. 369
14. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford, *T Spaces*, IBM Systems Journal, pp. 454–474, vol. 37, 1998. 369