Agent-Based Resource Discovery

Kyungkoo Jun, Ladislau Bölöni, Krzysztof Palacz, and Dan C. Marinescu
Computer Sciences Department,
Purdue University
West Lafayette, In, 47907
email: {junkk, boloni, palacz, dcm}@cs.purdue.edu

October 8, 1999

Abstract

In this paper we present a distributed discovery method allowing individual nodes to gather information about resources in a wide-area distributed system made up of autonomous systems linked together by a network technology substrate. We introduce an algorithm and a model for distributed awareness and a framework for dynamic assembly of agents monitoring network resources. Whenever an agent needs detailed information about individual components of another system it uses the information gathered by the distributed awareness mechanism to identify the target system, then creates a description of a monitoring agent capable to provide the information about remote resources, and sends this description to the remote site. There an agent factory assembles dynamically the monitoring agent. This solution is scalable and suitable for heterogeneous environments where the architecture and the hardware resources of individual nodes differ, the services provided by the system are diverse, the bandwidth and the latency of the communication links cover a broad range.

Contents

1 Introduction

In this paper we address the problem of resource discovery in a wide-area distributed system made up of autonomous systems linked together by a network technology substrate. The system is heterogeneous, the architecture and the hardware resources of individual nodes differ, the services provided by the system are diverse, the bandwidth and the latency of the communication links cover a broad range.

Individual nodes in such a distributed system may cooperate to accomplish tasks that require resources above and beyond those available in any single node, clients and servers may need to negotiate the quality of service, system administrators may wish to gather synthetic data regarding resource utilization to identify bottlenecks. A data intensive problem may generate a request to assemble dynamically a cluster of workstations with a compound CPU rate, memory, and secondary storage space determined by the problem size. A system administrator may wish to determine the overall secondary storage utilization in a virtual Intranet.

Resource management in a distributed system can be delegated to a subset of nodes providing site-coordination, negotiation, resource monitoring, and other services. For example the Open Data Network, ODN, model [10] is based upon an hourglass architecture with four layers: applications, middleware services, transport services, and bearer services provided by LANs, wireless networks, ATMs, satellite networks and so on. The architecture is conceived to support services ranging from teleconferencing to financial services, from remote login to interactive education. In turn middleware services cover security, name services, multi-site coordination, file systems and so on, and use transport services for video, audio, text, fax, and other types of data. The diversity of the networking substrate, the heterogeneity and autonomy of the nodes, the variety of services provided by the system make all aspects of resource management in this model rather challenging and motivate the desire to search for solutions that are more scalable and able to accommodate rapidly changing heterogeneous environments.

Distributed algorithms for resource management have been known for some time. The flooding algorithm is used by routers in the Internet, broadcasting by local queries, known as "gossiping" [11], [12] have been used to maintain consistency in replicated databases [13] and to gather information about system failures [14].

Autonomous and mobile software agents are widely regarded as necessary components of large-scale distributed systems. Agents can facilitate access to existing services to thin clients, support nomadic computing, perform functions related to resource management, support negotiations among several parties involved in a transaction, reconfigure servers, and so on. For example mobile agents to map network topology were proposed in [2].

Autonomy implies that the agents are active objects with their own tread of control, they can exhibit intelligent behavior. Mobility ensures that the agents can operate in rapidly changing heterogeneous environments. Yet, ensuring code mobility in a heterogeneous environment when the architecture of the nodes is different and we have several operating systems installed is a non-trivial endeavor.

The implicit assumption of agent-based solution for resource discovery in a wide-area system is the existence of a framework for the interoperabilty of different agent families, like the one proposed in [9]. Throughout this paper we assume that a system like the one described in Section 3.2 is installed in every node and the system has an agent factory, an object able to respond to external requests and assemble agents based upon a description of an agent provided by the entity that initiated the request.

In this paper we introduce an agent-based model for resource discovery. Agents running at individual nodes learn about the existence of each other using a mechanism called *distributed awareness*. Each agent maintains information about the other agents it has communicated with over a period of time and exchange periodically this information among themselves. Whenever an agent needs detailed information about individual components of the system we use the information gathered by the distributed awareness mechanism and then assemble dynamically agents capable of reporting the state of remote resources and to negotiate the use of these resources. The remote agent creation and surgery techniques discussed in Section 3.4 are general and allow us to alter drastically the behavior of an agent. For example we can add additional planes for resource negotiations with clients and with the local resource manager, planes to reconfigure a local server and so on.

The contributions of this paper are an algorithm and a model for the distributed awareness and a framework for dynamic assembly of agents capable of providing detailed information about network resources.

The rest of this paper is structured as follows. Section 2 reviews some of the existing algorithms for resource discovery, presents their basic assumptions and relevant performance measures. Then it presents our distributed awareness algorithm and two models for its behavior. Section 3 introduces the agent-based resource discovery architecture and describes an implementation based upon *Bond* [6], a component-based agent framework.

2 Algorithms and Models for Distributed Awareness

A first step in all applications that require some knowledge about the other nodes of a network is to learn about the existence of each other. We call this process "distributed awareness", while other authors [1] refer to it as resource discovery. We believe that in a heterogeneous environment learning about the existence of other nodes is only the first step in a complex process and that resource discovery requires a set of progressively more intricate interactions with the newly discovered object.

2.1 Related work

We review briefly some of the algorithms presented in the literature, their basic assumptions, and the proposed performance measures to evaluate an algorithm. Virtually all algorithms model the distributed system as a directed graph, in which each machine is a node and edges represent the relation "machine A knows about machine B". The network is assumed to be weakly connected and communication occurs in synchronous parallel rounds.

One performance measure is the *running time of the algorithm*, namely the number of rounds required until every machine learns about every other machine. The amount of communication required by the algorithm is measured by: (a) the *pointer communication complexity* defined as the number of pointers exchanged during the course of the algorithm, and (b) the *connection communication complexity* defined by the total number of connections between pairs of entities.

The *flooding* algorithm assumes that each node v only communicates over edges connecting it with a set of initial neighbors, $\Gamma(v)$. In every round node v contacts all its initial neighbors and transmits to them updates, $\Gamma(v)^{updates}$ and then updates its own set of neighbors by merging $\Gamma(v)$ with the set $\{\Gamma(u)^{updates}\}$, with $u \in \Gamma(v)$. The number of rounds required by the flooding algorithm is equal with the diameter of the graph.

The *swamping* algorithm allows a machine to open connections with all their current neighbors not only with the set of initial neighbors. The graph of the network known to one machine converges to a complete graph on $O(log(n))$ steps but the communication complexity increases. Here n is the number of nodes in the network.

In the *random pointer jump* algorithm each node v connects a random neighbor, $u \in \Gamma(v)$ who sends $\Gamma(u)$ to v who in turn merges $\Gamma(v)$ with $\Gamma(u)$. A version of the algorithm called *the random pointer jump with back edge* requires u to send back to v a pointer to all its neighbors. There are even strongly connected graphs that require with high probability $\Omega(n)$ time to converge to a complete graph in the random pointer jump algorithm.

The *Name-Dropper* algorithm is proposed in [1]. During each round each machine v transmits $\Gamma(v)$ to one randomly chosen neighbor. A machine u that receives $\Gamma(v)$ merges $\Gamma(v)$ with $\Gamma(u)$. In this algorithm after $O(log^2 n)$ rounds the graph evolves into a complete graph with probability greater than $1 - 1/(n^{O(1)})$.

2.2 Distributed Awareness; Algorithm and Models

2.2.1 A Distributed Awareness Algorithm

Distributed awareness is a mechanism for the nodes of a wide area distributed system to learn about the existence of each other. Each node maintains an *awareness table* and exchanges the information in this table with other nodes. An entry in the awareness table contains: (1) `Node location`, the IP address of the node, (2) `lastHeardFrom`, the time when we last heard from the node, and (3) `lastSync` the time when awareness information was last sent to the node. The awareness information is piggybacked onto regular messages exchanged between two nodes.

Incoming/outgoing message handling and table merging are discussed now. The algorithm to add new or update existing items is:

```
for every incoming message
   find sender, S
   if the local awareness table has an item I with the same node location as S
      set lastHeardFrom of I as current time
```

```
else
   add a new item initialized with S and lastHeardFrom set as current time
if the incoming message has piggybacked awareness information
   execute table merging algorithm
```

The table merging algorithm is:

```
for each awareness item, I, of the piggybacked awareness table
   if the local table has item I_local with the same node location of I
      set lastHeardFrom of I_local with more recent time stamp
      between those of I_local and I
   else
      add I to the local table with lastSync set zero
```

.

The outgoing message handling algorithm appends the local awareness table to the outgoing message:

```
for an outgoing message M_outgoing destined to a node N
look up an item I with node location N in the local table
if lastSync of I reached a specified age,
   add the local table to M_outgoing
   set lastSync of I as current time
send out M_outgoing
```

Notice that `lastSync` is checked to control the interval between sending awareness information and that the awareness table is periodically purged based upon `lastHeardFrom` field.

2.2.2 Deterministic and Non-deterministic Models

Modeling and analysis of the distributed awareness algorithm is rather difficult. The problem is unstructured, in the general case we do not know either the network topology or the communication patterns among nodes thus it is rather difficult to make simplifying assumptions leading to a tractable analysis. Yet we need to get a rough idea of the overhead incurred by this method and the asymptotic properties of the algorithm. For example intuitively we expect that after some time all agents will learn about the existence of all other agents.

To model the distributed awareness we propose to use models similar with the ones for the spread of a contagious disease. An epidemics develops in a population of fixed size consisting of two groups the infected individuals and the uninfected ones. The progress of the epidemics is determined by the interactions between these two categories.

We introduce first a deterministic model. Given a group of n nodes this simple model is based upon the assumption that the rate of change in agent's awareness list, is proportional with the size of the group the agent is already aware of, y, and also with the size of the group the agent is unaware of, $n - y$. If k is a constant we can express this relation as follows:

$$y(t)' = k \times y(t) \times (n - y(t))$$

The solution of this differential equation with the initial condition $y(0) = 0$ is:

$$y(t) = \frac{n}{1 + (n-1)e^{-knt}}$$

This function is plotted in Figure 1 and shows that after time τ a node becomes aware of all the other nodes in the network. The parameter k as well as the value τ can be determined through simulation.

Call η the ratio of the awareness information exchanges to the total number of instances an agent communicates with other agents. A typical value for this parameter is $\eta = 0.001$. If the amount of awareness information is only a fraction b, say $b = 0.1$ of the payload carried out during communication between two agents, it follows that the additional load due to the distributed awareness is only a small fraction, in our example only $\eta \times b = 0.01\%$ of the total network traffic.

This deterministic model allows only a qualitative analysis. Rather than the smooth transition from 0 to n we should expect a series of transitions each one corresponding to a batch of newly

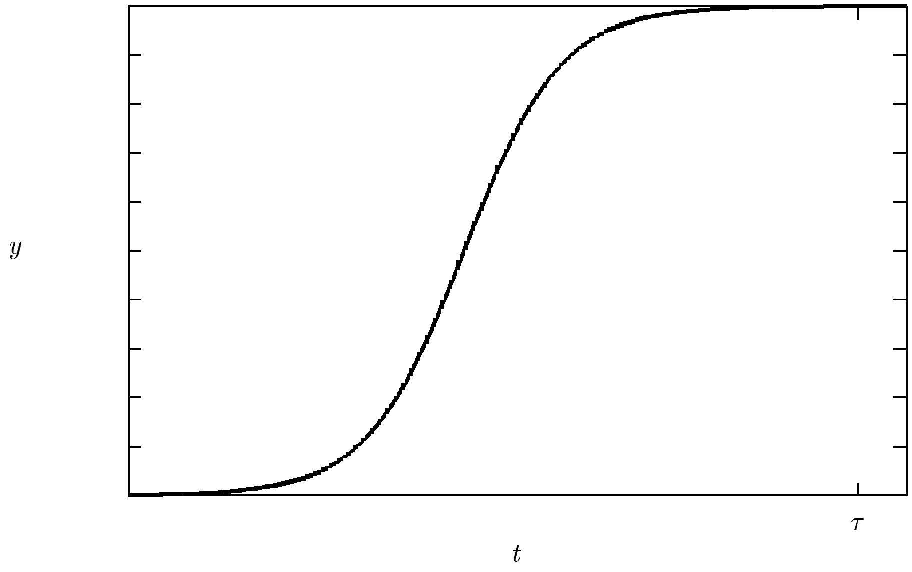

Figure 1: The number of agents known to a given agent, function of time, using a deterministic distributed awareness model. After time τ, each agent becomes aware of all the other agents in the network

discovered agents. Yet this simple model provides some insight into the overhead incurred during the learning phase of the resource discovery mechanism we propose.

A non-deterministic model is sketched below. New acquaintances occur in batches at time intervals determined by the overall rate of information exchange among nodes and by η. Call p the probability of contact between two agents such that as a result of the contact the awareness list are modified, and let $q = 1 - p$. Assume that the contacts between agents are stochastically independent and observe that the probability that among the i entries in the list supplied to a agent $k, \leq i$ entries are not already in its list is

$\binom{i}{k} \times p^k \times q^{i-k}$

Call $Y(s)$ the random variable denoting the number of entries in the list of the "typical" agent at discrete time $s = 1, 2, \ldots$. Then

$P(Y(s+1) = j | Y(s) = i) = \binom{i}{k} \times p^{i-j} \times q^j$

if $i \geq j$ and zero otherwise.

The probability distribution of $Y(s+1)$ is independent of the values assumed by the random variables $Y(r), \quad r \leq s$. Therefore $(Y(s))_{s \geq 0}$ is a Markov chain with states $0, 1, ...n$ and the transition matrix is:

$$
\begin{array}{c c}
 & \begin{array}{ccccc} 0 & 1 & 2 & \ldots & r \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ . \\ . \\ . \\ r \end{array} &
\left[\begin{array}{ccccc}
1 & 0 & 0 & \ldots & 0 \\
p & q & 0 & \ldots & 0 \\
p^2 & 2pq & q^2 & \ldots & 0 \\
. & . & . & \ldots & . \\
. & . & . & \ldots & . \\
. & . & . & \ldots & . \\
p^r & rp^{r-1}q & \binom{r}{2}p^{r-2}q^2 & \ldots & q^r
\end{array} \right]
\end{array}
$$

Once a large system is operational we can attempt to determine the parameters of the model, including the transition probabilities, and then validate the model. The large number of parameters make this model very cumbersome for analysis of a realistic system, with a large number of nodes. The model is useful for theoretical studies, assuming different communication patterns, but this is beyond the scope of this paper.

3 Monitoring Agents and Resource Discovery

3.1 Gathering Resource Information in a Wide-Area Distributed System

Information about the resources and the state of the nodes of a wide area distributed system is sometimes needed to coordinate the activity of a group of nodes, to provide synthetic information about resource utilization, or for other needs. A common approach taken by commercial as well as research systems is to install on each node a monitor to gather local resource information. The local monitors may update periodically a centrally stored database or provide the information on demand. Sometimes the information may be stored on a hierarchy of servers.

Several metacomputing projects [3], [5] rely on a group of central entities to maintain the resource information reported to them by local entities. Globus [3] provides a *Metacomputing Directory Service* where network resource information is stored in a tree-like structure and it is accessible using the Lightweight Directory Access Protocol [4]. Local monitors residing on each node report the structure and state of resources. Monitors have to be installed and configured for each site. Legion [5] uses *collections* as repositories for information describing the state of the resources comprising the system. The collection is a database of static information reported by local monitors on remote nodes. Resource management software provided by several companies including Tivoli [15] follow the same paradigm.

The information provided by a local monitor is determined at the time the monitoring program is installed. To provide additional information the program must be modified and reinstalled. The monitoring program must as non-intrusive as possible thus very rarely a monitor is configured to supply data seldom needed. Often the information obtained from static databases is obsolete. These considerations justify the need to investigate alternative means for gathering resource information.

Using software agents for resource discovery and monitoring has several advantages over the more traditional approach outlined above. Monitoring agents have an autonomous behavior and evolve based upon the characteristics of the local system and the requirements of the beneficiary agent. Agents can engage in a gradual discovery process and respond to a changing set of requirements. They are able to adapt to the architecture and the operating environment of the local node. An agent may change its behavior based upon the results of an inference process and the tasks assigned to an agent can be rather complex. On the other hand, the amount of resources used by the agents may be larger than resources required by a custom-made monitoring system

Now we describe an agent-based, distributed resource discovery architecture where agents are created at remote locations and modified as needed, to gather the information needed for resource management.

3.2 Bond; a Distributed Object System

Bond is a Java-based distributed object system and agent framework, with an emphasis on flexibility and performance. It is composed of (a) a core containing the object model and message oriented middleware, (b) a service layer containing distributed services like directory and persistent storage services, and (c) the agent framework, providing the basic tools for creating autonomous network agents together with a database of commonly used strategies which allow developers to assemble agents with no or minimal amount of programming.

Bond Core. At the heart of the Bond system there is a Java Bean-compatible component architecture. Bond objects extend Java Beans by allowing users to attach new properties to the object during runtime, and offer a uniform API for accessing regular fields, dynamic properties and JavaBeans style `setField/getField`-defined virtual fields. This allows programmers the same flexibility like languages like Lisp or Scheme, while maintaining the familiar Java programming syntax.

Bond objects are network objects by default: they can be both senders and receivers of messages. No post-processing of the object code as in RMI or CORBA-like stub generation, is needed. Bond uses *message passing* while RMI or CORBA-based component architectures use *remote method invocation*.

The system is largely independent from the message transport mechanism and several communication engines can be used interchangeably. We currently provide TCP-based, UDP-based, Infospheres-based, and, separately, a multicast engine. Other communication engines will be implemented as needed. The API of the communication engine allows Bond objects to use any communication engines without the need to change or recompile the code. On the other hand, the properties of the communication engine are reflected in the properties of the implemented application as a whole. For example the UDP based engine offers higher performance but does not guarantee reliable delivery.

All Bond objects communicate using an agent communication language, KQML [8]. Bond defines the concept of *subprotocols*, highly specialized, closed set of commands. Subprotocols generally contain the messages needed to perform a specific task. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol. An alternative formulation would be that subprotocols introduce a *structure in the semantic space of the messages.*

Subprotocols group the same functionality of messages which in a remote method invocation system would be grouped in an *interface*. But the larger flexibility of the messaging system allows for several new techniques which are difficult to implement in the remote method call case:

- The subprotocols implemented by objects are properties of the object, so two objects can use the property access subprotocol implemented by every Bond object, to find the common set of subprotocols they can use to communicate.

- An object is able to control the path of a message and to delegate the processing of the message to subcomponents called *regular probes*. Regular probes can be attached dynamically to an object as needed.

- Messages can be intercepted before they are delivered to the object, thus providing a convenient way to implement security by means of a fire wall, accounting, logging, monitoring, filtering or preprocessing messages. These operations are performed by subcomponents called *preemptive probes* which are activated before the object in the message delivery chain.

- Subprotocols, like interfaces, are grouping some functionality of the object, which may or may not be used during its lifetime. A subcomponent called *autoprobe* allows the object to instantiate a new probe, to handle an incoming message which can not be understood by the existing subcomponents attached to an object.

- Objects can be addressed by their unique identifier, or by their *alias*. Aliases specify the services provided by the object or its probes. An object can have multiple aliases and multiple objects can be registered under the same alias. The latter enables the architecture to support *load balancing* services.

These techniques can be implemented through different means in languages which treat methods as messages, e.g. Smalltalk. In Java and C++ they can be implemented at compile time, not at runtime, e.g. using the delegation design pattern. Techniques from the recent CORBA specifications e.g. the simultaneous use of DII, POA, trading service and others, also allow to implement a similar functionality, but with a larger overhead, and significantly more complex code.

Bond Services. Bond provides a number of services commonly used found in distributed object systems, like directory, persistent storage, monitoring and security. Event, notification, and messaging services, which provide message passing services in remote method invocation based systems are not needed in Bond, due to the message-oriented architecture of the system.

Some of Bond services perform differently than their counterparts in other middleware systems, like CORBA. For example, Bond never requires explicit registration of a new object with a service. Finding out the properties of a remote object, i.e. the set of subprotocols implemented by the object, is done by direct negotiation amongst the objects. The directory service in Bond combines the functionality of the naming and trading services of other systems and it is implemented in a distributed fashion. Objects are located by a search process which propagates from local directory to local directory. The directories are linked into a virtual network by a transparent *distributed awareness* mechanism, which transfers directory information by piggybacking on existing messages as discussed in the previous Section.

Compared with the naming service implementations in systems like CORBA or RMI, which are based on the existence of a name server, this approach has the advantage that there is no single point of failure, and the distributed awareness mechanism reconstitutes the network of directories even after catastrophic failures. However, a distributed search can be slower than lookup on a server, especially for large networks of Bond programs. For these cases, Bond objects can be registered to external directories, either to a CORBA naming service through a gateway object, or to external directory services using LDAP access.

3.3 Bond Agents

The *Bond agent framework* is an application of the facilities provided by the Bond core layer to implement collaborative network agents. Agents are assembled dynamically from components in a structure described by a multi-plane state machine, [7]. This structure is described by a specialized language called `blueprint`. The active components (*strategies*) are loaded locally or remotely, or can be specified in interpretive programming languages embedded in the blueprint script. The state information and knowledge base of the agents are collected in a single object called *model of the world* which allows for easy checkpointing and migration of agents. The multiplane state machine describing the behavior of agents can be modified dynamically, thus allowing for *agent surgery*.

The *behavior* of the agent is described by the *actions* the agent is performing. The actions are performed by the strategies either as reactions to external events, or autonomously in order to pursue the *agenda* of the agent. The current state of the multiplane state machine (described by a *state vector*) is specifying the strategies active at a certain moment. The multiple planes are a way of expressing parallelism in Bond agents. A good technique is to use them to express the various facets of the agents behavior: sensing, reasoning, communication/negotiation, acting upon the environment and so on. The *transitions* in the agent are modifying the behavior of the agent by changing the current set of active strategies. The transitions can be triggered by internal events or from external messages - these external messages form the *control subprotocol* of the agent.

Strategies, having limited interface requirements are a good way to provide code reuse. The Bond agent framework provides a strategy database, for the most commonly used tasks, like starting and controlling external agents or legacy applications. A number of base strategies for common tasks like dialog boxes or message handlers are also provided, which can be sub-classed by developers to implement specific functionality. External algorithms, especially if written in Java are usually easily portable to the strategy interface.

3.4 Remote Creation and Surgery of Monitoring Agents

In this section, we discuss the remote creation of an agent and its surgery. To illustrate the concepts outlined in Section 3.3 we present the creation and modification of a monitoring agent. Several entities are involved in this process: the beneficiary agent at the site where the resource information is needed, the agent factory at the target site, and possibly a blueprint repository. The target site is identified using the distributed awareness or by means of a name or directory service. The beneficiary agent obtains first the blueprint for the monitoring agent it wants installed. The blueprint can be gathered from a blueprint repository or may be created dynamically by an agent given a set of rules and facts. The blueprint can be embedded into the message or the location of the blueprint repository and the name of the blueprint may be provided. Figure 2 illustrates this process. A *Bond Resident* is a container object including directory, communicator, and all other objects. In this example the message sent by the beneficiary agent contains the blueprint:

```
(achieve :content assemble-agent :blueprint-program [agent blueprint])
```

The beneficiary agent in this example decides to create a single plane monitoring agent with the blueprint shown in Figure 3. Figure 4 shows the monitoring agent with one plane designed to gather information about the primary storage, e.g. the amount of physical memory available in the node, the amount of used and free storage, a list of the top users of memory, and so on. Recall that each plane describes a state machine.

The agent factory receives the message, interprets the blueprint, and creates a monitoring agent with one plane called `PrimaryStorage` using one strategy included into the blueprint as `JPython` program [16], associated with `MemoryCheck` state. The complete JPython strategy is shown in the appendix A. After creating the agent, the agent factory sends back an acknowledgment to the beneficiary agent.

Once started, the agent performs a transition to the `MemoryCheck` state. The Jpython strategy identifies the operating system running on that node and invokes the system calls, e.g. `vmstat`

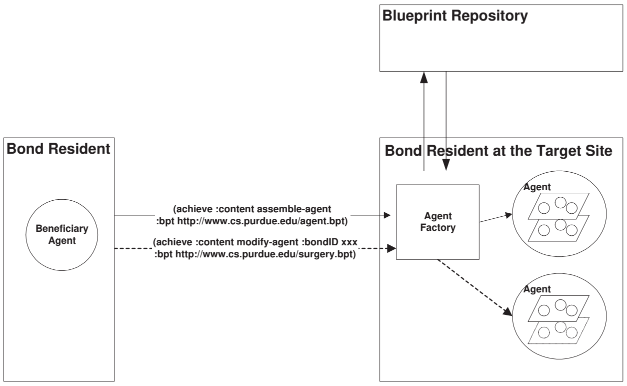

Figure 2: The communication between Beneficiary Agent, the Agent Factory and the Blueprint Repository. Messages instructing the agent factory to create the monitoring agent (solid line) and to perform surgery (dotted line) are shown.

```
create agent MonitoringAgent
plane PrimaryStroage
   add state Init with strategy InitCheck;
   add state MemoryCheck with strategy language python embedded {:
      def getcmdresults(cmd):
      ''''''Run a command and return its output
            as a string and exit value
      ''''''
      .
      .
      def vmstat():
      ''''''Return the statistics from vmstat output in form of a hashtable
      [list, exitcode] = getcmdresults('vmstat 1 2')
      .
      .
      def save(map, prefix = ''):
      ''''''Save a hashtable into model
      .
      .
      save(vmstat(), 'discover.')
      self.fsm.transition(``gotoReport'');
   :}
   add state MemoryReport with strategy ReportPS;
   add state Done with strategy End;

   internal transitions {
      from InitCheck to MemoryCheck on gotoCheck;
      from MemoryCheck to MemoryReport on gotoReport;
      from MemoryReport to Done on gotoDone;
   }
   model {
      BeneficiaryAddress = ``ResourceAgent@peter.cs.purdue.edu:2000''
   }
end plane;
end create.
```

Figure 3: The blueprint of a monitoring agent designed to gather information about available physical memory, the amount of used and free storage, and a list of top memory users

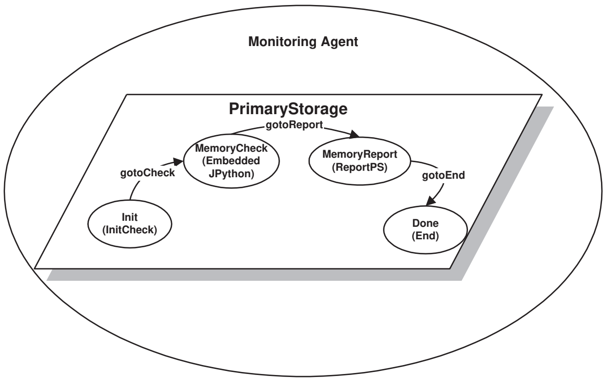

Figure 4: The monitoring agent built using the blueprint in Figure 3. The strategies associated with every state are shown in parenthesis.

in Unix, necessary to gather the information about the primary storage. If successful, the state machines performs a transition to the `MemoryReport` state with strategy `ReportPS`, and sends back the information to the beneficiary agent named in the `BeneficiaryAddress` and finishes its execution by means of a transition to the `Done` state with the `End` strategy.

The primary storage map changes in time, thus it might be desirable to have an agent able to report the information periodically. In addition, it may be necessary to gather information about the secondary storage, e.g. the total amount of disk space available, the amount in use, the free disk space, the number of file systems, etc.

To obtain the periodic memory report and secondary storage information, the agent can be modified through surgery as shown in Figure 5. In our example we (a) add another plane, called `SecondaryStorage`, to report the amount of free secondary storage space, and (b) modify the memory plane by adding transition from `MemoryReport` state to `MemoryCheck` state while deleting `Done` state and `gotoEnd` transition. As a result, the agent reports periodically the state of the primary storage. The reporting interval is specified in the blueprint as `Interval`, in this case, 5000 msec.

To perform the surgery, we send the agent factory at the target site the following message:

```
(achieve :content modify-agent :bondID [agent ID]
:blueprint-program [agent surgery script])
```

The message contains the unique Bond ID of the agent. This allows the agent factory to identify the target of the surgery request. Figure 6 shows the monitoring agent after the surgery of Figure 5. Agent surgery involves the modification of the data structure used to control the scheduling of various strategies in the planes of the agent. The surgery can be performed while the agent is running and the blueprint of the modified agent can be generated.

4 Conclusions

Information about the topology, resources and the state of the nodes of a wide area distributed system is sometimes needed to coordinate the activity of a group of nodes, to provide synthetic information about resource utilization, or for other needs. A common approach taken by commercial as well as research systems is to install on each node a monitor to gather local resource information. The local monitors may update periodically a centrally stored database or provide the information on demand.

Using software agents for resource discovery and monitoring has several advantages over the more traditional approach outlined above. Monitoring agents have an autonomous behavior and evolve based upon the characteristics of the local system and the requirements of the beneficiary agent. Agents can engage in a gradual discovery process and respond to a changing set of requirements. They are able to adapt to the architecture and the operating environment of the local node. An agent may change its behavior based upon the results of an inference process and the tasks assigned

```
modify agent Probing
plane SecondaryStorage
add state Init with strategy InitSS;
add state StorageCheck with strategy MeasureSS;
add state StorageReport with strategy ReportSS;
internal transitions {
from InitSS to StorageCheck on gotoCheck;
from StorageCheck to StorageReport on gotoReport;
from StorageReport to StorageCheck on gotoCheck;
}
end plane;
plane PrimaryStorage
delete state Done;
internal transitions {
delete from MemoryReport to Done on gotoEnd;
from MemoryReport to MeamoryCheck on gotoCheck;
}
model {
Interval = 5000;
}
end plane;
```

Figure 5: The agent surgery script. S second plane, `SecondaryStorage` is added and state machine of the first plane, `PrimaryStorage` is modifyed.

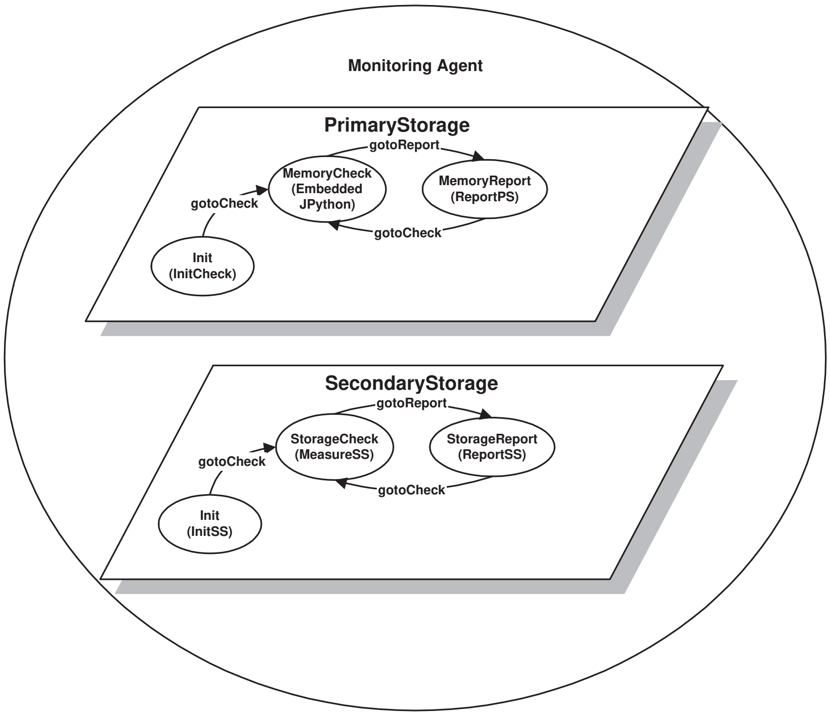

Figure 6: The agent after the surgery of Figure 5

to an agent can be rather complex. On the other hand, the amount of resources used by the agency may be larger than resources required by a custom-made monitoring system.

In this paper we introduce an agent-based model for resource discovery. Agents running at individual nodes learn about the existence of each other using a mechanism called *distributed awareness*. Each agent maintains information about the other agents it has communicated with over a period of time and exchange periodically this information among themselves. Whenever an agent needs detailed information about individual components of the system we use the information gathered by the distributed awareness mechanism and then assemble dynamically agents capable of reporting the state of remote resources and to negotiate the use of these resources. The remote agent creation and surgery techniques are general and allow us to alter drastically the behavior of an agent.

We present two models for distributed awareness, a deterministic model that supports a qualitative analysis and a more intricate, quantitative model. We introduce the Bond system and discuss the assembly and surgery of a monitoring agent capable to report the use of primary and secondary storage.

The Bond systems is available under an open source license from `http://bond.cs.purdue.edu`.

5 Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

6 Appendix: a JPython Strategy to Gather Memory Information

```
add state MemoryCheck with strategy language python embedded
{:

from java.lang import Runtime, StringBuffer
from java.io import InputStream, StringWriter

import string

def getcmdresults(cmd):
    """Run a command and return its output as a string
       also return the exit value as the tuple's second arg
       Runtime.exec() writes some nonsense on standard output
       at least in Linux
    """
    p = Runtime.getRuntime().exec_(cmd)
    p.waitFor()
    output = p.getInputStream()
    buf = StringWriter()
    c = output.read()
    while c != -1:
       buf.write(c)
       c = output.read()
    return (buf.getBuffer().toString(), p.exitValue())

def accustat(param):
    """Accumulate information about users and return a hashtable
       requires System V ps (Solaris 2.x, newest Linux)
       see man page for parameter names, try e.g. pmem
    """
    [list, exitcode] = getcmdresults('ps -eo user,'+ param)
    if exitcode > 0:
       return None
```

```
    broken =  string.split(list, '\n')
    map = {}
    for line in broken[1:]:
       spl =  string.split(line)
       if len(spl) != 2:
        continue
        [user, param] = spl
        if map.has_key(user):
           map[user] = map[user] + string.atof(param)
    else:
           map[user] = string.atof(param)
    return map

def vmstat():
    """ Return the statistics from the vmstat output in form
        of a hashtable. See manual page for the meanings of the keys (system
        dependent although some are common).
    """
    [list, exitcode] = getcmdresults('vmstat 1 2')
    if exitcode > 0:
       return None
    broken =  string.split(list, '\n')
    names = string.split(broken[1])
    values = string.split(broken[3])
    map = {}
    i = 0
    for name in names:
       map[name] = string.atoi(values[i])
       i = i + 1
    return map

def save(map, prefix = ''):
    """ save a hashtable into the model with optional prefix (should
        include the dot)
    """
    for name in map.keys():
        model.set(prefix + name, map[name])
save(vmstat(), 'discover.')
self.fsm.transition("gotoReport")
:};
```

References

[1] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource Discovery in Distributed Networks. In *Proceedings of PODC'99*, pg. 229-237, Atlanta, 1999.

[2] N. Minar, K. Kramer and P. Maes. Cooperating Mobile Agents for Mapping Networks. In *Proceedings of the First Hungarian National Conference on Agent Based Computation*, 1999.

[3] S. Fitzgerald, I. Foster, C. Kesselman, G. Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symp. on High-Performance Distributed Computing*, pg. 365-375, 1997.

[4] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. RFC 1777, 03/28 95. Draft Standard.

[5] S. Chapin, D. Katramatos, J. Karpovich and A. Grimshaw. Resource Management in Legion. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing in*

conjunction with the International Parallel and Distributed Processing Symposium, San Juan, Puerto Rico, April, 1999.

[6] L. Bölöni and D.C. Marinescu *An Object-Oriented Framework for Building Collaborative Network Agents* in *Intelligent Systems and Interfaces*, (A. Kandel, K. Hoffmann, D. Mlynek, and N.H. Teodorescu, eds). Kluewer Publising House, (1999), (in press).

[7] L. Bölöni and D.C. Marinescu *A Multi-Plane State Agent Model*, September 1999, (submitted).

[8] T. Finin, et al. Specification of the KQML Agent-Communication Language, DARPA Knowledge Sharing Initiative draft, June 1993

[9] MASIF - The CORBA Mobile Agent Specification. `http://www.omg.org/cgi-bin/doc?orbos/98-03-09`

[10] L. Kleinrock and John Major. Computing and Communications Unchained: The Virtual World., in *Defining A Decade*, National Research Council, National Academy Press, pp. 36-46, 1999.

[11] Sandra Hedetniemi, Stephen Hedetniemi, and A. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319-349, 1988.

[12] A. Pelc. Fault-tolerant broadcasting and gossiping in communication. *Networks*, 28(3):143-156, October 1996.

[13] D. Agrawal, a. Abbadi, and R. Steinke. Epidemic Algorithms in replicated databases. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161-172, Tucson, Arizona, 12-15 May 1997.

[14] S. Assmann and D. Kleitman. The number of rounds needed to exchange information within a graph. *SIAM Discrete Applied Math*, 6:117-125, 1983.

[15] `http://www.tivoli.com/products/overview`

[16] Jim Hugunin. Python and Java: The Best of Both Worlds. In *Proceedings of of the 6th International Python Conference*, October 14-17, 1997, San Jose, California