Malleable Services

Venu Vasudevan
Networks and Infrastructure Department
Motorola Labs
1301 E. Algonquin Rd.
Mail Stop IL02/2240
Schaumburg, IL 60196
847-576-4182
venuv@rsch.comm.mot.com

Sean Landis
Networks and Infrastructure Department
Motorola Labs
1301 E. Algonquin Rd.
Mail Stop IL02/2240
Schaumburg, IL 60196
847-576-3469
Sean.Landis@motorola.com

Abstract

Deploying distributed services over a complex network topology presents a challenge, one of mapping the objects to locations in an optimal manner. This mapping needs to be dynamic, taking current network conditions into consideration. Remapping services is manual-intensive, requires operator effort and may result in service downtime. The Mojave project described here investigates an architecture for implementing ***malleable*** *(auto-configuring) services using reactive and mobile agents. In contrast to past efforts, Mojave views agents as a wrapper technology implemented over a Jini-tuplespace based architecture. The paper describes the Mojave architecture and implementation, experiences in building an adaptive systems manager application, and benefits of the Mojave architecture for thin-client computing.*

1. Introduction

While distributed object technology has accelerated the implementation of large and complex services, their deployment presents a challenge. Distributed service deployment involves mapping application components to the available machine topology. This might entail placing compute-intensive components on large machines, data-intensive components close to the data sources, and components that perform sensitive tasks on platforms that are less vulnerable to intrusions or attacks. Given the continual change in an application's network environment, the mapping of components to locations and hardware is likely to continually change in response to changing resource conditions.

The current model of operator-mediated service mapping is laborious in terms of effort, and expensive in terms of potential service downtime. The goal in Mojave (MObile Jini [10] Agent enVironmEnt) is to build a component platform for auto-configuring (or *malleable*) services. Services that automatically and proactively configure themselves to changing conditions will experience reduced downtime while also reducing the operator overhead in service reconfiguration.

The approach taken in Mojave to implementing auto-configuration is to enable application components to be *transportable* and *environment-aware*. Components are transportable in that they can pause execution at one location and begin executing at another without loss of state. They are environment-aware in that the component code is augmented with a *mobility certificate*, a non-functional specification of the environment required for the component to operate effectively. The Mojave infrastructure monitors the mobility certificates of application components, and relocates the components whose certificates are violated. In aggregate, the mobility policy specifications of an application's components automate reconfiguration that the operator would otherwise carry out manually.

We refer to Mojave components as *mobile agents*, since they are autonomously mobile from the point of view of the operator. However, our emphasis on mobility management as a wrapper technology for adaptive computing differs from past usage of the term for mobile objects with pre-programmed (internal) itineraries. Table 1 contrasts the use of mobile agents in Mojave from other agent projects.

Table 1: Contrasts in mobile agent usage.

	Mojave	Other
Goal	Application adaptation, survivability	Bandwidth conservation
Agent Type	Reactive	Itinerant
Mobility Policy	External to agent implementation	Internal, part of agent code
Grouping	Peer relationships - part of mobility policy	Opaque

This paper describes the Mojave architecture and implementation, and its use in developing an auto-configuring systems management application. Systems management was chosen as an initial application focus as systems managers tend to be complex, high-priority services. Section 2 and Section 3 describe Mojave architecture and implementation. Section 4 describes the implementation of an adaptive performance management application that adjusts the numbers and location of performance manager objects based on the complexity of the management task. Section 5 surveys related work in mobile agents, related mobile code efforts, and systems management, and Section 6 presents conclusions and future work.

2. Mojave Architecture

The Mojave platform consists of three kinds of entities: *pods*, *agents* and *liaison(s)*. A Mojave application runs over a distributed collection of *pods*, each of which can host multiple *agents*. Pods provide an runtime context and support services for agent execution. They are analogous to Enterprise JavaBean containers in their hosting of executing components. *Liaisons* provide a store-and-forward communication portal for communities of pods and agents. Pods communicate with other pods by posting message tuples to the liaison. Figure 1 shows the high-level interaction among pods, agents and liaisons.

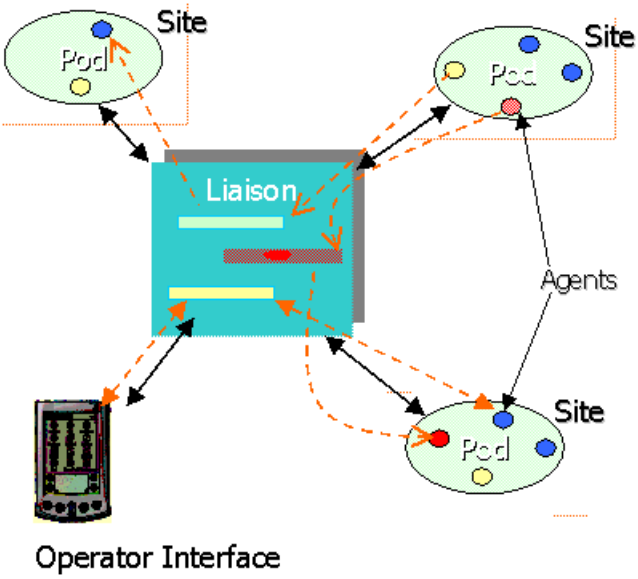

Figure 1 - Mojave application architecture.

Components that are not able to host a pod (such as the PDA in the figure) can still participate in a Mojave environment via interaction with the liaison.

2.1 Agents: Active Components

Agents are the smallest unit of active computation in Mojave. They augment conventional distributed objects by being *reactively mobile*, *environment-aware* and *clonable.* Agents are *reactive* in that they execute in a separate thread of control, and can "program" their mobility. Agents use mobility to move themselves to where they can function most effectively, and to move themselves out of pods where they are not operating effectively.

Agent definitions include a *mobility certificate* that describes the mobility policy suitable for a particular agent. The mobility certificate consists of a set of event-action rules, with the event describing the environmental conditions that cause the agent to move, and the action clause describing where the agent needs to move to. Mojave uses the mobility certificate to manage per-agent mobility. The mobility certificate thus encapsulates the agent's *environment-awareness*. A Mojave application composed of multiple mobile agents continually reorganizes itself as its agents move to best perform their function. The application as a whole becomes malleable in that it reconfigures itself piece by piece as needed to function efficiently in a changing environment. We have chosen an engineering approach to reactive mobility rather than a theoretical one. Therefore it is possible, with poorly specified mobility certificates, for the system to become unstable as agents constantly move to find an optimal configuration. Our initial

goal is to provide the mechanisms to make reactive mobility possible. From there, we can explore issues of stability and emergent behavior.

It is worth noting that a mobility certificate is bound to an agent *instance*, not to the type. Agent instances that are functionally identically may have different survivability and adaptability needs which are a function of the needs of their clients.

In Mojave we use *rooted* agents to model situated services. Rooted agents move once (device driver download) to their host pod and then remain situated. *Actuators* are a subclass of rooted agents that control a physical device such as a lamp or camera. A mobile agent manipulates a physical object by interacting with its actuator. A *sensor* rooted agent represents a telemetry provider. A sensor can represent a physical or logical entity. Thermometers and motion detectors are physical entities, whereas a resource like CPU utilization is a logical entity.

Mojave provides rooted agents to support mobile agent computations for a couple of reasons. Mobile agent applications need support services (e.g. ability to invoke operating system commands) from the agent platform to operate effectively. In addition, mobile agent applications may need hooks into the external world to manipulate actual "things" (e.g. robots, light switches etc.) that are collocated with the pod. In contrast to computational services, physical and support services are "situated". A scanning service may only be available from a pod (machine) to which the scanner is attached. A lamp in your family room may only be controlled from the PC that is within infrared range.

Mojave agents can be *cloned* to subdivide work and distribute tasks. A clone inherits a copy of original agent's state and functionality but gets assigned a unique agent identifier.

Agents can communicate via a distributed event mechanism. Agents use an API to register for events based on a subscription string. This form of messaging supports one to many asynchronous one way communication via the liaison. Future plans include a form of mediated point-to-point communication that is consistent with the Mojave philosophy of name-based addressing.

2.2 Pods: Smart Places

The role of a pod in agent computations can be viewed from one of several perspectives - *place, active container*, and *router*. Pods are located at places (room X in building Y), and this location is significant to incoming agents. The support capabilities of a pod vary based on the rooted agents it contains. From the *place* perspective, pods advertise the capabilities of a place to a trader [11] service, and participate in the matchmaking protocol by which an agent chooses a particular pod to execute in. As *containers*, pods differ from EJB containers in that they are *active* in their management of executing agents. Pods coordinate environment monitoring for conditions that violate the mobility certificates of hosted agents. For instance, if one of the rules of an executing agent is that the CPU load be less than 75%, the pod is responsible for monitoring CPU load (probably via a rooted agent), and alerting the agent to move when the predicate is violated. If the pod is unable to satisfy all its mobility certificates, it may need to jettison some agents so the others can operate effectively.

Pods are message *routers* that isolate hosted agents from the details of inter-agent communication. An agent provides the pod with a message and a destination agent identifier, and the pod handles the mechanics of messaging, and other reliable message delivery issues.

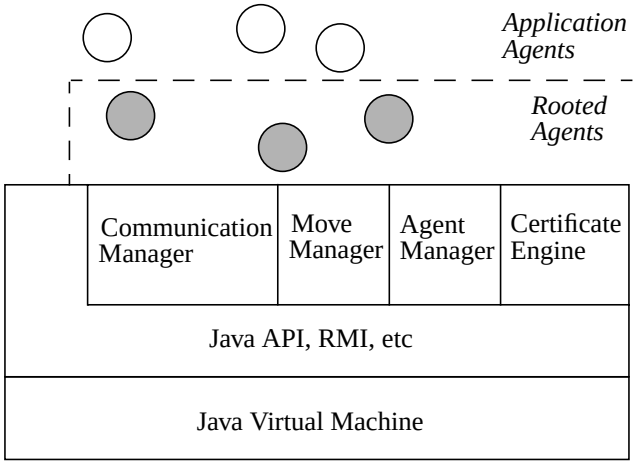

Figure 2 - Pod architecture.

To support these functions, a pod internally resembles a mini operating system, as depicted in Figure 2. Incoming agents hand over their mobility certificates to the pod, which in turn monitors the relevant mobility events using a *certificate engine*. The certificate engine monitors the environment, and notifies the agent of any constraint violation. The engine must interpret the certificates from incoming (or new) agents and determines the environment events that need to be monitored. An agent that decides to reactively migrate is atomically moved to a different suitable pod by the *move manager*. The move manager in the target pod coordinates with the *communication manager* to ensure that messages issued while an agent is moving, are delivered in order when the agent has completed its move. The *agent manager* is responsible for agent lifecycle managment and coordination of agent operations with respect to the other managers in the pod.

Pods provide infrastructure for agent lifecycle operations such as creation, destruction, cloning, and moving. An agent is created by sending a request to a pod that it create an agent of a certain type. Agent desctruction can be

requested by the agent or externally by specifying the agent identifier and its host pod. Once the agent exists, it can initiate lifecycle transitions by sending requests to its pod. Furthermore, lifecycle transitions can be initiated by the pod or by an external entity (possibly subject to authorization). Lifecycle operations can be advisory in nature: the agent can reserve right of refusal to, for example, protect critical computations.

All non-local agent lifecycle operations flow through the liaison. To tell a pod to create an agent, a create command is constructed and pushed to the target pod via the liaison. Moving an agent to a new pod involves serializing the agent, embedding it in a command, and sending to through the liaison to the target pod.

2.3 Liaisons: Mediated Messaging

In Mojave, we have chosen to avoid direct address-based point-to-point communication, and have instead opted for mediated, name-based messaging using an infrastructure called a *liaison*. A liaison is a network communication buffer with database capabilities. Messages are written to a liaison, which are then dispatched to the target via a registered callback that is selected according to the target's name (or other distinguishing fields). Since communication is name-based, we avoid the complicated address management infrastructure required in mobile agent platforms that support point-to-point communication [8][6][2]. While mediated messaging involves an indirection, it supports an architecture that scales up to large numbers of pods, and is suited to wide-area agent computing with its intermittent connectivity and mobility management issues.

Databases naturally support store-and-forward mechanisms, and atomic transactions. Because a liaison is a database, it can store messages for agents that are moving or are disconnected, and deliver the stored messages to them when they are "back online". Mojave can use the liaison's pattern matching capabilities to support selective multicasts and other messaging paradigms that would otherwise require complex implementations. Because databases support atomicity guarantees via transactions, a liaison can trivially support guaranteed delivery of agent to agent messages, which becomes increasingly important in large-scale computations.

Given these compelling arguments, it is natural to ask why mediated messaging isn't the defacto messaging standard. The reason is that wide-area thin-client computing (aka "network computing") which has recently emerged as a hot topic, has a different set of requirements from LAN-based distributed computation. Wide-area applications operate over bursty networks, thus making the real-time efficiency of direct messaging less relevant. The challenge in wide-area, thin-client applications is to deal with mobile, intermittently connected users and mobile software components. Mediated messaging avoids point-to-point communication and therefore deals nicely with issues of physical and logical mobility. The penalty of reading and writing to a network communication buffer (a tuple space) might have seemed prohibitive in a LAN-based application, but is insignificant compared to the over-the-wire and over-the-air delays of wide-area communication. Furthermore, advances in main-memory database technology are mitigating this inefficiency. Mediated messaging scales well to large device populations where devices are too small to network directly with all other devices.

3. Mojave Implementation

Mojave relies on Jini and Tspaces for a number of support services, and on XML for mobility certificate design and implementation. Here we describe some of the specifics of how we use Jini and TSpaces in our architecture. The section also discusses choices in mobility certificate design, and our position in this space. The Mojave system monitor (and a thin-client implementation of the same) is discussed as an application that leverages the fact that all agent messaging and movement is routed through the liaison.

3.1 Component Assembly and Software Distribution

Mojave uses Jini for assembling and managing the system components. Jini's service discovery and dynamic proxy downloading mechanisms are well-suited for system component composition and software distribution. Mojave components: pods, liaisons, and agents, map to Jini services, but each has different characteristics.

Every pod is a Jini service that registers its service proxy with the Jini Lookup Services (JLUS) for the purpose of sharing information with other pods. The pod proxy contains information such as the pod identifier, the pod's host name and its IP address. A Jini helper class called the *Service Discovery Manager* is used by pods to maintain a local cache of all registered proxies (i.e. all the reachable pods). The Jini infrastructure automatically maintains the cache via a remote event protocol that updates cache state according to JLUS membership changes. As new pods register or become disconnected, Jini automatically refreshes the distributed caches. Agents can access the local pod cache as a basis for determining where to move next.

A liaison registers its proxy with the JLUS for the purpose of encapsulating communication with a tuplespace. Every pod looks up a liaison proxy to establish a connection to the tuplespace for mediated messaging. This type of

proxy is often referred to as a *protocol proxy* because it encapsulates service-specific protocols behind an interface.

Agent proxies differ from pods and liaisons in that the proxy being distributed via the JLUS is a template definition of agent itself, and there is no backend service. When a pod is directed to create an agent (via a liaison-mediated message), the pod looks up the agent by type in the JLUS. After the agent is downloaded, the pod assigns it a unique identifier and the agent manager starts the agent running. Once an agent is injected into the Mojave system and given an identity, it is no longer necessary to use Jini for that agent instance.

3.2 Mobility and Messaging

While Jini services are used for component assembly via dynamic proxy distribution, a liaison is used to distribute the serialized state of a mobile agent. Furthermore, all coordination among pods and agents also flows through a liaison.

IBM's TSpaces [15] is used to implement the liaison. TSpaces is a distributed communication buffer that supports a combination of Linda-like tuple operations, event push operations, and database operations. Tuple spaces support transactions for guaranteed message delivery, and a stronger query model than JLUS. Adding new meta-data fields to a tuple schema is a lot simpler than changing the service entries in a JLUS. The pattern matching-based callbacks that are supported by TSpaces allow pods to express selective subscriptions more succinctly than would be feasible with a JLUS.

Mojave operations take the form of specially formatted tuples. Pods register with the TSpace using tuple templates that match specific commands. When a command tuple is written to the TSpace, it may match one or more registrations. The TSpace will call back to all matching registrations, thus pushing the command to the appropriate pods. Mojave assigns globally unique identifiers to pods and agents thus ensuring that callbacks are delivered to the right end points. Before an agent moves, its pod deregisters callbacks for that agent, and when an agent arrives at a new pod, its callbacks are reregistered.

This architecture is appropriate for thin clients because they require only a small and fixed number of network connections (whether they support pods or not). Furthermore, agents and pods do not need to maintain pointers to end points. Many agents systems use complex techniques such as tombstones or forwarding to maintain communication pointers to agents as they move. Mojave uses identifiers and mediated tuple pushing for communication. Thus, no matter where the agent is, it can easily be located through callback registrations.

Agents are moved from one pod to another by embedding them within a tuple. The challenge is to move an agent instance without requiring that the TSpace and the pod have previous knowledge of the agent's type. Simply serializing the agent is insufficient because both the TSpace JVM and the target pod's JVM will fail to load the class when the agent instance arrives. Our solution is to embed the agent within a `java.rmi.MarshalledObject`. When an instance is embedded into a `MarshalledObject`, it is serialized and annotated with its codebase URL. We insert the `MarshalledObject` into a field in a *move agent* tuple that is typed as a `MarshalledObject` rather than as an agent instance. Serialization provides the agent state mobility, the anotated codebase allows the target pod to load the agent class, and the tuple is weakly typed with respect to agents thus eliminating the need for agent class knowledge in the TSpace.

Another implementation challenge is coordinating agent message delivery with agent mobility. The challenge is to deliver messages that are sent to an agent while it is in-transit and thus, not registered with any pod. No messages must be missed, they must be delivered in total order, and their delivery must be coordinated with message flow occuring after the agent has arrived at its destination. We solve this problem by taking advantage of several features of TSpaces:

- the space can be configured to return results in FIFO order
- an expiration time can be specified for each tuple written to the space
- fields can be indexed to maintain an ordering with respect to queries
- tuples are timestamped when the are written into the space
- a default index exists on timestamps
- the space can be scanned according to a query criteria and an index

These features are used as follows. The liaison (TSpace) is configured to maintain FIFO order for all tuples. All agent messages are written to the space with an expiration of 30 seconds. When an agent arrives at a pod as a result of a move, it registers for events with the option to receive backlog (i.e. events that occured while the agent was in-transit). If backlog is requested, then all events that occur from that point forward are held in the event queue while backlog is retrieved. Backlog is gathered by composing a scan query like, "scan for all events for subscription *S* that are new than timestamp *T*, using index *TIMESTAMP*." The agent keeps track (transparent to application code) of the timestamps of the most recent events of each subscription

type. This scan query returns a list of event tuples that are then merged with the event queue. Once the merge is complete, the event queue is unblocked to allow events to be delivered to the agent. If backlog is not requested, then any events that occur will expire from the tuple space without being delivered.

3.3 Externalized Mobility Policy

Separation of agent function and mobility policy provides a high degree of flexibility and reuse for component developers and administrators. Two aspects of the mobility policy are externalized: the policy description called a *mobility certificate*, and agent-specific monitoring tasks called *sentries*. The logic for mobility policy enforcement is located in the pod, rather than the agent, to improve overal scalability and performance. The enforcement logic is contained in the certificate rules engine which accepts incoming certificates and sentries as parameters for enforcement.

3.3.1 Mobility certificates. Mojave agent mobility policies are expressed as an XML mobility certificate that can be associated with an agent at creation time and edited later at runtime with standard XML tools. Identical certificates can be attached to multiple agents and instances of the same agent type can have different certificates. This allows a designer or administrator to, for example, elevate the importance of a particular agent by giving it a mobility certificate whose rules always move it to a friendly and safe environment.

A mobility certificate contains predicate-based rules relating to computational and physical constraints, and declarative rules for agent peer constraints. Figure 3 shows the logical breakdown of the kinds of statements contained in a mobility certificate.

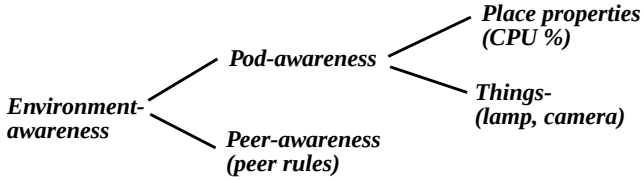

Figure 3 - Logical breakdown of mobility certificate statements.

A declarative existence constraint on a thing can be expressed as “only move to pods that have lamps in them.” A certificate with this pod-awareness rule in it will prevent the associated agent from moving to a pod without lamps. *Move rules* are pod-awareness rules that are triggered by an event, that can cause an agent to move. Event sources fall into three categories: pod state, place state, and thing state. An example of a pod state move rule would be, “if the CPU on this pod is more than 80% loaded, then move.” A place state example would be, “if the temperature in the room which the pod is representing is more than 80F, then move.” This statement presumes that there is a way to measure ambient temperature at the pod place; this might be expressed as an existence constraint on a rooted sensor agent. A thing state rule would look like, “if any lamp is on in the room that this pod represents, then move.”

Peer rules express affinity between agents. A *pull* (peer) *rule* declares a collocation constraint between two agents. For example if agents A and B are associated by a pull rule, this means that agent A's move causes agent B to move. A pull rule can be refined to include a neighborhood qualification which represents how close A needs to be from B. A neighborhood specification of “subnet” says that it is sufficient if A and B are in the same subnet. We plan to have constructs for performance-based and demand-based agent affinity as well.

Rule firing may cause chaining where the execution of one mobility rule can fire off another, and so on.

3.3.2 Sentries. The certificate rule engine is located in the pod, thus introducing the issue of where to locate the event monitoring machinery. For example, should the pod have the built-in ability to monitor CPU utilization? What about monitoring things such as lamps? We solve this problem with sentries. Sentries are Jini proxies registered in the JLUS that can be downloaded to the pod’s certificate rule engine as needed, according to the incoming agent mobility certificates. Dynamically downloaded sentries free the pod from the details of monitoring and from any system-specifics that may be involved.

Reactive, event-based mobility is achieved via *watch for* predicates in the move rules as described above. Agents rely upon sentries for monitoring and sending events to the pod's certificate rule engine. A sentry encapsulates a monitoring mechanism, and possibly a threshold value for event generation (active sentries). Four types of sentries are shown in Table 2.

Table 2: Sentry types.

	Passive	Active
Unshared	1 agent/sentry Agent polls sentry	1 agent/sentry Sentry notifies
Shared	N agents/sentry Agents poll sentry	N agents/sentry Sentry notifies

Passive sentries are polled by the rule engine and are best suited for monitoring an instantaneous value. Active sentries have their own thread of control that detects threshold violations and generates events. Active sentries

are suited for monitoring trends. Shared sentries are intended to mitigate the bloat that may occur as complex sentries proliferate the system. For example, many agents would use a CPU utilization sentry. A single shared sentry can monitor CPU thresholds for multiple agents.

Below is a sample Mojave mobility certificate illustrating two kinds of rules: a place related rule that specifies conditions that must hold true in a pod for it to be suitable for an agent's execution, and a peer rule that causes an agent to move as a consequence of the movement of closely collaborating peer agents.

```
?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE amos SYSTEM "/usr/venuv/Puma.dtd">
<amos>
<watchFor>
 <!-- cpu units are percentage -->
 <placeState
  sentry=com.mot.mojave.sentry.CPUSentry>
  cpu < 80
 </placeState>
</watchFor>

<peerRule>
<pull>
 <agentId>
  intrusion-detection-agent1
 </agentId>
 <distanceFunction>
  subnet
 </distanceFunction>
</pull>
</peerRule>
```

The `sentry` attribute specifies the name of a sentry interface to be located in a JLUS. Although the sentry encapsulates the actual monitoring constraint, the same constraint is specified again in the body of the `placeState` tag. This is done for two reasons. First, the value, 80, is used to parameterize the sentry. And second, by specifying the rule in the XML, it can be used for the purpose of lookup and trading.

An untested but desirable result would be that operators could look at a mobility "document" of an agent application (the union of the mobility certificates of all its agents), and be able to edit this document at runtime as they identify mobility policies that are ineffective. Such an iterative build-and-test approach to defining mobility policies would allow the mobility policy to converge quickly based on empirical evaluation.

3.4 Mojave System Monitoring

One of the notable advantages of the Mojave architecture is the ease in which a system monitor can be created. The monitor simply registers for all the command tuples relating to agent lifecycle changes, and then reacts to the callbacks by updating its model of the system and its user interface. The tuplespace is like a universal observation deck which gives comprehensive visibility into an agent application's state with a simple event subscription.

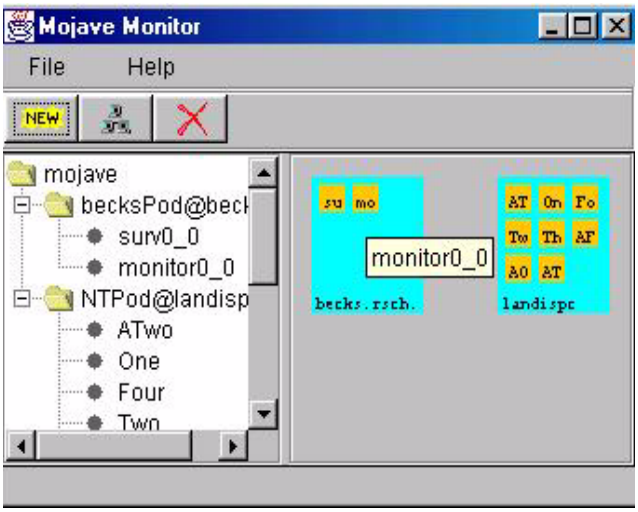

Figure 4 - Mojave Monitor.

Figure 4 shows a screen capture of the Mojave Monitor. The panel on the left shows structural view of pods and agents in the system, the panel on the right shows the same information graphically. The monitor supports the ability to inject agents into the system, to dispose of agents, and to log and replay sessions.

We are currently building a palm pilot application, called the *Mojave Remote Control* (MRC), to demonstrate remote control of Mojave agents. The architecture is shown in Figure 5.

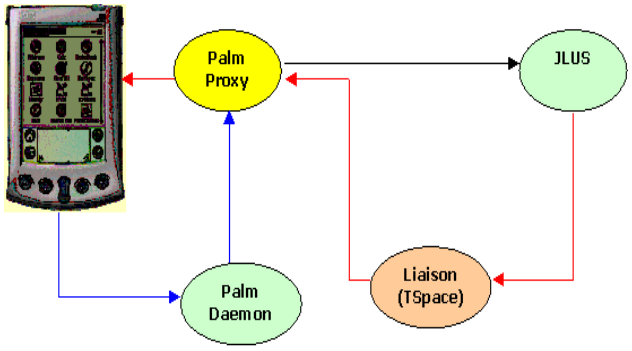

Figure 5 - MRC architecture.

The PDA interface allows mobile operators to poll the locations of agents and pods in the system. A Palm V is configured with the J2ME CLDC 1.0 VM (KVM) from Sun [17]. The Palm V connects over a modem using a TCP/IP remote call via PPP. A Palm Daemon process running on a server listens on a specified port for Palm V connection requests. Once a connection is established, the Palm

Daemon instantiates a Palm Proxy process. The Palm Proxy locates a liaison via the JLUS and monitors the Palm V connection for commands. The Palm V user interacts with a UI to initiate commands to the Palm Proxy that writes them to the liaison for dispatch to the appropriate pod(s). The Palm Proxy is registered for replies and receives them when they are written by the pod(s). The Palm Proxy forwards the replies to the application running on the Palm V that updates the UI.

4. Example Application: Systems Management

Our goal was to build a systems management application that demonstrates the self-installation and self-reconfiguration aspects of Mojave. We wanted to keep the application logic simple (but useful) so the reconfiguration aspects are highlighted. A systems management example was motivated by the following:

- Even simple network management tasks are hard to setup, as the placement of tasks on machines involves time-consuming scripts.
- Even after an application is started up, there is a lot of administrative overhead in re-mapping the application as machines are taken down for administration, repair, or failure.
- Mobile agents for automated software (re)distribution can enable application components that self-distribute.
- Mobility certificates can specify the policies for component migration (for load redistribution).

Our example application tackles a performance monitoring (PM) task for a moderately sized network (1000 machines). The management task runs on machines that are multi-use and not dedicated to this management task; therefore, the PM task has to share cycles with other, possibly unrelated system and user tasks. The task runs over some subset of the 1000 machines on which pods are installed. The overall PM task measures the "network stress" of each machine over time, where network stress is trivialized in terms of the ICMP in/out traffic.

This task is divided into a group of identical Mojave PM-agents, each of which monitors some subset of the IP addresses in the 1000 node network. The creator agent instantiates one PM-agent on a host and begins feeding it IP addresses as they are discovered. As the number of IP addresses grow, the PM-agent creates clones and assigns disjoint sets of IP addresses to each clone agent and causes it to move off to a suitable host. Each of these agents has a mobility certificate that contains a CPU-sensitive mobility rule so that the PM-agent moves when load surpasses a threshold. As clones are created, they migrate from the initial machine to the various other pod-hosting machines in the network, and begin their monitoring. Every so often, they send information to a display agent about abnormally stressed machines. When CPU load on agent-hosting machines violates threshold, say when a user logs on to the machine, the resident PM-agent(s) migrate to lightly used machines.

The PM-agent takes advantage of the monitoring capability of the Simple Network Management Protocol (SNMP) architecture [12] by relying on SNMP protocol agents to abstract a device's status into SNMP variables. SNMP protocol agents are lightweight and widely deployed in most commercial computing devices (workstations, routers etc.). Management Information Bases (MIBs) that describe the set of SNMP variables pertinent to an equipment class, have been standardized for a number of equipment classes. SNMP's API for MIB access and manipulation is straightforward.

Our example architecture utilizes SNMP's strong monitoring capabilities while replacing its static management infrastructure.

5. Related Work

Mobile agents have become an active area of research since the internet has made a global computing infrastructure widely available. Java-based mobile agent frameworks include Aglets [3], Sumatra [9],, Voyager [2] and Hive [6]. *Hive* is closest to Mojave in its aim to support coordinated actions via large "ecologies" of small agents. Also similar is Hive's focus on scaling down to small devices, or "things". Unlike Hive (which is built over Java-RMI), Mojave layers over Jini to take advantage of the services it offers for wide-area distributed computing. While both Mojave and Hive avoid direct agent-to-agent communication, Mojave's use of mediated messaging (via tuplespaces) allows for more powerful message routing than is possible with RMI. Tuplespaces support message buffering as well as database like matching, both of which are enablers for smart routing. An elegant aspect of the Hive architecture is that it models cells (the equivalent of pods) as smart containers that have attributes as well (e.g the location of the cell, the kind of agents it hosts). This is an important idea, in *located computing* where an agent wants to execute only at locations that satisfy a particular predicate. Future implementations of Mojave will similarly model containers as stationary agents whose attributes may model aspects of located computing. Hive makes extensive use of trading for dynamically composing agents, the novelty in Hive being that the trader is written in RDF, a dialect of XML with semantic capabilities.

FarGo [8] (and related Hadas [7]) proposes component architectures for reconfigurable distributed computing, and provide a Java-based platform for the same. Unlike Mojave and Hive, FarGo presupposes direct referencing between agents, and maintains virtual references across objects that move relative to each other. Additionally FarGo *complets* lack the autonomous mobility of agents (and require external entities to move them around). Nevertheless, the FarGo component relocation ideas are equally applicable to mobile agents with indirect referencing models. FarGo's relocation scripting language is event-driven and allows movement policies to be expressed as event-action pairs Events could include environmental conditions at a pod (e.g rapid change in CPU load), a pod-collective (e.g the total number of complets), and related-agent events (e.g a cohort agent is moving) This rule-based mobility scheme where complets encapsulate movement policies allows an application to reconfigure itself without the bottleneck of a centralized mobility manager.

The Agent Development Kit (ADK) [5] focuses on reusing JavaBeans development environments to facilitate the creation of new agent types. It proposes a universal class-level organization of all agents into three types of beans - navigators, performers and reporters. The idea is to separate the function of an agent from its itinerary and the reporting format. By encoding agent components as beans, developers can leverage their understanding of JavaBeans in writing agents. The notion of reusing JavaBean IDEs for Mojave is attractive and one we plan to examine further. However, ADK's organization of agent code seems restrictive.

The MbD [4] project implemented a malleable application platform directly over standard operating system services. While this approach has the advantage of being language-independent, there is substantial re-implementation of object support services that are standard in Jini and other distributed object platforms. Further, the widespread acceptance of Java has made the language-dependent argument far less compelling. MbD explores the use of malleable architecture for a number of systems management scenarios, most of which are still relevant to Mojave.

6. Conclusions and Future Work

This paper describes our initial experiences with building adaptive, auto-configuring applications using a wrapper framework to encapsulate typical java components. Our experience is that the wrapper approach to agent management makes it possible to reuse components in different contexts. The use of a tuple space as the messaging and mobility hub enables changes in an applications coordination behavior without rewriting components. Areas of further work include examining application development environments (and methodologies) for reactive agents, richer mobility certificate vocabularies and capabilities. security issues in agent applications that span multiple security domains, and dealing with system stability issues in agent computation with large numbers of agents.

Ongoing work in Mojave will focus on making applications easier to create, extending the Mojave thin-client platform, mobile agent security, examining wide-area issues for applications that span security domains, and adaptation issues such as optimizing performance and resource allocation, system stability, and emergent behavior.

The near-term direction for improving ease-of-use for Mojave application programmers will be to use a JavaBean like environment to graphically specify design patterns among the agents comprising an application. We also wish to integrate the creation and reuse of mobility certificates and sentries into the bean environment.

For the thin-client platform we are looking at wireless implementations to demonstrate a mobile operator interface to Mojave. Another direction being looked into is the use of the Wireless Application Protocol [16] to connect a wireless device to Mojave via a web server. WAP is a fairly straightforward bridge between thin clients and a Mojave bridge.

Our current work assumes a trusted environment, which covers a large class of enterprise applications, but future work will involve enhanced agent security to support untrusted environments. There are many known issues in mobile agent security [18] including protection between agents, protecting pods from agents, and protecting agents from pods. We will rely to some extent upon security capabilities of the JVM, Jini, RMI, and TSpaces. The JVM and TSpaces have security models, whereas RMI and Jini models are still in the specification process.

Both wide-area systems management and other potential application areas for Mojave (e.g Home Networking) require it to operate across multiple locales (those of the remote home controller and the controlled home). In this respect, we are looking into layering an agent mobility protocol over HTTP, which has the advantage of being ubiquitously deployed.

There are several areas of long-term research we plan to explore. We have begun to look at market-based approaches to performance and resource management. We hope to augment our mobility certificates and sentries with the ability to participate in a resource marketplace which does not have a central point of control. This approach is promising but it is not clear how efficient market-based approaches are, and whether overall system stability can be ensured. Finally, we wish to explore the notion of emergent behavior where the dynamic composition of various simple

agents can result in dramatically more sophisticated computations.

7. References

[1] Cardelli, L., "Wide-Area Computation", *ICALP'99 Invited Paper. Lecture Notes in Computer Science, Vol. 1644*, Springer, 1999.

[2] Glass, G., "ObjectSpace Voyager Home Page", http://www.objectspace.com.

[3] Lange, D. B. and Chang, D. T., "IBM Aglets Workbench: Programming Mobile Agents in Java," *Technical Report*, IBM Tokyo Research Lab, September 1996.

[4] Goldszmidt, G. et al., "Distributed Management by Delegation," PhD Thesis, Columbia University, 1995.

[5] Gschwind, T. et al., "ADK- Building Mobile Agents for Network and Systems Management from Reusable Components,", IBM Zurich Labs Technical Report

[6] Minar, N. et al, "Hive: Distributed Agents for Networking Things,"*Proceedings of ASA/MA '99*, 1999.

[7] Holder, O. et al., "System Support for Dynamic Layout of Distributed Applications," *Technical Report EE Pub #1191*, The Technion - Israel Instt. of technology, 1998.

[8] Holder, O, "Dynamic Layout of Distributed Applications in FarGo," *Technical Report EE Pub #1199*, The Technion - Israel Instt. of technology, 1998.

[9] Ranganathan, M. et al., "Network-Aware Mobile Programs," *Proceedings of the 1997 USENIX Technical Conference*, 1997, pp.91-94.

[10] Arnold, K. et al., "The Jini Specification," http://www.sun.com/jini.

[11] Vasudevan, V., Bannon, T., "WebTrader: Discovery and Programmed Access to Web-Based Services," Poster Session at the *Eighth Intl. WWW Conference*, 1999, also at http://www.objs.com/agility/tech-reports/9904-WebTraderWWW8Poster.html.

[12] Stallings, W., *SNMP, SNMPv2, and RMON*. Addison-Wesley, Reading, Mass., 1996.

[13] AdventNet "SNMP Agent Toolkit", http://www.adventnet.com.

[14] Motorola, "Oracle portal-to-go announcement", http://industry.java.sun.com/javanews/stories/print/0,1797,19466,00.html.

[15] TSpaces project home page, http://www.almaden.ibm.com/cs/TSpaces.

[16] WAP Forum Homepage, http://www.wapforum.org.

[17] Sun Microsystems, "The K virtual machine (KVM)", http://java.sun.com/products/kvm/wp.

[18] NIST, "Mobile Agent Threats, Countermeasures, and New Research Areas", http://csrc.nist.gov/mobileagents/web-overview/index.htm.