Naplet: A Flexible Mobile Agent Framework for Network-Centric Applications*

Cheng-Zhong Xu
Department of Electrical and Computer Engineering
Wayne State University, Detroit, Michigan 48202
czxu@ece.eng.wayne.edu

Abstract

As an alternative distributed programming paradigm, mobile agent technology has recently received much attention with the explosive growth of the Internet. To support mobile agent oriented programming, we have developed a Java-compliant system, Naplet, for increasingly important network-centric distributed applications. The system provides programmers with constructs to create and launch agents and with mechanisms for management of agent execution. Its distinctive features include a structured navigation facility, reliable agent communication mechanism, open resource management policies, and secure service interfaces between alien agents and agent servers. This article reports the architecture of the Naplet system and its application in network management.

1 Introduction

An agent is a sort of special object that has autonomy. It behaves like a human agent, working for clients in pursuit of its own agenda. A mobile agent has its defining trait ability to travel from machine to machine on open and distributed systems, carrying its code, data, and running state. Mobility of the software agents, particularly their flow of control, leads to a novel distributed processing paradigm. In the conventional client/server paradigm, a server exposes pre-defined service interfaces and clients request services by sending data to the server. By contrast, the mobile-agent based processing paradigm allows the clients to define their own preferred ways of processing in agents. The agents fulfill their missions autonomously by roaming between the servers.

Mobile agent grew out earlier technologies like mobile objects and process migration [11]. Early mobility research was mostly targeted at "closed" distributed systems, in which programmers have complete knowledge about the system and full control over the disposition of the machines. By contrast, agents are tailored to open and distributed environments because of their ability to perceive, reason, and act in their environments. It is the autonomy that makes an agent survive intermittent or unreliable Internet connections and decouple clients from servers.

Many researchers speculated that mobile agents are inevitable for open and distributed environments. Lange and Oshima specifically seven good reasons for mobile agents[9], including (a) reducing the network load; (b) overcoming network latency; (c) encapsulating protocols (self-explained data); (d) executing asynchronously and autonomously; (e) adapting to the change of environment (agility). It is also inherently heterogeneously, robust and fault-tolerant. Although none of the individual advantages represents an overwhelming motivation for their adoption, their aggregate advantages facilitate many new network services and applications, Harrison, et al. argued [6].

Until recently, mobile agent systems were developed primarily based on script languages like Tcl [4] and Telescript [16]. Latest proliferation of mobile agent technology is mostly due to the popularity of Java. The Web opened vast opportunities for applications suited for mobile agents. The Java virtual machine and its dynamic class loading model, coupled with several of Java other features, most importantly serialization, remote method invocation, and reflection, facilitates constructions of first class mobile agent systems. Many systems have been developed, most notably including Aglet [8], Ajanta [13], Concordia [15], D'Agent [4], Odyssey [16], and Voyage [12]. Readers are referred to [5][14] for excellent reviews.

In [17][18], we presented a mobile agent system Traveler for global high performance computing services. As a companion, the Naplet system is an experimental framework in support of Java-compliant mobile agent for network-centric distributed applications. Like other existing systems, it provides constructs for agent declaration, confined agent execution environments, and mechanisms for agent monitoring, control, and communication. The Naplet system is built upon two first-class objects: Naplet and NapletServer. The former is an abstract of agents, which defines hooks for application-specific functions to be performed on the servers and itineraries to be followed by the agent. The latter is a dock of naplets. It provides naplets with a protected runtime environment within a Java virtual machine. Although more than one JVMs can be running simultaneously on a host, each host can contain at most one NapletServer. The NapletServers are running autonomously and they collectively form an agent flow space for the Naplets.

* This research was supported in part by NSF grants ACI-0203592, CCR-9988266, and EIA-9729828.

Compared with other mobile agent systems, the Naplet system has the following distinctive features: structured navigation facility, reliable inter-agent communication mechanism, open resource management policies, and secure interface between alien agents and naplet servers. In the following, we present the NaLet architecture and delineate these features. We conclude this paper with an application of the system in network management.

2 Naplet Architecture

2.1 Naplet Class

Naplet is a template class that defines the generic agent. Its primary attributes include a system wide unique immutable identifier, an immutable codebase URL, and a protected serializable container of application-specific agent running states.

```
public abstract class Naplet implements Serializable, Cloneable {
    private NapletID nid;
    private URL codebase;
    private Credential cred;
    private NapletState state;
    private transient NapletContext context;
    private Itinerary itin;
    private AddressBook aBook;
    private NavigationLog log;
    public abstract onStart();
    public void onInterrupt() {};
    public void onStop() {};
    public void onDestroy() {};
}
```

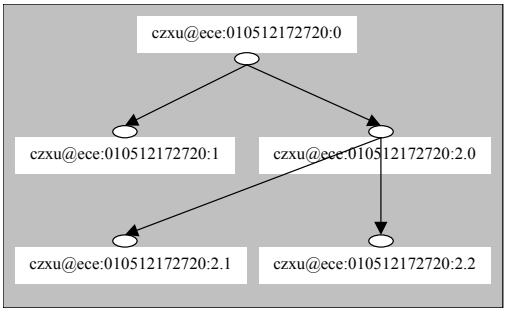

Figure 1. Hierarchical Naplet ID

The naplet identifier contains the information about who, when, and where the naplet is created. In support for naplet clone, the naplet identifier also includes version information to distinguish the cloned naplets from each other. Since a naplet can be recursively cloned, we use a sequence of integers to encode the heritage information and reserve 0 for the originator in a generation. For example, a naplet identifier "czxu@ece.eng.wayne.edu: 010512172720:2.1" represents the naplet that was cloned from the original one created by a user "czxu" at 17:27:20 May 12, 2001 in the host "ece.eng.wayne.edu". The heritage information is as follows.

The Naplet system supports lazy code loading. It allows classes loaded on demand and at the last moment possible. The codebase URL points to the location of the classes required by the naplet. The classes and their associated resources such as texts and images in the same package can be zipped into a JAR file so that all the classes and resources needed are transported at a time.

Note that both the naplet identifier and codebase URL are immutable attributes. They are set at the creation time and can't be altered in the naplet life cycle. To ensure their integrity, they can be certified and signed by the naplet creator's digital signature. The naplet credential will be used by naplet servers to determine naplet-specific security and access control policies.

As a generic class, Naplet is to be extended by agent applications. Application-specific agent states are contained in a NapletState object. Any objects within the container can be in one of the three protected modes: private, public, and private. They refer to the states accessible to the naplet only, any naplet servers in the itinerary, and some specific servers, respectively. For example, a shopping agent that visits hosts to collect price information about a product would keep the gathered data in a private access state. The gathered information can also be stored in a protected state so that a naplet server can update a returning naplet with new information.

NapletState attribute aside, Naplet class provides a number of hooks for application-specific functions to be performed in different stages of the agent life cycle: *onStart*(), *onStop*(), *onDestroy*(), and *onInterrupt*(). onStart() is an abstract method and must be instantiated by extended agent applications. It serves a single entry point when the naplet arrives at a host. The naplet executes in a confined environment, defined by its NapletContext object. The context object provides references to dispatch proxy, message, and stationary application services on the server. The context object is a transient attribute and is to be set by a resource manager on the arrival of the naplet. It can't be serialized for migration. The agent behavior can also be remotely controlled by its creator via onInterrupt(). Details of these will be discussed in the NapletServer architecture.

Mobile agents have their defining trait the ability to travel from server to server. Each naplet is associated with an Itinerary object for the way of traveling among the servers. It is noted that many mobile applications can be implemented in different ways by the same agent, associated with different travel plans. We separate the business logic of an agent from its itinerary in Naplet class. Each itinerary is constructed based on three primitive patterns: singleton, sequence, and parallel. Complex patterns can be composed recursively. In addition to the way of traveling, itinerary patterns also allow users to specify a post-action after each visit. The post-action mechanism facilitates inter-agent communication and synchronization. De-

tails about the itinerary mechanism will be discussed in the navigation section.

Many mobile applications involve multiple agents and the agents need to communicate with each other. In addition, an agent in travel may need to communicate with creator from time to time. In support of inter-agent communication, we associate with each naplet an AddressBook object. Each address book contains a group of naplet identifiers and their initial locations. The locations may not be current, but they provide a way of tracing and locating. The address book of a naplet can be altered as the naplet grows. It can also be inherited in naplet clone. We restrict communications between naplets who know their identifiers.

The last attribute of Naplet class is NavigationLog for naplet management. It records the arrival and departure time information of the naplet at each server. The navigation log provides the naplet owner with detailed travel information for post-analysis.

2.2 NapletServer Architecture

NapletServer is a class that implements a dock of naplets within a Java virtual machine. It executes naplets in confined environments and makes host resources available to them in a controlled manner. It also provides mechanisms to facilitate resource management, naplet migration, and naplet communication.

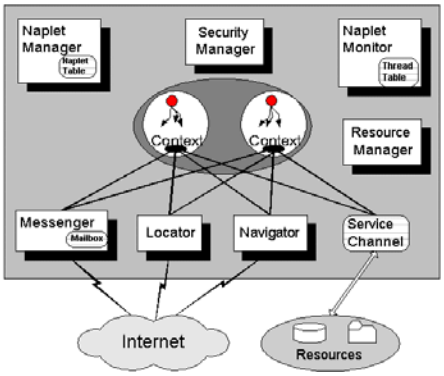

Figure 2. The NapletServer Architecture

Although more than one JVM can be running simultaneously in a host, we restrict that each host installs at most one naplet server. The naplet servers are run autonomously and cooperatively to form a naplet space where naplets live in pursuit of their agenda on behalf of their creators. Each naplet has a home server in the space. The naplet space can be operating in one of the two modes: with and without a naplet directory. The directory tracks the location of naplets and the centralized directory service simplifies the task of naplet management. Figure 2 presents the NapletServer Architecture. It comprises seven major components: *NapletMonitor*, *NapletSecurityManager*, *ResourceManager*, *NapletManager*, *Messenger*, *Navigator*, and *Locator*. ServiceChannel is dynamically created by the ResourceManager for communication between naplets and application-specific restricted privileged resources.

Each naplet is launched through its home NapletManager. It provides local users or application programs with an interface to launch naplets, monitor their execution states, and control their behaviors. The NapletManager maintains the information about its locally launched naplets in a naplet table. Footprints of all past and current alien naplets are also recorded for management purposes.

Naplet launch is actually realized by its home Navigator. The launching process is similar to agent migration. On receiving a request for migration from an agent or NapletManager, the Navigator consults the NapletSecurityManager for a LAUNCH permission. Then, it contacts its counterpart in the destination NapletServer for a LANDING permission. Success of a launch will release all the resources occupied by the naplet. Finally, the Navigator will also report a DEPART event to a NapletDirectory, if it exists.

On receiving a naplet launch request from a remote Navigator, the Navigator consults the NapletSecurityManager and ResourceManager to determine if LANDING permission should be issued. When the naplet arrives, the Navigator reports the arrival event to the NapletManager and possibly registers the event with the NapletDirectory. It then passes the control over the naplet to the NapletMonitor.

A naplet server can be configured or re-configured with various hardware, software, and data resources available at its host. The hardware resources like cpu time, memory size, and traffic volume on network ports constitute a confined basic execution environment. The software and data resources are largely application-dependent and often configured as services. For example, naplets for distributed network management rely on local network management services; naplets for distributed high performance computing need access to various math libraries. The ResourceManager provides a resource allocation mechanism, leaves application-specific allocation policy for dynamic re-configuration.

Note that the services available to alien naplets can be run in one of the two protection modes: priviledged and non-priviledged. Non-priviledged services, like routines in math libraries, are registered in the ResourceManager as open services and can be called via their handlers. By contrast, priviledged services like getting workload information and system performace must be accessed via ServiceChannel objects. The service channels are communication links between alien naplets and local restricted priviledged services. The ResourceManager creates the channels on requests. It passes one endpoint to the requesting naplet and the other

endpoint to the priviledged service. The priviledged resources are allocated by the ResourceManager and the access control is done based on naplet credentials in the allocation of service channels.

Each NapletServer contains a Messenger for inter-naplet communication. There are two types of messages: System and User. System messages are used for naplet control (e.g. callback, terminate, suspend, and resume); user messages are for communicating data between naplets. On receiving a system message, the Messenger casts an interrupt onto the running naplet thread. How the control message should be reacted by the naplet is left unspecified. It is defined by the naplet creator by defining a method onInterrupt(). On receiving a user message, the Messenger puts the message onto the naplet mailbox. It is the naplet that decides when to check its mailbox.

The Messenger relies on a Locator for naplet tracing and location services and supports location-independent communication. NapletID-based message addresses are resolved through a centralized naplet directory service or a distributed service based on the NapletManagers. Due to the network communication delay, the location information maintained in the NapletDirectory and the NapletManagers may not be current. The Messenger provides a message forwarding mechanism to handle messages that arrive earlier or later than the target naplet.

3 Structured Itinerary Mechanism

Mobility is the essence of naplets. A naplet needs to specify functional operations for different stages of its life cycle in each server as well as an itinerary for its way of traveling among the servers. The functional operations are mainly defined in the methods of onStart() and onInterrupt() in an extended Naplet class. The itinerary is defined as an extension of Itinerary class. Separation of the itinerary from the naplet's functional operations allows a mobile application to be implemented in different ways following different itineraries. One objective of this study is to design and implement primitive constructs for easy representation of itineraries.

The itinerary of a naplet is mainly concerned about visiting order among the servers. Each visit is defined as the naplet operations from the arrival event through the departure event. The visiting order encoded in the itinerary object is often enforced by departure operations at servers. Correspondingly, we denote a *visit* as a pair <*S; T*>, where *S* represents the operations for server-specific business logic and *T* represents the operations for itinerary-dependent control logic. For example, consider a mobile agent based information collection application. One or more agents can be used to collect information from a group of servers in sequence or in parallel. At each server, the agents perform information-gathering operations (S) (e.g. workload measurement, system configuration diagnosis, etc), as defined by the application. The operations are followed by itinerary-dependent operations (T) for possible inter-agent communication and exception handling. Different itineraries would lead to different communication patterns between the naplets. Different itineraries would also have different requirements for handling itinerary related exceptions. For example, in the case of a parallel search, naplets needs to communicate with each other about their latest search results. Success of the search in a naplet may need to terminate the execution of the others.

We note that servers listed in a journey route may not be necessarily visited in all the cases. Many mobile applications involve conditional visits. For example, in a mobile agent-based sequential search application, the agent will search along its route until the end of its route or the search is completed. That is, all visits except the first one should be conditional visits. We denote a *conditional visit* as <C→S; T>, where C represent the guardian condition for the visit <S; T>.

Based on the concepts of visit and conditional visit, we define visiting order in recursively constructed journey routing pattern. Its base is a SingletonPattern, comprising of a single visit or conditional visit. Assume P and Q are two itinerary patterns. We define three primitive composite operators *seq*, *alt*, and *par* over the P and Q patterns for constructions of SeqPattern, AltPattern, and ParPattern. Specifically,

- seq(P, Q) refers to a pattern that the visit of P is followed by the visit of Q;
- alt(P,Q) refers to a pattern that either the visit of P or the visit of Q is carried out by one naplet;
- par(P,Q) refers to a pattern that the visits of P and Q are carried out in parallel by a naplet and its clone.

Formally, the itinerary pattern is defined in BNF syntax as

<Visit V> **::=** <S> | <S; T> | <C→S; T>

<ItineraryPattern P> **::=** Singleton(V) | Seq(P, P) | Alt(P, P) | Par(P, P)

Following are a number of possible itinerary patterns constructed from visits and conditional visits.

Example 1: Consider an information collection application over a group of servers s_1, s_2, …, s_n. It relies on a single agent to accumulate information. The final results are reported back after the last visit. We declare the post action class as an implementation of a serializable and cloneable interface Operable.

```
class ResultReport implements Operable {
    public void operate ( Naplet nap ) {
        nap.getListener().report( ...);
              //get Listener and callback via a report method
    }}
class MyItinerary extends Itinerary {
    public MyItinerary( String[] servers ) {
        Operable act = new ResultReport();
        setItineraryPattern(new SeqPattern(servers, act) );
}}
```

Example 2: In the mobile information collection application, the servers can be visited by multiple agents simultaneously. The agents can report their results to their home directly.

```
class MyItinerary extends Itinerary {
    public MyItinerary( String[] servers ) {
        Operable act = new ResultReport();
    ItineraryPattern[]  _ip = new ItineraryPattern[ servers.length ];
    for (int i=0; i<servers.length; i++)
        _ip[i] = new SingletonItinerary( servers[i], act );
    setItineraryPattern (new ParPattern(_ip , act) );
}}
```

If needed, the agents can be synchronized with each other. Following is a generic operator for collective communications between the naplets.

```
class DataComm implements Operable {
    public void operate( Naplet nap ) {
        NapletID myID = nap.getNapletID();
        AddressBook aBook = nap.getAddressBook();
        Iterator iter = aBook.iterator();
        while (iter.hasNext()) {
            AddressEntry entry = (AddressEntry) iter.next();
            NapletID nid = entry.getNapletID();
            Messenger  handler = nap.getNapletContext().getMessenger();
            try {
                handler.postMessage( entry.getServerURN(), nid, message);
            } catch (NapletCommunicationException nce) {}
        }
        for (int i=0; i<aBook.size; i++)
            Message msg = handler.getMessage();
}}
```

Example 3: Four servers are to be visited by two naplets. The servers can be visited in a way like "par(seq(s0, s1), seq(s2, s3))".

```
class MyItinerary extends Itinerary {
    public MyItinerary( String[] servers ) {
        Operable act = new DataComm();
        String[] path0 = {servers[0], servers[1] };
        String[] path1 = {servers[2], servers[3] };
        ItineraryPattern[] _ip = new ItineraryPattern[2];
        _ip[0] = new SeqPattern( path0, act );
        _ip[1] = new SeqPattern( path1, act );
        setItineraryPattern( new ParPattern( _ip) );
}}
```

4 Naplet Location and Communication

4.1 Naplet Tracing and Location

The naplet system provides a reliable mechanism for location-independent communication between naplets. The mechanism relies on naplet tracing and location services provided by a class *Locator*. Recall that the naplet system can be running in one of the two modes: with and without naplet directory services. In a system with a naplet directory installation, the Locator can locate long-lived naplets by looking up the directory.

On launching or receiving a naplet, the Navigator registers the ARRIVAL and DEPARTURE events with the directory. The departure event is reported after a naplet is successfully dispatched. However, there is no guarantee of the time when the naplet arrives at the destination. The arrival event is reported after the naplet lands. We postpone the execution of the naplet until the arrival registration is acknowledged. This guarantees that the directory keeps the current location information about the naplets. If the latest registration about a naplet in the directory is a departure from a server, the naplet must be in transmission out of the server. If its latest registration is an arrival at a server, the naplet can be either running in or leaving the server. (Departure registration may not be needed)

Notice that the NapletDirectory services can be provided collaboratively by the NapletManager at each server. Since each naplet has its own home server and the home information is encoded in its naplet identifier, the naplet location information of can be maintained in their home managers. Correspondingly, any naplet tracing and location requests are directed to respective managers.

In a system without naplet directory services, naplets are traced by the use of naplet trace information maintained by the NapletManager of each server. The NapletManager maintains the source and destination information about each naplet visit. On receiving a tracing request from Messenger, the Locator checks with the NapletManager and returns with the current location if the naplet is in. Otherwise, the message will be forwarded to the server for which the naplet left.

The Locator service is demanded by Messenger for inter-naplet communication or by NapletManager for naplet management. The Locator class also caches recently inquired locations so as to reduce the response time of subsequent naplet location requests. The buffered naplet location information can be updated on migration either by home naplet managers in systems with distributed naplet directory services, or by remote residing naplet servers in systems with message forwarding.

4.2 Post-Office Messaging Service

The Messenger class provides a post office mechanism in support of persistent and asynchronous communication. On receiving a naplet, the Messenger creates a mailbox for its subsequent correspondences with other naplets or its home naplet monitor. A naplet can communicate to any other naplets presented in its AddressBook. The post office communication protocol works as follows.

Assume a naplet A residing on server Sa is to communication naplet B. The naplet A makes a request to Sa's Messenger. The Messenger checks with its associated Locator to find out naplet B's most recent server. If there is no directory service, the Messenger obtains the recent server according to the information in naplet A's address book. The address book contains information

about at least one residing server for each naplet. Expectedly, this server information may not be current either. In either case, we assume the naplet B used to be in server Sb.

Messenger in server Sa sends the message to its counterpart in server Sb. On receiving this message,

1. if naplet B is still running in the server, Sb's Messenger replies to Sa with a confirmation and meanwhile inserts the message into naplet B's mailbox. The confirmation message is kept in Sa's Messenger only for further possible inquiry from naplet A.
2. if naplet B is no longer in server Sb, Sb's Messenger checks with NapletManager against its *naplet trace* and forwards the message to the server to which the naplet moved. The forwarding continues until the message catches up the naplet B, say in server Sc. The Sc's Messenger replies to Sa with a confirmation and inserts the message onto B's mailbox.
3. if naplet B has not arrived in sever Sb yet (it is possible because the naplet might be temporarily blocked in the network), Sb's Messenger checks with NapletManager against its naplet trace and finds no record of naplet B. The Messenger will insert the message into a special mailbox, waiting for the arrival of naplet B. On receiving the naplet B, Sb's Messenger creates a mailbox and dumps the B's messages in the special mailbox to B's mailbox.

5 Security and Resource Management

A primary concern in the design and implementation of mobile agent systems on the Internet is security. Most existing computer security systems are based on an identify assumption. It asserts that whenever a program attempts some action, we can easily identify a person to whom that action can be attributed. We can also determine if the action should be permitted by consulting the details of the action, and the rights that have been granted to the user running the program. Since mobile agent systems violate this important assumption, their deployment involves more security issues than traditional stationary code systems.

5.1 Naplet Security Architecture

While agents are running on a server, the server resources are vulnerable to attacks by the agents. On the other hand, the agents are exposed to the server. Their carried confidential information can be breached and their business logic can even be altered on purpose. The design and implementation of mobile agent systems needs to protect agents and servers from any hostile actions from each other.

The Naplet system assumes naplets to run trustworthy servers. Security measures focus on protection of servers from any possible naplet attacks. The naplet security system is based on the JDK1.2 security architecture. The naplet security behavior is specified by application-specific security policies. A security policy is an access-control matrix that says what system resources can be accessed, in what fashion, and under what circumstances. Specifically, it maps a set of characteristic features of naplets to a set of access permission granted to the naplets. System administrators can configure the security policy according to the service requirements.

The first Naplet system release is compatible with the JDK1.2 security manager. Although no special security managers and class loaders have actually been implemented, many security features are left open for the future release, such as the work on authentication, and authorization of agents.

5.2 NapletMonitor

The JDK security architecture supports policy-driven, permission based, flexible and extensive access control. It leaves monitoring and control of resource consumption to application-specific resource management component. The Naplet system relies on the NapletMonitor to monitoring the naplet execution and control resource consumptions.

On receiving a naplet, the monitor creates a NapletThread object and a thread group for the execution of the naplet. The NapletThread object assigns a run-time context to the naplet and set traps for its execution exceptions. All the threads created by the naplet are confined to the thread group. The group is set to a limited range of scheduling priorities so as to ensure that the alien threads are running under the control of the monitor. The monitor maintains the running state of the thread group and information about consumed system resources including CPU time, memory size, and network bandwidth. It schedules the execution of the naplets according to resource management policies.

The current system release provides the monitoring and control mechanism. Various scheduling policies will be tested in future releases.

5.3 Resource Management and Access Control

Naplets can do few things without binding to server-side stationary services. The services include those provided by nodal operating systems, database management, and other user-level applications. The services can be implemented in legacy codes and most likely run in a privileged mode. That is, alien naplets should not be allowed to access these services directly. Although some resources can be opened to naplets by setting appropriate permission in

NapletSecurityManager, the security policy imposes naplet-specific control over resources.

To enact naplet-specific control policies, the ResourceManager creates service channels for communication between alien naplets and restricted privileged services. Each channel is essentially a synchronous pipe. The server assigns a pair of input/output endpoints, ServiceReader and ServiceWriter, to the service and leaves the another pair of endpoints to alien naplets. The service channel mechanism enables dynamic installation and reconfiguration of application services. Naplet-specific security permission policies can be easily implemented inside the service channels.

6 Naplet Based Network Management

In this section, we present an application example in network management to illustrate the Naplet-based agent-oriented programming paradigm. Network management involves monitoring and controlling the devices connected in a network by collecting and analyzing data from the devices. Conventional SNMP based approaches assume centralized management architecture and work in a client-server paradigm. Each managed device has a SNMP daemon (i.e. SNMP agent) running locally to collect network parameters and store them in a management information base (MIB) format. A management station communicates to the SNMP agents via a number of fine-grained get and set operations for MIB parameters. This centralized micro-management approach for large networks tends to generate heavy traffic between the management station and network devices and excessive computational overhead on the management station.

In [7], we proposed a mobile agent based distributed approach for network management, in which the management station programs demanded device statistics or diagnostics functions into an agent and dispatches the agent to the devices for on-site management. Figure 3 shows a Naplet based network management framework, MAN. The framework relies on privileged services provided by the local SNMP agent in each device.

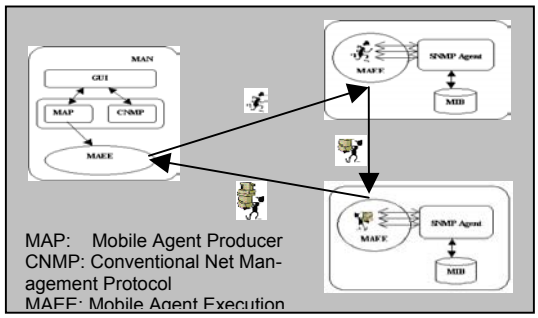

Figure 3. Mobile Agents for Network Management

6.1 Privileged Service for Access to MIB

In the MAN framework, network management station creates naplets and dispatch naplets to the target devices. The naplets access to MIB through local SNMP agents of the devices. For communication between Java-compliant naplets and SNMP agents, we deployed an AdventNet SNMP package on each managed device. The AdventNet SNMP package provides a Java API for network management.

Following is a NetManagement class extended from a naplet PrivilegedService base class. It is instantiated by the naplet ResourceManager and associated with a pair of *ServiceReader* and *ServiceWriter* channels: *in* and *out*. Through the input channel, the *NapletServer* gets input parameters from naplets and re-organize them into an AdventNet SNMP format. It then conducts a sequence of operations, as shown in private method *retrieve*(), to communicate with the AdventNet SNMP for required MIB information. The information is returned to the naplet through the out channel. The whole process can be repeated for a number of inquiries from the same naplet or different naplets.

```
import naplet.*;
import com.adventnet.snmp.beans.*;
public class NetManagement extends naplet.server.PrivilegedService  {
   public void run() {
      for (;;) {
         String cmd = in.readLine();
         Vector values = new Vector();
         StringTokenizer param = new StringTokenizer(cmd,";");
         while ( param.hasMoreElements() )
            values.addElement( (String)param.nextToken() );
         String result = retrieve( values );
         out.writeLine( result );
   }}
   private String retrieve( Vector parameters ) {
      StringBuffer result = new StringBuffer();
      String result = null;
      SnmpTarget target = new SnmpTarget();      //Create a SNMP target
      target.loadMibs("RFC1213-MIB");            // Load MIB
      target.setTargetHost(InetAddress.getLocalHost());
       //Set the SNMP target host
       target.setCommunity("public");
       Enumeration enum = parameters.elements();
       while(enum.hasMoreElements())   {
          target.setObjectID((String)enum.nextElement()+".0");
                                                  //set the required parameter
          result = target.snmpGet();
                                  // Issue an SNMP get request on Managed node
       }
       return result.toString();
    }}
```

6.2 Naplet for Network Management

The privileged service NetManagement is dynamically configured during the installation of *NapletServer*. It is accessed by incoming naplets through its registered name "serviceImpl.NetManagement". Following is a naplet example for network management. The NMNaplet class is

extended from the Naplet base class with name, a list of servers to be visited, and MIB parameters. It is also instantiated with a NapletListener object to receive information retrieved from the servers. All the information will be stored in a reserved ProtectedNapletState space. At last, the newly created NMNaplet object is associated with a custom designed itinerary.

On arrival a server, the naplet starts to execute its entry method: onStart(). It gets a handler to pre-defined NetManagement privileged service. It then sends parameters to NapletServer through a NapletWriter channel and waits for results from a NapletReader channel. Notice that NapletWriter and ServiceReader are two endpoints of a data pipe from naplets to servers. Another pipe links ServiceWriter to NapletReader.

When the naplet finishes its work on a server, it travels to the next stop. At the end of its itinerary, the naplet executes operate() method to report the results back to its home. Since NMItinerary defines a broadcast pattern, the naplet will spawn a child naplet for each server. The spawned naplets will report their results individually.

```
import naplet.*;
import naplet.itinerary.*;
public class NMNaplet extends Naplet  {
   private String parameters;            // MIB parameters to be accessed
   public NMNaplet(String name, String[] servers, String param, NapletListener
listener)  {
      super(name, listener);
      parameters = param;
      setNapletState(new ProtectedNapletState());// Set space to keep info.
      getNapletState().set("DeviceStatus",  new HashTable(servers.length));
      setItinerary (new NMItinerary (servers)); // Associate an itinerary
   }
   public void onStart() throws InterruptedException    {
      String serverName = getNapletContext().getServerURN().getHostName();
      Vector resultVector = new Vector();
      HashMap map = getNapletContext().getServiceChannelList();
      ServiceChannel channel = map.get("serviceImpl.NetManagemen);
      NapletWriter out = channel.getNapletWriter();
      out.writeLine( parameters );      // Pass parameters to servers
      NapletReader in = channel.getNapletReader();
      String result = null;
      while (  (result = in.readLine()) != EOF ) {
         resultVector.addElement( result );
      }
      Hashtable  status =  (Hashtable) getNapletState().get("DeviceStatus");
      status.put( serverName, resultVector );
      getItinerary().travel( this );
   }
   private class ResultReport implements Operable {
      public void operate( Naplet nap )       {
         if ( nap.getListener()!= null ) {
            Hashtable messages nap.getNapletState().get("message");
            nap.getListener().report( deviceStatus );
   }}}
   private class NMItinerary extends Itinerary { // Parallel Itinerary
      public NMtinerary( String[] servers )  {
         Operable act = new ResultReport();
         ItineraryPattern[] ip = new ItineraryPattern[servers.length];
         for (int i=0; i<servers.length; i++)
            ip[i] = new SingletonItinerary(servers[i], act);
         setRoute( new ParItinerary(ip) );
   }}}
```

7. Concluding Remarks

The Naplet mobile agent system was developed mainly for educational purpose. It separates mechanisms from policies in the aspects of migration, communication, and resource management so that various policies can be easily implemented. Its latest version is Naplet 0.12, available at http://www.ece.eng.wayne.edu/~czxu/ naplet.html.

References

[1] AdventNet, http://www.adventnet.com.

[2] A. Bieszczad and T. White, "management", In R. Glitho and S. Aidarous, editors, The *Fundamentals of Mobile agents for network Network Management*, Plenum, 1999.

[3] A. Fuggetta, G. Picco, and G. Vigna, "Understanding code mobility", *IEEE Transactions on Software Engineering*, 24(5):352—361, May 1998.

[4] R. Gray, D. Kotz, G. Cybenko, and D. Rus, "D'Agents: Security in a multi-language, mobile agent system", In G. Vigna, editor, *Mobile Agents and Security*, pages 154—187, Spring-Verlag, 1998.

[5] R. Gray, D. Kotz, G. Cybenko, and D. Rus, "Mobile agents: Motivations and state-of-the-art systems", In J. Bradshaw, editor, *Handbook of Agent Technology*, AAAI/MIT Press, 2001.

[6] C. Harrison, D. Chess, and Kershenbaum, "Mobile agents: Are they a good idea?", *Technical Report*, IBM, 1995.

[7] M. Kuna and C.-Z. Xu, "A framework for network management using mobile agents", In *Proceedings of the First Workshop on Internet Computing and E-Commerce (ICEC'01)*, April 2001.

[8] D. Lange and M. Oshima. *Programming and deploying Java mobile agents with Aglets*. Addison Welsey, 1998.

[9] D. Lange and M. Oshima, "Seven good reasonsfor mobile agents", *CACM*, 42(3):88—89, March 1999.

[10] A. Lienwand and K. Conroy. *Network Management: A Practical Perspective*. Addison Wesley, 1996.

[11] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration", *ACM Computing Surveys*, 32(3):241—299, December 2000.

[12] Objectspace, Inc. "Objectspace voyager core technology". www.objectspace.com.

[13] A. Tripathi, N. Karnik, et al, "Ajanta: A mobile agent programming system", Tech. Report TR98-016, Dept. of Computer Science, University of Minnesota, April 1999.

[14] D. Wang, N. Paciorek, and D. Moore, "Java-based mobile agents", *CACM*, 42(3):92—102, March 1999.

[15] D. Wang, et al, "Concordia: An infrastructure for collaborating mobile agents", In *Proceedings of the First Int'l Workshop on Mobile Agents*, pages 86—97, 1997.

[16] J. E. White, "Mobile agents make a network an open platform for third-party developers", *IEEE Computer*, 27(11): 89—90, November 1994.

[17] B. Wims and C.-Z. Xu, "Traveler: A mobile agent infrastructure for global parallel computing", In *Proceedings of the First IEEE Joint Symposium ASA/MA'99*, pages 258—259, October 1999.

[18] C.-Z. Xu and B. Wims, "Mobile agent based push methodology for global parallel computing", *Concurrency: Practice and Experience*, 14(8):705—726, July 2000.