NAPLET: A Flexible and Secure Mobile Agent Framework for Network-Centric Pervasive Applications

Cheng-Zhong Xu, czxu@wayne.edu
Wayne State University
http://www.ece.eng.wayne.edu/~czxu/software/naplet.html

1. Overview

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 (MA paradigm, in short) allows the clients to define their own preferred ways of processing in agents. The agents fulfill their missions autonomously by roaming between the servers.

The MA paradigm has the ability of (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 heterogeneous, 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.

The Naplet system is an experimental framework in support of Java-compliant mobile agent based distributed processing applications. 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.

In this article, we refer to NapLet as the Naplet system and naplet as an object of Naplet class. We also refer to naplet server (or server) as an object of NapletServer.
2. 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.

```java
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() {}
}
```

The naplet identifier contains the information about who, when, and where the naplet is created. In support of naplet clone, the naplet identifier 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 heritage information and reserve 0 for the originator in a generation. For example, a naplet id “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.

![Hierarchy Diagram](attachment://image.png)

**Figure 1. Hierarchical Naplet ID**
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 naplet classes and their associated resources, such as texts, images, or sounds in the same package can be zipped into a JAR file so that all the classes and resources the naplet needs 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. Details 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.

3. 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.

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 its exists.

On receiving a naplet launch request from a remote Navigator, the Navigator consults the NapletSecurityManager and ResourceManager to determine if a 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 and 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: privileged and non-privileged. Non-privileged 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 performance must be accessed via ServiceChannel objects. The service channels are communication links between alien naplets and local restricted privileged services. The ResourceManager creates the channels on requests. It passes one endpoint to the requesting naplet and the other endpoint to the privileged 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.

### 4. 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 \(<C \rightarrow S; T>\), 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 singleton pattern, comprising of a single visit or conditional visit.

```java
public class SingletonPattern implements ItineraryPattern {
    private URN server;   // The naplet server to be visited
    private Operable action;  // Pre and Post-visit actions
    private ItineraryIterator iter; // Locally defined serializable iterator over this itinerary pattern

    public void go( Naplet nap ) throw UnableDispatchException {
        if (iter.hasNext()) {
            URN server = ((SingletonPattern)iter.next()).server;
            nap.getNapletContext().getNavigator().toDispatch( server, nap );
        } else {
            action.operate( nap );  // Post action for normal and exceptional exits
            ItineraryPattern itin = nap.getItinerary().popItineraryPattern();
            if (itin !=null) {
                nap.getItinerary().setItineraryPattern( itin );
                itin.go( nap );
            }
        }
    }
}
```

Assume \(P\) and \(Q\) are two itinerary patterns. We define three primitive composite operators \(\text{seq}, \text{alt}, \text{par}\) over the \(P\) and \(Q\) patterns for constructions of sequential, alternative, and parallel patterns. Specifically,

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

```java
public abstract class CompositePattern implements ItineraryPattern {
    private ArrayList path;   // A composite pattern comprises of an array of patterns
    protected Operable action;  // Pre and Post visit actions
    protected ItineraryIterator iter; // Locally defined serializable iterator over the array of patterns

    public abstract void go( Naplet nap ) throw UnableDispatchException;
}
```
class IteratorImpl implements ItineraryIterator {
    Integer cursor;
    public IteratorImpl() {
        cursor = new Integer(0);
    }
    public synchronized Object clone() { … } // Deep clone
    public boolean hasNext() {
        return cursor.intValue() < path.size();
    }
    public ItineraryPattern next() {
        try {
            int cur = cursor.intValue();
            ItineraryPattern next = (ItineraryPattern) path.get(cur);
            cursor = new Integer( cur+1 );
            return next;
        } catch (IndexOutOfBoundsException e) {
            throw new NoSuchElementException();
        }
    }
}

public class SeqPattern extends CompositePattern {
    ...
    public void go( Naplet nap ) throw UnableDispatchException {
        if (iter.hasNext() ) {
            ItineraryPattern next = iter.next();
            nap.getItinerary().pushItineraryPattern( this );
            nap.getItinerary().setItineraryPattern( next );
            next.go( nap );
        } else {
            action.operate( nap );
            ItineraryPattern itin = nap.getItinerary().popItineraryPattern();
            if ( itin != null ) {
                nap.getItinerary().setItineraryPattern( itin );
                itin.go( nap );
            }
        }
    }
}

public class AltPattern extends CompositePattern {
    ...
    public void go( Naplet nap ) throw UnableDispatchException {
        ...
if (iter.hasNext() )
    ItineraryPattern next = iter.next();
}

public class ParPattern extends CompositePattern {
...
    public void go( Naplet nap ) throw UnableDispatchException {
        if (iter.hasNext() ) {
            // Clone naplets for each pattern in ParPattern
            // Dispatch the naplets
            for (int i=0; iter.hasNext(); i++) {
                ItineraryPattern next = iter.next();
                newNap[i].getItinerary().pushItineraryPattern(
                    nap.getItinerary().getItineraryPattern() );
                newNap[i].getItinerary().setItineraryPattern( next );
                next.go( newNap[i] );
            }
            action.operate( nap );
        } else {
            ItineraryPattern itin = nap.getItinerary().popItineraryPattern();
            if ( itin != null) {
                nap.getItinerary().setItineraryPattern( itin );
                itin.go( nap );
            }
        }
    }
}

Formally, the itinerary pattern P is defined in BNF syntax as

\[
<V> ::= <S> | <S; T> | <CÆS; T>
\]

\[
<ItineraryPattern P> ::= \text{Sin}(V) | \text{Seq}(P, P) | \text{Alt}(P, P) | \text{Par}(P, P)
\]

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 \( s_1, s_2, \ldots, s_n \). 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.

```java
class ResultReport implements Operable {
    public void operate ( Naplet nap ) {
        nap.getListener().report( …);  // get Listener and callback via a report method
    }
}
```

```java
class MyItinerary extends Itinerary {
    public MyItinerary( String[] servers ) throws InvalidItineraryException {
```
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 ) throws InvalidItineraryException {
        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 ) throws InvalidItineraryException {
        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 );
    }
}
Example 4: The servers can also be visited in a split-join way. The following code defines an itinerary of “seq( par(s0, s1), s2)” over three servers.

class MyItinerary extends Itinerary {
    public MyItinerary( String[] servers ) throws InvalidItineraryException {
        Operable bar = new Barrier();
        Operable kill = new BarrierTerminate();
        Operable act = new ResultReport();
        ItineraryPattern pip[] = new ItineraryPattern[2];
        pip[0] = new SingletonPattern(servers[0], bar);
        pip[1] = new SingletonPattern(servers[1], kill);
        ItineraryPattern[] sip = new ItineraryPattern[2];
        sip[0] = new ParPattern( pip );
        sip[1] = new SingletonPattern( servers[2] );
        setItineraryPattern( new SeqPattern( sip, act ) );
    }
}

5. Naplet Location and Reliable Communication

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. Note that we distinguish between two types of naplets: long-lived and short-lived in terms of their expected lifetime at each server. For its stability, the naplet tracing and location service is limited to long-lived naplets.

public interface NapletDirectory extends Remote {
    public void register (NapletID nid, URN server, Date time, int event) throws RemoteException;
    public URN lookup( NapletID nid ) throw RemoteException;
}

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.

```java
public interface Locator {
    public URN lookup( NapletID nid );
    public void update( NapletID nid, URN server);
}
```

Post-Office Messaging Service

The Messenger class provides mechanisms for asynchronous and synchronous communications. Asynchronous communication is based on a post-office protocol. 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 server 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.

```java
public interface Messenger {
    public void send(NapletID nid, Message msg) throws NapletCommunicationException;
    public void send(URN server, Message msg) throws NapletCommunicationException;
    public void receive(URN server, Message msg) throws NapletCommunicationException;
}
```

**Synchronous Communication Channel and Channel Handoff**

Messenger provides a NapletSocket interface for synchronous communication between naplets. At first, naplet A creates a NapletSocket object connecting to naplet B. The local Messenger locates the destination via its associated Locator and establishes an actual socket connecting to the destination server. The Messenger reserves a group of port numbers for synchronous communication between naplets. NapletSocket is an extension of Socket and supports the same methods as in Socket.

```java
public class NapletSocket extends Socket{
    public NapletSocket( NapletID nid ) {
        URN server = Messenger.locate( nid );
        int port = Messenger.toPost( … )  // Post a message to request a receiving port number
        super( server.getInetAddress(), port, null, localPort)
    }
}
```

Communication channels are distinguished between persistent and transient. Persistent channels need to be maintained during migration, while transient channels are
not. Consider a communication channel from naplet A to naplet B. If naplet A is to migrate, the channel needs to be closed before its migration because the channel is initiated by naplet A. On the migration of naplet B, naplet server Sb request a communication port in target server, say Sc and then tells naplet server Sa to close the existing channel and re-establish a new channel to Sc. This is transparent to naplet A.

Channel handoff shouldn’t occur until the servers are assured no messages are in transmission. A challenge is how a naplet monitors the status of its naplet sockets. By default the close() method returns immediately, and the system tries to deliver any remaining data. By setting a socket option SO_LINGER, the system is able to set up a zero-linger time. That is, any unsent packets are thrown away when the socket is closed.


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.

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.

Naplet Security Architecture

The naplet security system is based on the JDK1.2 security architecture. Unlike Java’s early sandbox security model, which hard coded security policies together with its enforcement mechanism in a SecurityManager class, the JDK1.2 security separates the mechanism from policies so that users can configure their own flavors of security policies without having to write special programs.
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. The security policy is represented and stored in an ASCII format. It can be configured by a system administrator of the NapletServer. Following is a policy example used in a mobile information collection application.

```ini
grant {
  permission java.net.SocketPermission "*:1024-65535", "connect, accept";
  permission java.io.FilePermission "/home/czxu/WWW/classes/-", "read";
  permission java.lang.RuntimePermission "modifyThreadGroup";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "getProtectedDomain";
}
grant codeBase "file:/home/czxu/naplet/naplet.jar" {
  permission java.security.AllPermission;
}
grant codeBase "file:/home/czxu/naplet" {
  permission java.io.FilePermission "/bin/ls", "execute";
  permission java.security.AllPermission;
}
```

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.

**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.

```java
class NapletThread implements Runnable {
  Naplet nap;
  NapletContext context;
  NapletThread( Naplet nap, NapletContext context ) {
    this.nap = nap;
    this.context = context;
  }
  public void run() {
    nap.init0( context );
  }
}
```
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 the future release.

Resource Management and Access Control

Naplets can do few things without access 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 leave 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.

Notice that the service channel is different from Java built-in pipe facility. Java pipe is symmetric in the sense that both ends rely on each other and the pipe can be destroyed by any party. The service channel is asymmetric in that the service channel can be allocated by the ResourceManager to any alien naplets as long as the service provider is alive.
7. Naplet Management

Naplet management relies on two mechanisms: NapletManager and NapletDirectory. NapletManager provides an interface for users to dispatch naplets and locate their current servers. The manager can also callback, suspend, resume, and terminate its outstanding naplets. The naplet management is realized by sending system messages to interrupt the naplets. The naplet manager also serves as an interface for its outstanding naplets to call back for communication with their home base.

The locally launched naplets aside, the naplet manager also records all alien naplet information in the naplet table. Each entry contains the footprint of a naplet visit, including where it comes from at what time and where it leaves for at what time. The naplet managers work collaboratively in support of naplet tracing and location services.

Another place where naplet running states are stored is the naplet directory. It tracks the location of naplets and supports a centralized way for naplet management.

User Interface

8. Agent-Based Programming

```java
import naplet.*;
import naplet.itinerary.*;
public class ICNaplet extends Naplet {
    public ICNaplet( String[] servers ) throws InvalidNapletException, InvalidItineraryException {
        try {
            Hashtable workLoad = new Hashtable( servers.length );
            NapletState state = new ProtectedNapletState();
            state.set("Workload", workLoad);
            setNapletState( state );
        } catch (NoSuchFieldException nsfe) {
            throw new InvalidNapletException();
        }
        setItinerary( new ICItinerary( servers ) );
    }
    public void onStart() throws InterruptedException {
        String serverName = getNapletContext().getServerURN().getHostName();
        HashMap map = getNapletContext().getServiceChannelList();
        ServiceChannel channel = (ServiceChannel) map.get("naplet.server.examples.load");
        double load = channel.read();   // get workload from service channel
        try {
            Hashtable workLoad = (Hashtable) getNapletState().get("Workload");
            workLoad.put( serverName, new Double(load) );
        } catch (NoSuchFieldException nsfe) {} 
        try {
            getItinerary().travel( this );
        }
    }
}  
```
public void onInterrupt() {};

private class ICItinerary extends Itinerary {
    public ICItinerary(String[] servers) throws InvalidItineraryException {
        setItineraryPattern(new SeqPattern(servers, null));
    }
}

catch (UnableDispatchException ude) {}