Chapter 12

Naplet: A Mobile Agent Approach

Naplet is an experimental mobile agent system for increasingly important network-centric distributed applications. It provides programmers with constructs to create and launch agents and with mechanisms for controlling agent execution. Its distinct features include a structured navigation facility, reliable agent communication mechanism, open resource management policies, and agent-oriented access control. This chapter presents the design and implementation of the Naplet approach and its application in network management.

12.1 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 as its defining trait ability to travel from machine to machine proactively in pursuit of its own agenda on open and distributed systems, carrying its code, data, and running state. The proactive mobility of autonomous agents, particularly their flow of control, leads to a novel distributed processing model on the Internet.

Until recently, mobile agent systems were developed primarily based on script languages like Tcl [137] and Telescript [324]. Recent advance of mobile agent technologies was mostly due to the popularity of Java run-time environment. Java Virtual Machine (JVM) and its dynamic class loading model, coupled with several of other Java features, most importantly serialization, remote method invocation, and reflection, greatly simplify the construction of mobile agent systems. Examples of the Java-based systems include Aglet [185], Ajanta [307], Concordia [76], and D'Agent [137]; see Chapter 11 for a brief review. Readers are also referred to [138, 330] for comprehensive surveys.

Naplet system [334] was started in early 1998, initially designed as an educational package for students to develop understanding of advanced concepts in distributed systems and to gain experience with network-centric mobile computing. Although distributed systems have long been in the core of computer science curriculum, there are few educational software platforms available that are small but full of key concepts in the discipline. A mobile agent system is essentially an agent dock that performs the execution of agents in a confined environment. In addition to mobility

support, the dock system not only needs to protect itself from attacks by malicious or misbehaved agents, but also requires isolating the performance of the agents that are concurrently running in the same server and ensuring reliable communication between collaborative agents while they are moving. Systems with such support serve an ideal middleware platform for experiments with related concepts such as naming, process migration, resource management, reliable communication, and security in distributed systems.

An educational system must also be organized in a way that mechanisms are separated from policies so that various algorithms could be implemented without changes in system infrastructure. We reviewed a number of representative mobile agent systems and found none of them met our needs. Most of the systems available in early 1998 were not open-source code yet, because they were mainly targeted at commercial markets. Naplet should be one of the early systems with open-source code. Since its alpha release in 1998, it has been used by hundreds of students at Wayne State University as a platform for programming laboratories and term projects of advanced distributed systems courses. Over the years, it has also evolved into a research testbed for the study of mobility approaches for scalable Internet services. The system was redesigned in the Spring of 2000.

Like other systems, Naplet provides constructs for agent declaration, confined agent execution, and mechanisms for agent monitoring, control, and communication. It has the following distinct features: structured navigation facility, reliable interagent communication mechanism, open resource management policies, and agent-oriented access control. In the following, we present the Naplet architecture and delineate these features. We refer to naplet as an object of `Naplet` class and Naplet server (or server) as an object of `NapletServer`. We conclude this chapter with an application of the Naplet system in network management. The latest release of the software package is available at `www.cic.eng.wayne.edu/software/naplet.html`.

12.2 Design Goals and Naplet Architecture

Code mobility introduces a new dimension to traditional distributed systems and opens vast opportunities for new location-aware network applications. In a networked world, one is obligated to specify not only how to execute designated tasks of an agent, but also where to execute them. A primary goal of the Naplet system is to support the agent-oriented programming paradigm. It is centered around a first-class object: `Naplet`. It is an abstraction of agents, defining hooks for application-specific functions to be performed on visited servers and itineraries to be followed by the agent.

Agent-oriented programming is supported by an object of `NapletServer`. It defines a dock of naplets and provides naplets with a protected run-time environment

within a JVM. Naplet server was designed with two goals in mind: *microkernel* and *pluggable*.

- Microkernel: To support the execution of naplets in a confined environment, Naplet server must provide an array of mechanisms for agent migration, agent communication, resource management, security control, etc. The server is designed in a microkernel organization. It is represented by a highly modular collection of powerful abstractions, each dealing with different but specific aspects of the mobility support.

- Pluggable: Each module in the Naplet server is made as an external service to the microkernel. Its default implementation can be easily replaced or enhanced with new implementations. Moreover, application services accessible to naplets should also be installed, configured, reconfigured, and removed dynamically without shutting down the server.

This pluggable and microkernel design not only makes coding easier, but also greatly enhances the service scalability, extensibility, and availability. More importantly, it makes it possible for self-installation of a Naplet server on the network.

12.2.1 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() {};
}
```

The naplet identifier (ID) contains the information about who, when, and where the naplet is created. In support for naplet clone, the naplet ID 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 inheritance information and reserve 0 for the originator in a generation. For example, a naplet ID

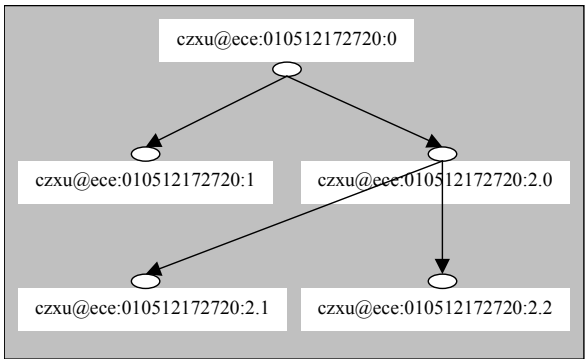

Figure 12.1: Hierarchical naming of naplet ID of a HelloNaplet.

"czxu@ece.wayne.edu:010512172720:2.1/HelloNaplet" represents the hello naplet that was cloned from the original one created by a user "czxu" at 17:27:20 May 12, 2001 in the host "ece.wayne.edu." The inheritance information is shown in Figure 12.1; the naplet name is omitted for brevity.

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 and images in the same package can be zipped into an JAR file so that all the classes and resources the naplet needs are transported at a time.

Note that both the naplet ID and code-base 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 owner's digital signature. The naplet credential is 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 object within the container can be in one of the three protected modes: *private*, *public*, and *protected*. 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.

The naplet executes in a confined environment, defined by its `NapletContext` object. The context object provides references to dispatch proxy, messenger, 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.

In addition to the attributes, `Naplet` class also 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 which must be instantiated by extended agent applications. It serves as a single entry point when a naplet arrives at a host. `onStop` and `onDestroy` are event handlers when respective events occur during the execution of the agent. The agent behavior can also be remotely controlled by its owner via the `onInterrupt` method. Details of these will be discussed in Section 12.2.2.

Mobile agents have as 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 five primitive patterns: singleton, sequence, parallel, alternative, and loop. Complex patterns can be composed recursively. In addition to the way of traveling, itinerary patterns also allow users to specify a postaction after each visit. The postaction mechanism facilitates interagent communication and synchronization. Details about the itinerary mechanism will be discussed in Section 12.3.

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 its home server from time to time. In support of interagent communication, we associate with each naplet an `AddressBook` object. Each address book contains a group of naplet IDs and their original locations. The locations may not be current, but they provide a starting point for tracing. The address book of a naplet can be altered as the naplet grows. It can also be inherited in naplet cloning. We restrict communications between naplets whose IDs are known to each other.

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 postanalysis.

12.2.2 NapletServer Architecture

`NapletServer` is a class that implements a dock of naplets within a JVM. 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. Each JVM can contain more than one naplet server. The Naplet servers are run autonomously and they collectively form an agent sphere for the naplets.

Naplet servers are run autonomously and cooperatively to form a *naplet sphere*, where naplets live in pursuit of their agenda on behalf of their owners. The naplet sphere can be operating in one of the two modes: with and without a naplet directory. The directory tracks the location of naplets; provisioning of the centralized directory service simplifies the task of naplet management. Figure 12.2 presents the Naplet server architecture. It comprises seven major components: NapletMonitor, NapletSecurityManager, ResourceManager, NapletManager, Messenger, Navigator, and Locator.

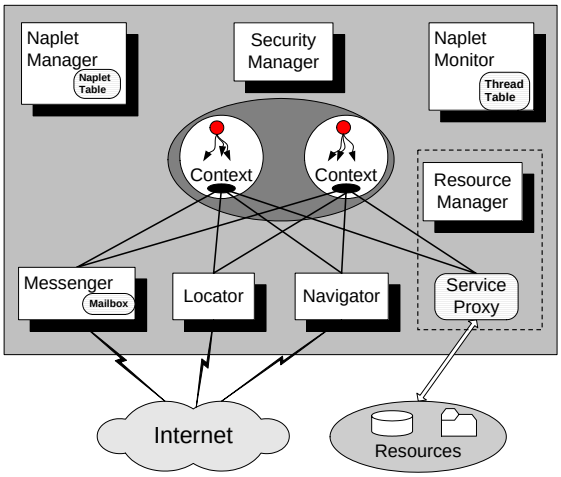

Figure 12.2: The Naplet server architecture.

Each naplet has a home server in the sphere where it is launched through a `Naplet Manager`. The manager provides local users or application programs with an interface to launching naplets, monitoring their execution states, and controlling their behaviors. In addition, the manager 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 its local Naplet manager, the navigator consults a local Naplet security manager for a LAUNCH permission. Then, it contacts its counterpart in the destination Naplet server for a LANDING permission. Success of a launch will release all of the resources occupied by the naplet. Finally, the navigator will also report a DEPART event to a `NapletDirectory`, if it exists. The Naplet directory provides a centralized naplet location service. The directory location is provided to naplet servers when they are installed. The directory service is not necessarily required for naplet tracing. In fact, naplet servers can be run without the presence of the directory service.

On receiving a naplet launch request from a remote server, the navigator consults the Naplet security manager and a locally installed `ResourceManager` to determine whether a LANDING permission should be issued. When the naplet arrives, the navigator reports the arrival event to the manager and possibly registers the event with the Naplet directory. It then passes the control over the naplet to a `NapletMonitor`.

A naplet server can be configured or reconfigured with various hardware, software, and data resources available at its host. The hardware resources like CPU cycles,

memory space, and network I/O 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. Resource manager provides a resource allocation mechanism, and leaves application-specific allocation policy for dynamic reconfiguration.

Naplets access to local services via a `ServiceProxy`. The proxy provides references to local services to visiting naplets and monitors their resource access operations on-the-fly. Local services available to alien naplets can be run in one of the two protection modes: privileged and nonprivileged. Nonprivileged services, like routines in math libraries, are registered in the resource manager as open services and can be called via their handlers as provided by the service proxy. In contrast, privileged services like getting workload information and system performance must be accessed via a `ServiceChannel`. Each channel is a communication link between alien naplets and local restricted privileged services. It is created by the local resource manager on request. After creation of a channel, the manager passes one endpoint to the privileged service and the other endpoint to the service proxy. It is the service proxy that hands off the service channel endpoint to a requesting naplet. Privileged resources are allocated by the resource manager and the access control is done based on naplet credentials in the allocation of service channels.

Each naplet server contains a `Messenger` for internaplet asynchronous persistent 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 application dependent and left for programmers to specify. The interrupt handler is given in method `onInterrupt` when a naplet is created. On receiving a user message, the messenger puts it into a mailbox associated with the receiving naplet. The naplet decides when to retrieve the message from its mailbox.

The messenger relies on a `Locator` for naplet tracing and location services and supports location-independent communication. Naplet ID-based message addresses are resolved through a centralized or distributed naplet directory service. Due to the mobility nature of naplets and network communication delay, the location information provided by the directory service may not be current. The messenger provides a postoffice mechanism to handle messages passing between mobile naplets. Section 12.5.1 gives details of the mechanism.

In addition to support for asynchronous communication, each naplet server provides a `NapletSocket` mechanism for a complementary synchronous transient communication between naplets. `NapletSocket` bears much resemblance to Java Socket in APIs, except it is naplet oriented. Conventional TCP has no support for mobility. To guarantee message delivery, an established socket connection must migrate with naplets continuously and transparently. Section 12.5.2 gives `NapletSocket` APIs. Details of the mechanism will be given in Chapter 15.

12.3 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.

Itinerary representation is a major undertaking of mobile agent systems and various itinerary programming constructs were developed. For examples, Mole provided sequence, set, and alternative constructs, as well as a priority assignment facility in support of flexible travel plans [302]. Ajanta implemented two additional constructs: split, split–join, and loop [307]. They demonstrated the programmability and expressiveness of the constructs mainly by examples. In this section, we define five core structural constructs in the Naplet system: *singleton*, *sequence*, *parallel*, *alternative*, and *loop*. Since each itinerary pattern is associated with a precondition and a postaction, the parallel construct provides flexible support for set, split, and split–join itinerary patterns. In Chapter 13, we extend the core itinerary constructs into a general-purpose mobile agent itinerary language, namely, MAIL. We analyze its expressiveness based on its operational semantics and show MAIL is amenable to formal methods to reason correctness and safety properties regarding mobility.

12.3.1 Primitive Itinerary Constructs

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 agent movement-dependent operations (T) for possible interagent 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 represents the guard 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 a single visit or conditional visit. Assume P and Q are two itinerary patterns. We define four primitive composite operators seq, alt, and par over the P and Q patterns for constructions of sequential, alternative, and parallel patterns. Specifically,

- `seq(P, Q)` refers to a pattern that the visits of P are followed by the visits of Q 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;
- `alt(P,Q)` refers to a pattern that either the visits of P or Q are carried out by one naplet;
- `loop(C → P)` refers to a pattern that the visits of P are repeated until the guard condition C becomes false.

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

$$< V > ::= < S > | < S; T > | < C \rightarrow S; T >$$
$$< P > ::= singleton(V) \mid seq(P, Q) \mid par(P, Q) \mid alt(P, Q) \mid loop(C \rightarrow P)$$

12.3.2 Itinerary Programming Interfaces

The abstraction of itinerary pattern, guard condition, and postaction are expressed in the following public programming interfaces in the Naplet system, respectively.

```
public interface ItineraryPattern extends Serializable, Cloneable {
    public void go(Naplet nap) throws UnableDispatchException;
}
public interface Checkable extends Serializable, Cloneable {
    public boolean check(Naplet nap);
}
public interface Operable extends Serializable, Cloneable {
    public void operate(Naplet nap);
}
```

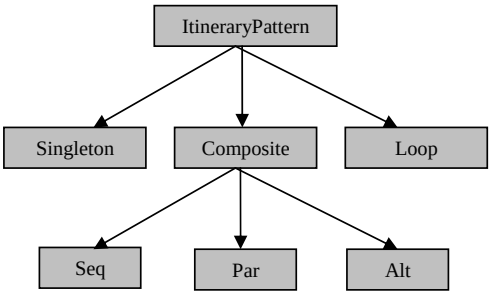

Figure 12.3: Built-in itinerary patterns in the Naplet system.

The Naplet system contains five built-in `ItineraryPattern` implementations: `Singleton`, `SeqPattern`, `AltPattern`, `ParPattern`, and `Loop`. Their class diagrams are shown in Figure 12.3.

In the following, we give two itinerary pattern examples constructed from visits and conditional visits to demonstrate itinerary programming. Consider a mobile agent-based information collection application. One or more agents can be used to collect information from a group of servers $s_1, s_2, \ldots, s_n$ in sequence or in parallel. At each server, the agents perform information-gathering operation (e.g., workload measurement, system configuration diagnosis, etc.), as defined by the application. They are followed by itinerary-dependent operations for possible interagent communication and exception handling. Different itineraries would lead to different communication patterns. In the case of a parallel search, naplets need to communicate with each other about their latest search results. Success of the search by a naplet may need to terminate the execution of the others.

Example 1: The class `MyItinerary1` defines a `SeqPattern` for `MyNaplet`, indicating a sequential information collection. We define `MyNaplet` class as an extension of the base class `Naplet` (line 1). The method `onStart` (line 3) is one of the hooks of the `Naplet` class for application-specific functions to be performed on agent arrival at a server. It contains a location-aware business logic `collectInfo` (line 4). After completion of this function, the agent travels according to its itinerary (line 6). The itinerary is defined in a private class `MyItinerary1` (line 10). It is a simple sequential visiting pattern over an array of servers (line 13). At the end of its itinerary, the agent reports its collected results back to its home by a postaction as defined in the class `ResultReport` (line 9). Since the itinerary class `MyItinerary1` is declared as a private inner class of the naplet, the postaction can be defined on the naplet states. The itinerary is set via a `setItinerary` method of the `Naplet` class (line 13).

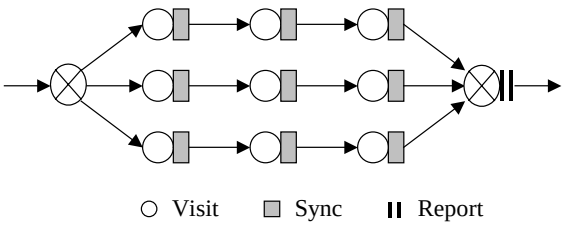

Figure 12.4: An example of parallel search itinerary using three cloned naplets.

```
1)  public MyNaplet extends Naplet {
2)      ......
3)      public void onStart() {
4)          collectInfo();      // Locaion-aware business logic
5)          try {
6)              getItinerary().travel( this );
7)          } catch (UnableDispatchException nde) {};
8)      }
9)      private class ResultReport implements Operable {...}
10)     private class MyItinerary1 extends Itinerary {
11)         public MyItinerary1(String[] servers) {
12)             Operable act = new ResultReport();
13)             setItinerary(new SeqPattern(servers, act));
14)         }
15)     }
16) }
```

Example 2: The class `MyItinerary2` defines a parallel search pattern, as shown in Figure 12.4, by the use of k cloned naplets, each for an equal number of the servers (for simplicity, we assume n can be divided by k). Let $m = n/k$. Totally $m \times k$ visits are defined (lines from 12 through 22). Each visit is of class `Singleton`, comprising a checkable object `ResultVerify` as its guardian precondition (line 12). Whenever a naplet finds the target, it will skip the rest of its servers and meanwhile inform the others. The synchronization is realized by a collective operation defined in the operable object `DataComm` (line 13). A cloned naplet i, $0 \leq i \leq k$, will visit m servers in sequence, as defined in a `SeqPattern` object `journeys[i]` (line 21). The naplets report their results to their home at the end of their journeys via postaction `ResultReport`. All the journeys together form a `ParPattern` itinerary (line 23). The itinerary object is set via a `setItinerary` method of the base class `Naplet`. Naplet cloning is due to a journey of the `ParPattern` itinerary.

```
10) private class MyItinerary2 extends Itinerary {
11)     public MyItinerary2(String[] servers, int k) {
12)         Checkable guard = new ResultVerify();
13)         Operable sync = new DataComm();
14)         Operable report = new ResultReport();
15)         int n = servers.length; int m = n/k;
16)         Singleton[][] visits = new Singleton[k][m];
17)         SeqPattern[] journeys = new SeqPattern[m];
18)         for (int i=0; i < k; i++) {
19)             for (int j=0; j < m; j++)
20)                 visits[i][j]=new Singleton(guard,servers[i*k+j],sync);
21)             journeys[i]=new SeqPattern(visits[i],report);
22)         }
23)         setItinerary(new ParPattern(journeys));
24)     }
25) }
```

12.3.3 Implementations of Itinerary Patterns

In Chapter 13, we will show that the set of itinerary patterns in Figure 12.3 is regular-completeness in the sense that any itinerary in a regular trace can be constructed based on these primitive patterns. However, they are insufficient to express itineraries like "Visiting site s_1 for x times, followed by visiting of s_2 for the same number of times." In the following, we show the implementation details of `Singlton` and `SeqPattern` as programming examples of user-defined itinerary patterns.

In Naplet system, each customized itinerary associated with a naplet is extended from a serializable `Itinerary` class. The itinerary contains a reference to the current pattern and keeps in a stack the naplet trace for recursive traverse.

```
public class Itinerary implements Serializable, Cloneable {
    private ItineraryPattern cur;    // Current itinerary pattern the naplet is on.
    private Stack patterns;          // Stacked itinerary patterns the naplet was on.
    ......
    protected ItineraryPattern popPattern() { return patterns.pop(); }
    protected void pushPattern(ItineraryPattern itin) { return patterns.push(itin); }
    public final void setCurPattern( ItineraryPattern itin ) { cur = itin; }
    public final void travel( Naplet nap ) { current.go( nap ); }
}
```

The `Singleton` class defines the visit of a single server, coupled with precondition and postaction. The `visited` flag is defined as a type-wrapper class `Boolean`, instead of a primitive `boolean` data type. It is because a boolean variable is allocated in a JVM stack and its value will be lost after the migration of

the `Singleton` object. The `go()` method shows the details of a naplet migration. Recall that the execution of a naplet is confined to environment defined by a `NapletContext`. The context object contains references to navigator, messenger, and other mobility support services in a naplet server. The migration is accomplished by the navigator.

```
public class Singleton implements ItineraryPattern {
    private URN server;             // Site to be visited
    private Checkable guard;        // precondition of the visit
    private Operable action;        // Post-action of the visit
    private Boolean visited;        // Serializable visit status
            ... ...
    public void go( Naplet nap ) throw UnableDispatchException {
        if ( ! visited.booleanValue() ) {
            visited = Boolean( true );
            if ( guard==null || guard.check(nap) )
                nap.getNapletContext().getNavigator().toDispatch( next, nap );
            else { backtrack(nap); }
        } else {
            if (action ≠ null)
                action.operate( nap );      // Post action
            visited = Boolean( false );
            backtrack(nap);
        }
    }
    private void backtrack(Naplet nap) {
        ItineraryPattern itin = nap.getItinerary().popPattern();
        if (itin ≠ null) {
            nap.getItinerary().setCurPattern( itin );
            itin.go( nap );
        }
    }
}
```

The `SeqPattern` class is defined as an extension of a `CompositePattern` class. A `CompositePattern` object contains a collection of itinerary patterns to be visited, together with a precondition and a postaction. The itinerary patterns are stored in an `ArrayList` data structure, indexed by an `ItineraryIterator` object. The iterator is defined as serializable to replace the Java `ArrayList` iterator. It is because the Java built-in iterator is nonserialiable and its index information will be lost after migration. The `CompositePattern` class is defined as an abstract class because it leaves the visiting order of the servers in the `ArrayList` unspecified.

The `SeqPattern` class defines a sequential visit order in its `go()` method. It recursively traverses each itinerary pattern recorded in the agent itinerary until a `Singleton` is reached.

```
public abstract class CompositePattern implements ItineraryPattern {
    private ArrayList path;              Composite itinerary is stored in an array list
    protected Checkable guard;           Terminate condition for the loop
    protected Operable action;           Post action after the loop
    protected ItineraryIterator iter;    A serializable itierator over the array list
    public abstract void go(Naplet nap) throws UnableDipatchException;
    // Inner object defines a custom iterator over the itinerary array list
    class IteratorImpl implements ItineraryIterator { ... }
}

public class SeqPattern extends CompositePattern {
    ......
    public void go( Naplet nap ) throw UnableDispatchException {
        if (iter.hasNext() ) {
            ItineraryPattern next = iter.next();
            if (guard==null || guard.check(nap)) {
                nap.getItinerary().pushPattern( this );
                nap.getItinerary().setCurPattern( next );
                next.go( nap );   // traverse next pattern recursively
            } else { backtrack( nap ); }
        } else {
            if (action ≠ null) action.operate( nap );
            iter.reset();
            backtrack( nap );
        }
    }
}
```

12.4 Naplet Tracking and Location Finding

12.4.1 Mobile Agent Tracking Overview

Mobility is a defining characteristic of mobile agents. Mobility support poses a basic requirement for tracking agents and finding their current locations dynamically. The agent location information is needed not only for home servers to contact their outstanding agents for agent management purposes, but also for interagent communication.

In general, there are three classes of approaches for the tracking and location finding problem: *broadcast*, *location directory*, and *forward pointer*. A broadcast approach sends a location query message to all servers. It is simple in concept and easy to implement if the system supports broadcast in network and transport layers. In large-scale networks with unreliable communication links, reliable broadcasting is

nontrivial by any means. Moreover, any agent location change during the process of broadcasting makes the approach impractical.

A location directory approach is to designate one or more directory servers to keep track of agent locations. The directory service is advertised so that any location query message is directed to the directory. The directory can be organized in a centralized, distributed, or hierarchical way. A centralized organization maintains all location information in a single server. Due to its simplicity in management, this centralized directory approach has been widely used in today's mobile agent prototypes, such as MOA [226], Grasshopper [17], and Aglets [185]. A major drawback of this approach is poor scalability. In highly mobile agent systems, the centralized directory is prone to bottleneck jams.

The hierarchical organization enhances the scalability by deploying a group of directory servers. Each low-level server provides directory services for agents in a region and a high-level server keeps track of the agent regions. This two-level hierarchy can be extended to multilayer hierarchies. Because agents are bound to regional directories dynamically according to their current locations, a challenge with this approach is to ensure location information consistency between the directories when highly mobile agents move across regions. The hierarchical directory approach has never been used in any mobile agent system, because few systems were widely deployed. Stefano and Santoro gave the consistency issue a rigorous treatment [300].

A distributed directory organization maintains the agents' location information in their respective home servers. Whenever an agent migrates, its home server is updated of the new location. This approach was used in OMG's MASIF [75]. Unlike the hierarchical approach that binds agents to different regional directories dynamically, the distributed directory approach binds agents to directories in their home servers statically. This requires that a home server address be retrievable from agent naming.

The third class of agent tracking approach is forward pointer (or path proxies). It relies upon agent footprints left on visited servers to chain them together in a visiting order. In the approach, a location query message will first be sent to the agent home server. The message will then be passed down the agent visiting path. Since the approach requires no updates of agent locations, it incurs no extra overhead in migration. The forward printer approach is used in Mole [29] and Voyager [258].

Finally, we note that the problem of agent tracking bears much resemblance to location finding of a mobile device in a wireless network environment. Both problems are to find mobile entities (logical agents versus physical devices) that are traveling in the network. They differ in a number of aspects. First, physical mobility is often dealt with in network and transport layers, while logical mobility is mainly supported in session layer. Second, physical mobility in wireless networks is characteristic of slow movements in neighbor cells. In contrast, agents can migrate quickly, depending on their business logic to be conducted in each server, and agent migration is lack of locality in service overlay networks. These differences raise different requirements for tracking and location finding algorithms. Broadcasting and distributed directory are widely used in tracking of physical mobility.

12.4.2 Naplet Location Service

The naplet location service interface is defined by `Locator` in the following. It is supported by both location directory and footprint organizations.

```
public interface Locator {
    public URN lookup(NapletID nid) throws NapletLocateException;
    public URN lookup(NapletID nid, long timeout) throws NapletLocateException;
}
```

Recall that `NapletServer` can be running in one of the two modes: with and without naplet directory service. In systems without a directory service, naplets are traced by using naplet footprint information recorded in the naplet manager in each server. `NapletFootPrint`, as defined in the following, contains the source and destination information about each naplet visit.

```
public class NapletFootPrint {
    private URN source;              // Where the naplet comes from
    private Date arrivalTime;
    private URN dest;                // Where the naplet leaves for
    private Date departTime;
}
```

On receiving a location finding request from `Messenger` or other high-level location-independent services, the local `Locator` first checks with its local naplet manager to find out whether the target naplet is in. If not, the `Locator` then retrieves the home server address of the target naplet, as encoded in its `NapletID` object and sends a query message to the home server. The query message is forwarded along the path, starting from the home server, according to the agent footprints left in visited servers until the message catches the target naplet. Note that a query message may arrive at a server before its target because the query message and the naplet may be transferred in different physical routes and the naplet may be blocked in the network. If the query message arrives after the naplet's landing and before its departure, it is responded with the current server location; otherwise, the message needs to be buffered for a certain time period. Readers are referred to Section 12.5.1 for details about in-order message/agent delivery on non-FIFO communication networks.

Since a query message for an agent needs to traverse its whole path, the `lookup` service will be time-out as the agent path stretches out. This is an inherent problem with the forward pointer approach. As a remedy, the Naplet system requires updating the home server with the new location whenever a query is responded. Another solution is to use a forward pointer together with a location directory.

In systems with an installation of `NapletDirectory`, the `Locator` can locate naplets by looking up the directory. Although the location information from the directory may not be current due to the communication delay between a naplet server and the directory, it can be used as a starting point for tracing via the complementary forward pointer approach. Note that we distinguish between two types of naplets:

long-lived and short-lived in terms of their expected lifetime at each server. For 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)
                                throw DirectoryAccessException
    public URN lookup( NapletID nid ) throw RemoteException;
}
```

On launching or receiving a naplet, the `Navigator` component of a naplet server 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). The `NapletDirectory` is currently implemented as a component of the naplet server, although its installation is optional. In fact, it can be realized as a stand-alone Lightweight Directory Access Protocol (LDAP) service. One LDAP server can be installed for each naplet sphere (i.e., a collection of naplet servers), independent of naplet servers. To access the LDAP service, each naplet server must authenticate itself to the service.

As we reviewed in Section 12.4.1, a location directory is not necessarily implemented in a centralized manner. The naplet directory services can be provided collaboratively by the naplet manager of each server. Since each naplet has its own home server and the home information is encoded in `NapletID` objects, the naplet location information of can be maintained in their home managers. Correspondingly, any naplet tracing and location requests are directed to respective home managers.

The naplet location service is demanded by `Messenger` for internaplet communication or by `NapletManager` for naplet management. The location service 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 forward pointers.

12.5 Reliable Agent Communication

12.5.1 PostOffice Messaging Service

The messenger service in a Naplet server supports asynchronous message passing between naplets. The messages are naplet ID oriented and location independent.

A naplet can take messages from a specific naplet, any naplet from a naplet server, or any naplet in the sphere if the message sender is not specified. The asynchrony is realized based on a mailbox mechanism. On receiving a naplet, the messenger creates a mailbox for its subsequent correspondences with other naplets or its home naplet manager. Recall that we distinguish messages into two classes: system message for naplet control and user message for data communication. System messages are delivered to their target naplets immediately via interrupts, while user messages are stored in respective mailboxs for the target naplets to retrieve. For flexibility in communication, each messenger also keeps the mailbox open to its naplet so that the naplet can access the mailbox directly, bypassing the send/receive interface.

```
public interface Messenger {
    public void send(NapletID dest, Message msg) throws NapletCommException;
    public void receive(URN server, Message msg) throws NapletCommException;
    public void receive(Message msg) throws NapletCommException;
    public Mailbox getMailbox(NapletID nid);
}
```

The mailbox-based scheme provides a simple and reliable way for asynchronous communication between naplets. Under the hood is a postoffice delivery protocol inside each messenger that implements message forwarding to deal with agent mobility. Each messenger maintains a `MailboxCabinet` to contain all mailboxes of the residing naplets. In addition, it has a special system mailbox, called *s-box*, to temporarily store undelivered messages.

Assume naplet A residing on server Sa is to communicate with 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 or its home server. Due to the mobility nature of naplets and communication delay, this server information is not necessarily current. Without loss of generality, 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 its naplet manager against its naplet trace and forwards the message to the server to which the naplet moved. The forwarding continues until the message catches up to naplet B, say in server Sc. 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 its naplet manager against its naplet trace and finds no record of naplet B. The

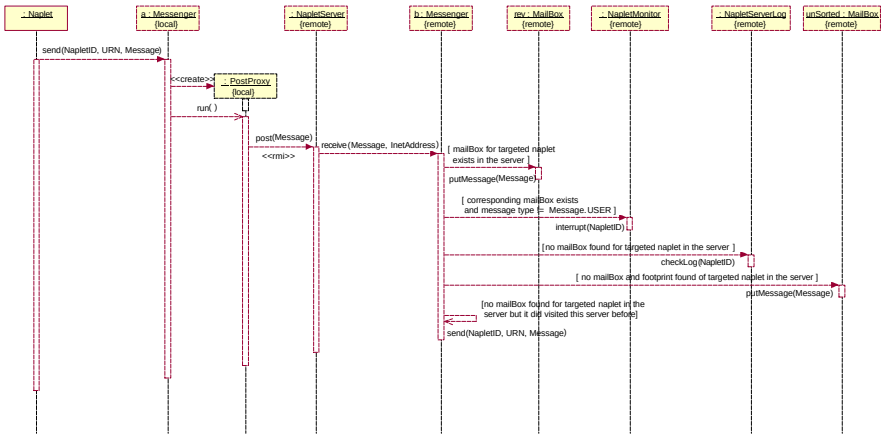

Figure 12.5: Sequence diagram of interagent asynchronous communication.

messenger will insert the message into the *s-box*, waiting for the arrival of naplet B. On receiving the naplet B, Sb's messenger creates a mailbox and transfers the B's messages in the *s-box* to B's mailbox.

12.5.2 NapletSocket for Synchronous Communication

Mailbox-based asynchronous communication aside, Messenger provides a NapletSocket service for synchronous communication between naplets. The NapletSocket service is built on a pair of classes: `NapletSocket` and `NapletServerSocket`. They are implemented as wrappers of Java `Socket` and `ServerSocket`, respectively, providing similar APIs to the Java socket service:

```
public class NapletSocket {
    public NapletSocket(NapletID dest, boolean isPersistent) {}
    public void close() {}
    public InputStream getInputSteam() {}
    public OutputStream getOutputStream() {}
}

public class NapletServerSocket {
    public NapletServerSocket(NapletID dest, boolean isPersistent) {};
    public NapletServerSocket(boolean isPersistent) {};
    public NapletSocket accept() {};
    public void close() {};
}
```

Unlike the Java socket service which is network address oriented, the Naplet socket service is oriented toward location-independent `NapletID`. Assume naplet B runs a server socket and naplet A wants to establish a synchronous communication channel with B. Naplet A creates a `NapletSocket` object connecting to naplet B. The local messenger locates naplet B via its associated Locator and establishes an actual socket connecting to the destination:

```
Naplet B at server side: accept connections from any remote naplets
    NapletServerSocket nss = new NapletServerSocket(true);
    NapletSocket ns = nss.accept();
    InputStream in = ns.getInputStream;
    OutputStream out = ns.getOutputStream;
    ... ...

Naplet A at client side:
    NapletSocket sock = new NapletSocket( B );
    InputStream in = sock.getInputStream();
    OutputStream out = sock.getOutputStream();
    ... ...
```

An established socket can be closed by either side. In addition, the NapletSocket service supports connection migration. We distinguish communication channels between *persistent* and *transient*. Persistent channels need to be maintained during migration, while transient channels are not. Consider the socket between naplet A and naplet B. If naplet A is to migrate and naplet B is stationary, the socket is simply suspended before A's migration and resumed as it arrives at the destination. In the case naplet B is about to leave, the Messenger of naplet B needs to suspend all of its outstanding sockets and inform them of its destination for reconnection. This is accomplished by Messenger, transparently to naplet A.

Channel hand-off 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. Details of the mechanism for synchronous persistent Naplet socket services will be presented in Chapter 15.

12.6 Security and Resource Management

A primary concern in the design and implementation of mobile agent systems is security. Most existing computer security systems are based on an identity assumption. It asserts that whenever a program attempts some action, we can easily identify

a user to whom that action can be attributed. We can also determine whether 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.

12.6.1 Naplet Security Architecture

Over the course of agent execution on a server, server resources are vulnerable to illegitimate access by residing 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 a mobile agent system need to protect agents and servers from any hostile actions from each other. These two security requirements are equally important in an open environment because mobile agents can be authored by anyone and executed on any site that has docking services. However, server protection is more compelling in a coalition environment where the sites are generally cooperative and trustworthy, although mobile codes from different sites may have different levels of trustiness.

The Naplet system assumes naplets are run on trustworthy servers. Security measures focus on protection of servers from any possible naplet attacks. The Naplet Security Architecture (NSA) is based on the standard Java security manager to prevent untrusted mobile codes from performing unwanted actions. Unlike Java's early sandbox security model, which hard coded security policies together with its enforcement mechanism in a `SecurityManager` class, the Java 2 security architecture separates the mechanism from policies so that users can configure their own security policies without having to write special programs.

A security policy is an access-control matrix that says what system resources can be accessed, in what manner, and under what circumstances. Specifically, it maps a set of characteristic features of naplets to a set of access permissions granted to the naplets. It can be configured by a naplet server administrator. It is our belief that any naplet server should be prepared to run an overwhelming number of alien agents from different places. It is cumbersome, if not impossible, to manage the security needs of each individual agent. NSA supports the concept of agent group. A group of agents represent a collection of agents that share certain common properties. For example, cloned agents belong to a group naturally which should be granted similar access permission; agents from the same owner, organization, or geographical region may form a group that shares the same access control policies. Moreover, agents can also be grouped in terms of their functionalities/responsibilities or particular resources that the agents need to access. Such a group is often referred to an agent role. Administrator, anonymous agents, and normal agents are examples. NSA defines security policies for agents as well as groups and roles. Following is a policy example that grants agents from an Internet domain "`ece.wayne.edu`" to look up a yellow page service.

grant Principal NapletPrincipal "ece.wayne.edu/*" {

```
    permission NapletServicePermission("yellow-page", "lookup");
}
```

Early Java security architecture was targeted at code source. That is, authorization is based on where the code in execution comes from, regardless of the subject of code execution. Subject-based access control is not supported until JDK 1.2. `Naplet` is one of the primary subjects we defined in NSA. Other subjects include `Administrator` and `NapletOwner`. Their authentication is based on a Username/Password LoginModule defined in Java Authentication and Authorization Service (JAAS). In contrast, a naplet is authenticated by the use of its carried certificate. The certificate is issued by the home Naplet manager on behalf of its owner.

The agent-oriented access control is realized via an array of additional security permissions: `NapletServicePermission`, `NapletRuntimePermission`, and `NapletSocketPermission`. They grant access control privileges to system resources as well as application-level services in a flexible and secure manner. Details of the access control model will be presented in Chapter 14.

12.6.2 Resource Management

NSA supports policy-driven and permission-based access control to prevent visiting agents from illegitimate access to local services of a server. It leaves monitoring of the naplet execution and control of resource consumption to naplet monitor and resource manager components of the naplet server.

12.6.2.1 NapletMonitor

A critical type of resource to be monitored is CPU cycles. A mobile agent system without appropriate resource management is vulnerable to denial-of-service attacks. The objective of NapletMonitor is to schedule the execution of multiple agents in a fair-share manner. The Naplet system supports migration and remote execution of multithreaded naplets.

On receiving a naplet, the monitor creates a `NapletThread` and a thread group for the execution of the naplet. The `NapletThread` object assigns a run-time context to each naplet thread and sets 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 running states of each thread group and information about consumed system resources including CPU time, memory size, and network bandwidth. It schedules the execution of all residing naplets according to different fair-share resource management policies.

Note that thread scheduling is one of the troublesome problems in Java because it is closely dependent on the underlying scheduling strategies in operation systems. JVM uses fixed-priority scheduling algorithms to decide the thread execution order. If more than one thread exists with the same priority, JVM would switch between the threads in a round robin fashion, if the underlying operating system uses time-slicing

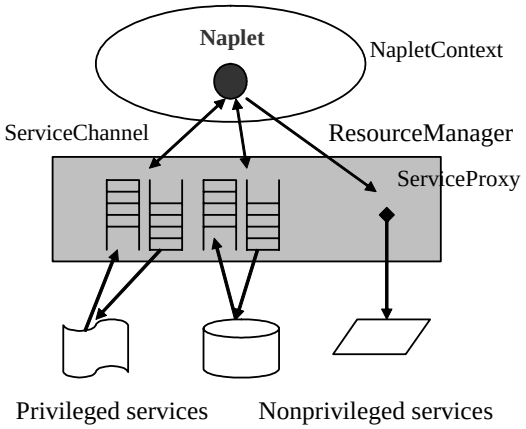

Figure 12.6: Access control over privileged and nonprivileged service.

scheduling. For example, JVM on MS Windows supports fair-share scheduling between threads with the same priority; in contrast, a Java thread on Sun Solaris would continue to run until it terminates, gets blocked, or is preempted by a thread of higher priority. Because of this platform-dependent effect, thread priority is unreliable to support fair-share scheduling. In fact, it is recommended by Java language specification to be used as guides to efficiency only.

Thread priority aside, another performance factor of multithreaded agents is the number of threads. Since JVM cannot distinguish threads from different naplets, a malicious agent can block the execution of other agents by spawning a large number of threads. To ensure fair-share scheduling between naplets, `NapletMonitor` needs to implement a scheduling policy to isolate the performance of naplets, regardless of their priorities and number of threads. The current system release provides the monitoring and control mechanism, leaving scheduling policies undefined.

12.6.2.2 Access to Local Services

Naplets can do few things without access to local services installed on servers and external to the Naplet system. The services include those provided by local operating systems, database management, and other user-level applications. They may be implemented in legacy codes and most likely run in a privileged mode. Although such local services can open to visiting naplets by setting appropriate permissions in `NapletSecurityManager`, visiting naplets should not be allowed to access these services directly. To prevent any threats from misbehaved naplets, resource access operations must be monitored all the time. This is realized by the use of `ServiceProxy` and `ServiceChannel` objects inside the resource manager, as shown in Figure 12.6.

The `ServiceChannel` class defines a bidirectional communication channel be-

tween local services and accessing naplets. Each channel is created by the resource manager and attached to a local service by assigning a pair of input/output endpoints: `ServiceInputStream` and `ServiceOutputStream` to the local service. The other pair of endpoints: `NapletOutputStream` and `NapletInputStream` are left open. The open endpoints will be assigned to a visiting naplet after its service access permission is granted. Once the naplet receives the endpoints, it can start to communicate with the local service under the auspices of the proxy.

Note that the service channel is essentially a synchronous pipe. But it is different from a 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. In contrast, a service channel is asymmetric in that the channel can be allocated by the service proxy to any authorized naplet as long as the service provider is alive. The asymmetry of service channels enables dynamic installation and reconfiguration of application services.

Local privileged services are accessed via service channels. A naplet server may also be configured with nonprivileged services. Examples are small utility services and math libraries. Nonprivileged services are published with access handlers (e.g., math library function calls). It is also the responsibility of ServiceProxy to allocate the handlers to requesting naplets.

12.7 Programming for Network Management in Naplet

Agent-based mobile computing has been experimented with in various applications, such as distributed information retrieval, high performance distributed computing, network management, and e-commerce; see [37, 43, 236, 164, 176, 340] for examples. It provides a number of advantages over conventional distributed computing paradigms [149, 186]. For example, in the client/server model, clients are limited to services that are predefined in a server. Agent-based mobile computing overcomes this limitation by allowing an agent to migrate carrying its own new service implementation. Due to its unique property of proactive mobility, a featured agent should also be able to find necessary services and information in an open and distributed environment. Other advantages include low network bandwidth due to its on-site computation property, resumed execution after a disconnection from the network is reconnected, ability to clone itself to perform parallel computation, and migration for performance scalability or fault tolerance.

In this section, we illustrate Naplet programming in a network management application. Network management involves monitoring and controlling the devices connected in a network by collecting and analyzing data from the devices. Conventional network management is mostly based on the Simple Network Management Protocol (SNMP). It assumes a centralized management architecture and works in a client/server paradigm. SNMP daemon processes (i.e., SNMP agents) reside on network devices and act like servers. They communicate on request device data to

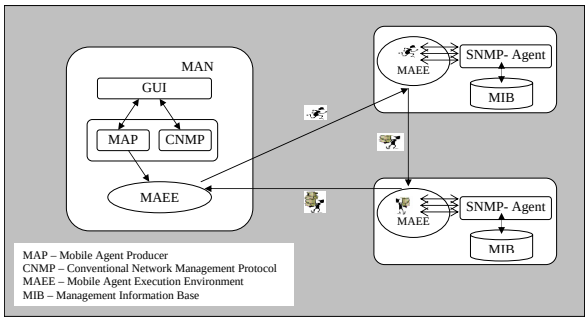

Figure 12.7: Architecture for mobile agent-based network management.

a network management station. The device data are stored in a Management Information Base (MIB) and accessible to local SNMP agents. The management station requests remote MIB information through a pair of fine-grained `get` and `set` operations on primitive parameters in MIBs. This centralized micromanagement approach for large networks tends to generate heavy traffic between the management station and network devices and heavy computational overhead on the management station.

The performance issues in centralized network management architecture can be resolved in many ways. One of the attempts is a mobile agent-based distributed approach. Instead of collecting MIB information from SNMP agents, in this approach, the network management station programs required device statistics or diagnostics functions into an agent and dispatches the agent to the devices for on-site management. Figure 12.7 shows a Naplet-based network management framework. The mobile agent-based framework, namely MAN, is in a hybrid model [176]. It gives the manager the flexibility of using mobile agents or SNMP protocol according to the requirements of management activities.

The MAN management system relies on privileged services provided by the local SNMP agent in each device. In the following, we first present an implementation of the privileged services and then define a naplet class for network management.

12.7.1 Privileged Service for Naplet Access to MIB

In the MAN framework, a network management station creates naplets and dispatches them to 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 AdvenNet SNMP package on each managed device. The AdventNet SNMP packages provide a Java API for network management.

Following is a `NetManagement` class extended from a `PrivilegedService` base class. It is instantiated by a resource manager and associated with a pair of `ServiceReader` and `ServiceWriter` channels: in and out. Through the input channel, the naplet server gets input parameters from naplets and reorganize them

into an AdventNet SNMP format (lines from 6 to 10). It then conducts a sequence of operations, as shown in private `retrieve` method, to communicate with the AdventNet SNMP for required MIB information. The information is returned to the naplet through the out channel (line 12). The whole process can be repeated for a number of inquiries from the same naplet or different naplets.

```
1)  import naplet.*;
2)  import com.adventnet.snmp.beans.*;
3)  public class NetManagement extends PrivilegedService {
4)      public void run() {
5)          for (;;) {
6)              String parms = in.readLine();
7)              Vector values = new Vector();
8)              StringTokenizer paramTokenizer = new StringTokenizer(parms,";");
9)              while ( paramTokenizer.hasMoreElements() )
10)                 values.addElement( (String)paramTokenizer.nextToken() );
11)             String result = retrieve( values );
12)             out.writeLine( result );
13)         }
14)     }
15)     private String retrieve( Vector parameters ) {
16)         StringBuffer result = new StringBuffer();
17)         String result = null;
18)         SnmpTarget target = new SnmpTarget();       // Create an enquiry SNMP target
19)         target.loadMibs("RFC1213-MIB");              // Load MIB
20)         target.setTargetHost(InetAddress.getLocalHost());     //Set the SNMP target host
21)         target.setCommunity("public");
22)         Enumeration enum = parameters.elements();
23)         while(enum.hasMoreElements()) {
24)             target.setObjectID((String)enum.nextElement()+".0");
25)             result = target.snmpGet();   // Issue an SNMP get request on managed node
26)         }
27)         return result.toString();
28)     }
29) }
```

12.7.2 Naplet for Network Management

The privileged service defined in `NetManagement` is dynamically configured during the installation of a naplet server. It is accessed by requesting 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, 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 parallel itinerary pattern shown in lines from 37 to 45.

On arrival at a server, the naplet starts to execute its entry method: `onStart()`. It gets a handler to predefined NetManagement privileged service (lines 16 and 17). It then sends parameters to the server through a `NapletWriter` channel and waits for results from a `NapletReader` channel. Notice that `NapletWriter` and `ServiceReader` are two ends of a data pipe from naplets to servers. Another pipe links a `ServiceWriter` to a `NapletReader`.

When the naplet finishes work on a server, it travels to the next stop (line 27). At the end of its itinerary, the naplet executes an `operate()` method (lines from 30 to 34) to report the results back to its home. Since `NMItinerary` defines a broadcast pattern (lines 40 to 43), the naplet will spawn a child naplet for every server. The spawned naplets will report their results individually.

```
1)  import naplet.*;
2)  import naplet.itinerary.*;
3)  public class NMNaplet extends Naplet {
4)      private String parameters;              // MIB parameters to be accessed
5)      public NMNaplet(String name, String[] servers, String param, NapletListener ch)
                                throws InvalidNapletException, InvalidItineraryException {
6)          super(name, ch);
7)          parameters = param;
8)          setNapletState(new ProtectedNapletState());      // Set space to keep device info.
9)          getNapletState().set("DeviceStatus", new HashTable(servers.length));
10)         setItinerary (new NMItinerary (servers));    // Associate an itinerary with NMNaplet
11)     }
12)     // Entry point of a naplet at each server
13)     public void onStart() throws InterruptedException {
14)         String serverName = getNapletContext().getServerURN().getHostName();
15)         Vector resultVector = new Vector();
16)         HashMap map = getNapletContext().getServiceChannelList();
17)         ServiceChannel channel = map.get("serviceImpl.NetManagement");
18)         NapletWriter out = channel.getNapletWriter();
19)         out.writeLine( parameters );              // Pass parameters to servers
20)         NapletReader in = channel.getNapletReader();
21)         String result = null;
22)         while ( (result = in.readLine()) != EOF ) {
23)             resultVector.addElement( result );
24)         }
25)         Hashtable deviceStatus = (Hashtable) getNapletState().get("DeviceStatus");
26)         deviceStatus.put( serverName, resultVector );
27)         getItinerary().travel( this );
28)     }
29)     private class ResultReport implements Operable {
30)         public void operate( Naplet nap ) {
31)             if ( nap.getListener()!= null ) {
32)                 Hashtable messages = (Hashtable) nap.getNapletState().get("message");
33)                 nap.getListener().report( deviceStatus );
34)             }
35)         }
```

```
36)     }
37)     private class NMItinerary extends Itinerary {
38)         public NMtinerary( String[] servers) throws InvalidItineraryException {
39)             Operable act = new ResultReport();
40)             ItineraryPattern[] ip = new ItineraryPattern[servers.length];
41)             for (int i=0; i<servers.length; i++)
42)                 ip[i] = new SingletonItinerary(servers[i], act);
43)             setRoute( new ParItinerary(ip) );
44)         }
45)     }
46) }
```