Naplet Tutorial

Naplet system is a Java-based secure and flexible mobile agent framework, in support of network-centric distributed applications. It features a structured navigation mechanism, a flexible inter-agent communication facility, and secure interfaces to legacy and privileged Internet services.

The distribution of Naplet system can be found on the site:

http://www.ece.eng.wayne.edu/~czxu/software/naplet.html

This tutorial is for Naplet0.19 version, and aims to help people, who are interested in Naplet system, to install, understand, and utilize the system. It is composed of three parts:

1. **Guide of system installation and test drive of the examples:** provides guidance with respect to Naplet system installations and the execution of the example applications distributed with the system.

2. **Naplet system:** offers an in-depth exploration of Naplet system.

3. **Naplet application examples:** studies representative application examples based on Naplet system.

As their name suggest, these three parts serve people with different primary interest in Naplet system. For readers seeking help of system installation, part 1 would be a good choice. For readers trying to further develop Naplet system, part 2 offers thorough analysis of the system. And for readers interested in developing mobile-agent applications based on Naplet system, part 3 provides first-hand examples for the basic programming skills. The three parts are independent with each other. It is up to readers to identify their goals for using this tutorial, and to choose the appropriate part to continue.

Finally, no matter which part you select, we welcome you to the Naplet world. Enjoy your trip and have fun.

1. Guide of system installation and test drive of the examples

1.1 Naplet System Installation

Distributed in "jar" file format, Naplet system is a free open-source software. Its latest version is released on site www.ece.eng.wayne.edu/~czxu/software/naplet.html.

This part of the tutorial describes how to set up a p recompiled version of Naplet system and customize environmental variables to execute the application examples packed with the system. The installation starts from unpacking the "jar" package.

Naplet system can be installed in both Unix/Linux and Windows environment. If the OS does have effect on the system installation in some point, they are addressed separately. However, when things come to the directory path, we use all backward slash, which is the convention of Windows OS. Users have to substitute it with forward slash if his/her working environment is Unix/Linux.

- Preliminaries

  Since Naplet system is developed in Java language, jdk or any other forms of JVM should be installed to set up Naplet system. Java run time environment could be downloaded in the official web site of java language: www.java.sun.com. Try to use their search function for your convenience.

- Install Naplet system

  To install Naplet system, simply unpack all the files into the directory of your choice, but note that you might encounter minor problems if there is a space anywhere in the directory name.

  After unpacking the package, there should be four subdirectories altogether: "classes", "docs", "examples" and "src". The "classes" subdirectory contains the compiled source file. The "docs" subdirectory includes a naplet overview article and all the HTML files of Naplet generated by Javadoc documentation tool. The "examples" subdirectory holds naplet application examples. And the "src" subdirectory contains the source files of Naplet system.

- Customize the system to execute examples

  In the subdirectory "examples", there are a number of mobile-agent application examples based on Naplet system for users to test drive. These examples aim to display the basic programming techniques in Naplet system. These techniques are elaborated in another part of this tutorial: Naplet Application Example.

Under some necessary customizations, you may execute these examples in your own computing environment, or develop new applications by following these examples. The customizations here mainly related to three environment variables: $(NAPLET_SHARE), $(NAPLET_APP_ROOT), and $(NAPLET_DEV_ROOT).

- $(NAPLET_SHARE): points to a directory where mobile code and RMI stub and skeleton are stored. Before you start testing naplet application examples, you have to assign a directory to $(NAPLET_SHARE). Basically, any directory works. However, please make sure that the directory you specified is readable, since RMI may remotely read the files in this directory.
- $(NAPLET_APP_ROOT): denotes the absolute path of the "examples" subdirectory. For example, if you unpack the Naplet package in directory "e:" or "\home", $(NAPLET_APP_ROOT) would equal to "e:\Naplet\examples" or "\home\Naplet\examples", respectively.
- $(NAPLET_DEV_ROOT): represents the absolute path of Naplet system. For example, if the directory you choose to unpack the "jar" file of Naplet system is "e:" or "\home", $(NAPLET_DEV_ROOT) would be "e:\Naplet" or "\home\Naplet", respectively.

Please follow the following steps to do the customizations:

1. In Unix/Linux OS, according to your choice, set the above three environment variables the corresponding values in ".cshrc" file (refer to the examples above for how-to). For example, if you choose NAPLET_SHARE as "\home\WWW\classes", write "setenv NAPLET_SHARE \home\WWW\classes" in the ".cshrc" file. To make the settings take into effect, source the ".cshrc" file after finishing setting.

   In Windows OS, you need to modify the values of these three variables in two "dmake.bat" files: the first one is in the directory of "examples\naplet-client"; the second one is in the directory of "examples\naplet-server". Modify the values of above mentioned three variables in these "dmake.bat" files to the corresponding values of your choice.

2. Open the file "java.policy" in the directory of "examples\naplet-server". In the first pair of braces, you need to modify the content within the first pair of quotation marks of the second permission: change the part before "/-" to the value of variable $(NAPLET_SHARE) as your choice. In the second grant, you need to modify the content within the quotation marks: change the part between "file:" and "/naplet.jar" to the value of variable $(NAPLET_APP_ROOT) as your choice. Similarly, for the content of next pair of quotation marks, change the part between "file:" and "/naplet-server" to the value of variable $(NAPLET_APP_ROOT).

3. Open each file with extension name ".cfg" in the directory of examples\naplet-server. Set the value of variable $(CodeBase) to the value of variable $(NAPLET_SHARE).

If you still have difficulties in following the steps above, please refer to the section FAQ for more help.

- Execute the examples

  After customizing the Naplet package to your computing environment, you can start to execute the naplet application examples.

  In order to run Naplet examples, you need to open at least two console windows in your computer: one is for naplet client and the other is for naplet server.

  In the window for the server, please change your directory to "examples/naplet-server" (add necessary path at the beginning to form an absolute path). In this window, please follow the steps below:

  - Build the service by typing "make service" (in UNIX/Linux OS) or "Dmake service" (in Windows OS) in the command line.

  - a) In Unix/Linux OS, install the server by typing "make serverN", where the capital 'N" following "server" denotes the number you assign to this server. For example, if you want the server being installed to be server1, N equals to 1. Note that there is no white space between server and 1. Besides, "serverN" has to have a corresponding "serverN.cfg" file, which is in the directory of "examples/naplet-server".

    b) In Windows OS, installing the server by typing "dmake server serverN.cfg". Similarly, N is a number and serverN should have a corresponding "serverN.cfg" file.

  If everything above goes successfully, on the screen there should appear a line like "cic27:2099/NS1 is installed"(yours may be different at the server name part -- "cic27:2099/NS1" in this case, depending on your computer's name and the value of "N" you typed in the command line). This line indicates that a naplet server has been successfully installed.

  Here we want to give some explanation about the server name, since we will need the knowledge when we run the examples later. A full name of the naplet server is in the format: "HostName:port/NapletServerName". Take the instance of "cic27:2099/NS1", of which "cic27" is the name of the host where the naplet server is installed; "2099" is the port number that is specified for the variable "ServerPort" in file "serverN.cfg" under the directory of "examples/naplet-server". The naplet is launched through this port from naplet server to naplet server; "NS1" is the name of the naplet server, which is assigned to variable "ServerName" in file "serverN.cfg", where the value of N depends on the user choice. In the case that several naplet servers are installed on the same machine, the name of the naplet server can be in a simplified version as "/NapletServerName". In the example of "cic27:2099/NS1", "/NS1" could be used to denote the server.

Please note that Naplet system support multiple naplet server installation, as well as single naplet server installation per machine. In most cases, users can install multiple naplet servers in one machine by opening multiple console windows and following the previous procedure. Though be careful to server different "serverN.cfg" files, when installing servers in different windows. For applications making use of NapletSocket, however, the system requires single naplet server installation. In such situation, users can simply deploy the file "server.cfg" when installing the naplet server. Further, for NapletSocket applications, users need to turn on SocketControllerPort and ControlChannelPort, which are specified for NapletSocket in file "server.cfg": be sure to delete the comment mark at the beginning of tow lines with these two ports' names, otherwise exceptions will be thrown if users try to execute the applications containing NapletSocket.

After the naplet server is installed, back to the console window for the client. In this window, please change your directory to "examples/naplet-client" (add necessary path at the beginning to form an absolute path). There are a number of subdirectories in this directory, each of which contains the source files of a mobile agent application example based on Naplet system. To illustrate how to run these examples, here we use "example_name" to denote the names of these subdirectories.

- To compile the example: in Unix/Linux OS, type "make example_name" in the command line, while in Windows OS, type "dmake example_name.make", instead. For instance, if you intend to compile example "hello", either "make hello" or "dmake hello.make" will be your choice.
- To execute the example: in Unix/Linux OS, type "make example_name.run" in the command line. The servers to visit are listed on the top part of the "Makefile" in directory "examples/naplet-client", under the variable of "SERVER_LIST". Users can modified it as needed; in Windows OS, type "dmake example_name.run" followed by the server names that construct the naplet itinerary. Please note that the number of server that each naplet is to visit is specified in the corresponding part of the "dmake.bat" file in directory "examples/naplet-client". Users can refer to the comments to easily modify it. As to how to specify the server name, please refer to the previous paragraph, which explains the naplet server name quite clearly. For example, to run the hello example, it will be either "make hello.run" or "dmake hello.run /NS1 /NS2". In Windows environment, if the number of the server offered in the command line is not exactly the same as what is specified in the "dmake.bat" file, error message will be output on the screen.
- To clean the files generated by compiling the example, simply type "make example_name.clean" or "dmake example_name.clean" will help you in this regard.

In the client window, once a Naplet example is executed, the specified itinerary information of the naplet will be shown. After the naplet finishes visiting servers in its designated itinerary, corresponding results from the Naplet server, if any, will also be

displayed. In the server window, once a naplet arrives or is to be dispatched, the log information will be shown.

Please note that in some examples; more than one naplet server may be needed in a naplet's itinerary. In this case, you have to open more console windows and follow the same procedure above to install a naplet server in each window. Notice that each naplet server has to be assigned a unique server number "N".

In the cases we discussed above, all the naplet servers are physically installed in one machine, although they are assigned different server number and treated as distinct servers. In Naplet system, this is designed for initial program debugging or testing purposes. Whereas, for most mobile agent applications, normally naplet servers are distributed over a network of machines. In the cases like this, a naplet directory is required to be installed with each naplet server such that naplets' location could be registered and traced through the naplet directory in the network. In such situation, except the console windows opened for servers and client as above, there should be another window opened to install a naplet directory in each machine:

1. Change the directory to "examples/naplet-server" in the window for naplet directory.

2. In Unix/Linux OS, install NapletDirectory by typing "make directory"; in Win OS, install NapletDirectory by typing "dmake directory"

If NapletDirectory is successfully installed, on screen there would appear a message "NapletDirectory is bounded in registry".

There is no specific order between naplet server installation and naplet directory installation. However, both of them have to be set up before clients begin to launch naplets.

If you are familiar with UML diagrams, the sequence diagrams for installations of naplet server and naplet directory are available as further explanations to their installation procedure.

If after finish this section, you still have problems about setting up Naplet system, please continue to the next section FAQ. For other related issues, such as uninstalling the system, reporting bugs, please refer to section Other issues.

1.2 FAQ

Q: How to customize the environment variable "NAPLET_SHARE"?

A: The environment variable "NAPLET_SHARE" points to a directory where mobile code and RMI stub and skeleton are stored. Basically, any directory, even a new created directory, will work. However, the directory you choose has to be readable for other users except the owner, since RMI may need to remotely read the files in this directory.

In Unix/Linux OS, command "chmod" will help to extend the readability of directories or files to other users. For instance, if you assign directory "\home\czxu\WWW" to "NAPLET_SHARE", you may need to type command "chmod 711 \home\czxu\WWW" to make sure that other users can also read and execute the files in directory "\home\czxu\WWW".

In Win OS, you need go to windows explorer, select the directory of your choice, right click the mouse and choose the "Properties" option in the popped up menu. In "Properties" dialogue box, choose the "Sharing" tab. Follow the instructions to set the directory as a sharable directory.

Q: How to customize the environment variable "NAPLET_APP_ROOT"?

A: The environment variable "NAPLET_APP_ROOT" denotes the absolute path of the "examples" subdirectory in Naplet system. For example, if you unpack "jar" file of Naplet system package in directory "e:" or "\home\czxu", NAPLET_APP_ROOT would equal to "e:\Naplet\examples" or "\home\czxu\Naplet\examples", respectively.

Q: How to customize the environment variable "NAPLET_DEV_ROOT"?

A: The environment variable "NAPLET_DEV_ROOT" specifies the current directory containing the Naplet package. For example, if the directory you choose to unpack Naplet package is "e:" or "\home\czxu", "NAPLET_DEV_ROOT" would be "e:\Naplet" or "\home\czxu\Naplet", respectively.

Q: Which files need to be modified during customizations?

A: In Unix/Linux OS, three environment variables: "NAPLET_SHARE", "NAPLET_APP_ROOT", and "NAPLET_DEV_ROOT" need to be set in file ".cshrc", which is usually in the root directory of the user. For instance, if you choose "NAPLET_SHARE", "NAPLET_APP_ROOT", and "NAPLET_DEV_ROOT" as "\home\czxu\WWW\classes", "\home\czxu\Naplet\examples", and "\home\czxu\Naplet", respectively, add three lines: "setenv NAPLET_SHARE \home\czxu\WWW\classes", "setenv NAPLET_APP_ROOT \home\czxu\Naplet\examples", and "setenv NAPLET_DEV_ROOT \home\czxu\Naplet" in file ".cshrc". After doing that, remember to type "source .cshrc" in command line to make the setting effective.

In Win OS, these three environment variables are set in two batch files named "dmake.bat" in directory "examples\naplet-client" and "examples\naplet-server", respectively. In these two files, instead of adding lines, you need to modify the values of three variables to the values of your choice.

Following modifications are OS independent.

In the directory "examples\naplet-server", file "java.policy" serves as a java security file for Naplet system. In its first pair of braces, you need to modify the content within the first pair of quotation marks of the second permission: change the part before "/-" to the value of variable "NAPLET_SHARE" as your choice. For instance, if you set "NAPLET_SHARE" as "\home\czxu\WWW\classes", that line would become "permission java.io.FilePermission '\home\czxu\WWW\classes\-' 'read';". In the second grant, you need to modify the content within the quotation marks: change the part between "file:" and "/naplet.jar" to the value of variable "NAPLET_APP_ROOT" as your choice. For examples, if you set "NAPLET_APP_ROOT" as "\home\czxu\Naplet\examples", that line would become "grant codeBase 'file:\home\czxu\Naplet\examples\naplet.jar'". Similarly, for the content of next pair of quotation marks, change the part between "file:" and "/naplet-server" to the value of variable "NAPLET_APP_ROOT". Take the same example as above, the line would become "grant codeBase 'file:\home\czxu\Naplet\examples\naplet-server'", correspondingly.

Furthermore, each file with extension name ".cfg" in the directory "examples\naplet-server" also need to be modified: of those assignment equations, variable "CodeBase" has to be set to the value of variable "NAPLET_SHARE".

Q: Why are there always error information like "cannot create socket for connection" when installing naplet server?

A:Usually these kind of errors are caused by abnormally terminating naplet server processes when naplet applications finish executing last time, such that these processes still run on background and consume the computer network resources. Solution: manually kill the suspicious java zombie processes.

Q: Why is there always "Address already in use" exception when installing the naplet server?

A:This kind of exception is thrown because there is already naplet server installed which consumes the naplet server port. Solution: kill the suspicious java processes.

1.3 Other issues

This section offers some other issues about Naplet system that users may be interested in.

1. Uninstalling

   If you want to uninstall Naplet, simply delete all the files and subdirectories from the directory where it was unpacked, and the files and subdirectories from the directory you choose as "NAPLET_SHARE". Naplet dose not install or update any files in system directories or anywhere else.

2. Troubleshooting

   Unpacking the distribution

   If you encounter troubles in unpacking the "jar" file distribution of Naplet system, there are a number of possible causes: you may not use the command "jar" correctly, or your computer does not support "jar" application. Please refer to the web site developer.java.sun.com/developer/Books/javaprogramming/JAR/basics/ for more help.

   If you believe you have unpacked the distribution correctly and are still encountering problems, refer the section on Further Information below.

3. Installing the naplet server

   If you encounter troubles in installing the naplet server, 99% of the possibility comes from incorrect modifications of environment variables and the corresponding files. Please carefully read the installation guide, making sure that you did everything right.

4. Further information

   As the publish site of Naplet system, www.ece.eng.wayne.edu/~czxu/software/naplet.html releases the latest information of Naplet system. And since Naplet system is a developing system, new version is updated on that site for free download when ready.

5. Reporting bugs

   If you encounter a bug in Naplet system, we would like to hear about it. First check section FAQ to see whether the bug is resulted from incorrect modification of required files. If you decide that it is a bug in Naplet system, send a message to czxu@wayne.edu, describing the bug, the version of Naplet system you are using, and the operation system that you are running on.

2. Naplet system

2.1 What is Naplet System?

As the advanced part of the tutorial, Naplet system is explored in detail in following sections. Basically, the issues addressed here aim to help those whose primary interest is in further developing Naplet system. For readers who are interested in developing mobile agent applications based on Naplet system, we offer another choice in the tutorial: Naplet application examples, where different examples are analyzed to display basic programmatic skills related to naplet applications.

The material in this part is organized as followed:

1. Overview of the Naplet system: provides an introduction of Naplet system.
2. Naplet class: analyzes the architecture of class Naplet.
3. NapletServer architecture: explores the components of class NapletServer, and the interaction between naplets and servers.
4. Structured itinerary mechanism: addresses the logic of itinerary.
5. Naplet tracing and location: presents the naplets' tracing and locating mechanism.
6. Post-office messaging service: examines the messaging service technique.
7. NapletSocket communication mechanism: elaborates the naplet socket communication mechanism.
8. Resource management and access control: discusses the resource management and access control mechanism.
9. NapletMonitor architecture: addresses the control of naplets.
10. Naplet management: discusses the management of naplets.
11. Naplet security architecture: discusses the security concern in Naplet system.

You are suggested to start from the beginning, and follow the order of each section, because sections are arranged in a way that knowledge is delivered from the basic to advanced, from the general to specific.

2.2 Overview of Naplet System

Simply speaking, Naplet system is a mobile-agent based application system. Conceptually, 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 heterogeneously, robust and fault-tolerant. Although none of the individual advantages represents an overwhelming motivation for their adoption, their aggregate advantages facilitate many new network services and applications.

Deploying MA paradigm, Naplet system is developed as 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; meanwhile, naplet servers are running autonomously and collectively form an agent flow space for the naplets.

Class diagram of Naplet package offers an overview of the implementation of Naplet system.

In this tutorial, we refer naplet as an object of Naplet class, and naplet server (or server) as an object of NapletServer.

2.3 What's the architecture of Naplet class?

Naplet is a template class that defines the generic agent. Its primary attributes include a system wide unique immutable identifier *NapletID*, a protected serializable container of application-specific agent running states *NapletState*, and an itinerary *Itinerary* to specify the way naplet travels among servers. The interface of Naplet class is shown as followed:

The interface of Naplet class is as followed:

```
public abstract class Naplet
  implements java.io.Serializable, Cloneable
{
  private NapletID nid;
  private NapletState state = null;
  private NavigationLog navLog;
  private transient NapletContext context = null;
  private NapletListener listener = null;
  private Itinerary itinerary;
  private Boolean longLived = new Boolean(true);
  private AddressBook aBook;

  ...
  public abstract void onStart() throws InterruptedException;
  public void onInterrupt() {};
  public void onStop() throws RemoteException {};
  public void onDestroy() throws RemoteException {};
}
```

- **NapletID**: as the naplet identifier, *NapletID* contains the information about who, when, and where the naplet is created. In support for naplet clone, it also includes version information to distinguish the cloned naplets from each other.

- **NapletState**: 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 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. Details of these will be discussed in the *NapletServer* architecture.

- **NavigationLog**: for naplet management, *NavigationLog* records the arrival and departure time information of the naplet at each server. The *NavigationLog* provides the naplet owner with detailed travel information for post-analysis.

- **NapletContext**: the naplet executes in a confined environment, defined by its *NapletContext* object. The context object provides references to server URN, navigator, messenger, service handler for application services and service channel for privileged services on the server naplet stays. It is a transient attribute and is to be set by a resource

manager on the arrival of the naplet. For its transience, *NapletContext* cannot be serialized for migration.

- **NapletListener**: for applications expecting results from naplets, instead of letting naplets consume the information on the way, *NapletListener* provides a mechanism to transfer the results back to clients.

- **Itinerary**: naplets have their defining trait ability to travel from server to server. Each naplet is associated with an Itinerary object for the way of traveling among the servers. Each itinerary is constructed based on three primitive patterns: singleton, sequence, and parallel. More complex patterns can be composed recursively by combining these three patterns. In addition to the way of traveling, itinerary patterns also allow users to specify a pre-condition before each visit and a post-action after each visit. The pre-condition is main part of the conditional itinerary, while the post-action mechanism facilitates inter-agent communication and synchronization. Details about the itinerary mechanism will be discussed in the *Structured Itinerary Mechanism* section.

- **LongLived**: for a server, a naplet may stay for a long time for its tasks. Meanwhile, it may also leave immediately after arriving without doing anything. Boolean variable *LongLived* is used to distinguish these two cases. From the perspective of servers, a *LongLived* naplet may need some long-term facilities, such as mail box, to be better managed and served, whereas, a non-LongLived naplet does not worthy doing so because of its short life in the server. Currently, for simplicity reason, all naplets are considered LongLived in Naplet system.

- **AddressBook**: 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, each naplet is associated with 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.

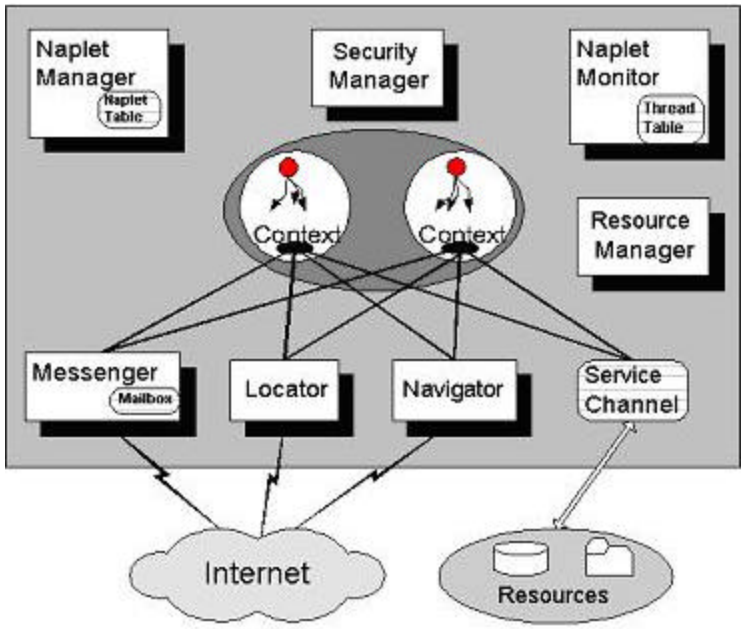
- **Methods**: in addition to the attributes discussed above, Naplet class also provides a number of hooks for application-specific functions to be performed in different stages of an agent life cycle: *onStart()*, *onStop()*, *onDestroy()*, and *onInterrupt()*. *onStart()* is an abstract method which must be instantiated by extended agent applications. It servers a single entry point when a naplet arrives a server. The agent behavior can also be remotely controlled by its creator via *onStop()*, *onDestroy()*, and *onInterrupt()*.

The class diagram of Naplet class provides more information of relationships between Naplet and other components in Naplet system.

2.4 What's the architecture of NapletServer Class?

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.

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. The figure below 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*.

The sequence diagram of Naplet launch provides more detailed information about the interactions and cooperations discussed above.

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

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

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

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

The interface of NapletServer class is as followed:

```
public class NapletServerImpl extends UnicastRemoteObject
    implements NapletServer, ServerProperty, Runnable
{
  private URN serverURN;
  private NapletServerLog log;
  private NapletDirectory directory;
  private NapletManagerImpl manager;
  private NavigatorImpl  navigator;
  private Messenger messenger;
  private LocatorImpl locator;
  private NapletMonitorImpl monitor;
  private ResourceManagerImpl resManager;
  private int codePort;
  private String codebase;

  public NapletServerImpl(Map config)
    throws NapletConfigurationException, RemoteException {}
  public void run() {}
  public void dispatch( NapletDesc desc ) throws RemoteException {}
  public void dispatch( Naplet nap ) throws RemoteException {}
  public void post( Message msg ) throws RemoteException {}
  private InetAddress getSenderInetAddress() throws Exception{}
  public void control(int cmd, Object[] param)
    throws RemoteException {}
  private void initialize(Map config)
    throws NapletConfigurationException, RemoteException {}
  public final URN getServerURN() { }
  public final NapletServerLog getNapletServerLog() {}
  public final NapletDirectory getNapletDirectory() {}
  public final Navigator getNavigator() {}
  public final Messenger getMessenger() {}
  public final Locator getLocator() {}
  public final NapletMonitor getNapletMonitor() {}
  public final ResourceManager getResourceManager() {}
}
```

The class diagram of NapletServer provides a global view of the relationships between NapletServer and other components in Naplet system.

2.5 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.

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

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

Based on the concepts of visit and conditional visit, we define visiting order in recursively constructed journey routing pattern.

Assume P and Q are two itinerary patterns. We define three primitive composite operators "Sin", "Seq" and "Par" over the P and(or) Q patterns for constructions of singleton, sequential, and parallel patterns. Specifically,

- **Sin(P) or Sin(Q)** refers a pattern comprising of a single visit or conditional visit.
- **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.

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

```
<Visit V> ::= <S> | <S; T> | <C'S; T>
<ItineraryPattern P> ::= Sin(V) | Seq(P, P) | Alt(P, P) | Par(P, P)
```

The class diagram of Itinerary provides a global view of the itinerary structure in Naplet system.

2.6 Naplet Location and Tracing

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.

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.

The class diagram of Naplet tracing offers more information of the locating and tracing mechanism in Naplet system.

2.7 Post-Office Messaging Service

The Messenger class provides a mechanism for asynchronous persistent communications for naplets. The 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,

- 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.
- 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.
- if naplet B has not arrived in sever Sb yet (it is possible because the naplet might be temporarily blocked in the network), Sb's Messenger checks with NapletManager against its naplet trace and finds no record of naplet B. The Messenger will insert the message into a special mailbox, waiting for the arrival of naplet B. On receiving the naplet B, Sb's Messenger creates a mailbox and dumps the B's messages in the special mailbox to B's mailbox.

The Messenger interface is shown as followed:

```
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;
}
```

Class diagram of Message and sequence diagram of Post-office messaging service provides more detailed information about the implementation of this service in Naplet system.

2.8 NapletSocket Connection Migration Mechanism

As a complementary mechanism to asynchronous persistent post-office communication, NapletSocket connection provides a synchronous transient communication mechanism between naplets. It builds atop of conventional socket and TCP/IP protocol, featuring a reliable and secure connection migration mechanism for mobile agents.

In Naplet system, NapletSocket connection migration mechanism offers an agent-oriented socket programming interface for location-independent socket communication. Similar to Java Socket, its interface comprises of two classes: *NapletSocket* and *NapletServerSocket*, which resemble Java Socket and ServerSocket in semantics, respectively. For connection migration, the NapletSocket interface also provides two methods *suspend()* and *resume()*. They can be called either by naplets for explicit control over connection migration, or by Naplet Navigator for transparent connection migration.

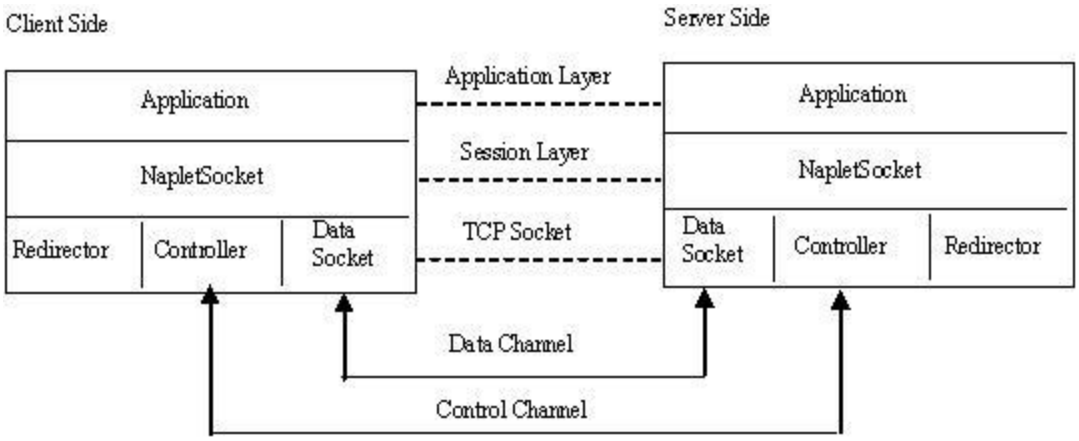


Fig. 1. NapletSocket Architecture

As shown Fig. 1, the NapletSocket is comprised of three components: *data socket*, *controller* and *redirector*, of which, the data socket is the actual channel for data transfer; the controller is for management of connections and operations that need access right to socket resource; and the redirector is to redirect socket connection from a remote naplet to a local resident naplet.

To open a connection, the controller of the client-side naplet sends a request to its counterpart at the server side. If the request is permitted, the client connects to the redirector at the server side and the connection is then handed off to the desired naplet. When a connection is established, its two naplets communicate with each other by accessing the socket, no matter where their communication parties are located. Under the hood is a sequence of operations by the NapletSocket library. The underlying data socket is first closed, when the NapletSocket takes a suspend action before naplet migration. After the naplet lands on the destination, the NapletSocket system resumes the connection by connecting to the redirection server of the peer. The data sockets of both client and server are then replaced by the new socket and new Input/Output stream are re-created atop of the socket accordingly.

The communication protocol of NapletSocket is extended from TCP/IP protocol. In support of the location-independent communication, except three existing states of TCP/IP protocol: *CLOSED*, *LISTEN* and *ESTABLISHED*, additional nine states are added: *CONNECT_SENT*, *CONNECT_RCVD*, *SUS_SENT*, *SUS_RCVD*, *SUSPENDED*, *RES_SENT*, *RES_RCVD*, *CLOSE_SENT* and *CLOSE_RCVD*. A NapletSocket connection is in one of these twelve states. The NapletSocket system takes an action when a certain event occurs, according to the current state of the connection. Generally, there are two types of events: calls from naplets and receipt of messages from its remote peer. Possible actions are to send messages to remote naplets or call other internal services.

Fig. 2 shows the state transitions of NapletSocket. The solid lines show the transitions of clients connecting to servers and the dotted lines are for the servers. Details of the open/suspend/resume/close transactions are as follows.

- *Open a connection*: Both client and server are initially at the **CLOSED** state. When an agent does an active open, a **CONNECT** request is sent to the server and the state of the connection changes to **CONNECT_SENT**. If the request is accepted, the client side NapletSocket receives an **ACK** and a socket ID to identify the connection. Then it sends back its own ID and the state changes to **ESTABLISHED**.

  Connection in server side switches to the **LISTEN** state once an agent does a listen. When a **CONNECT** request comes from a client, the server acknowledges it by sending back an **ACK** and a socket ID. It then changes to the **CONNECT_RCVD** state. After the socket ID of the client side is received, it switches to **ESTABLISHED** and the NapletSocket connection is established. Now the data can be transferred between the two peers as normal socket connection.

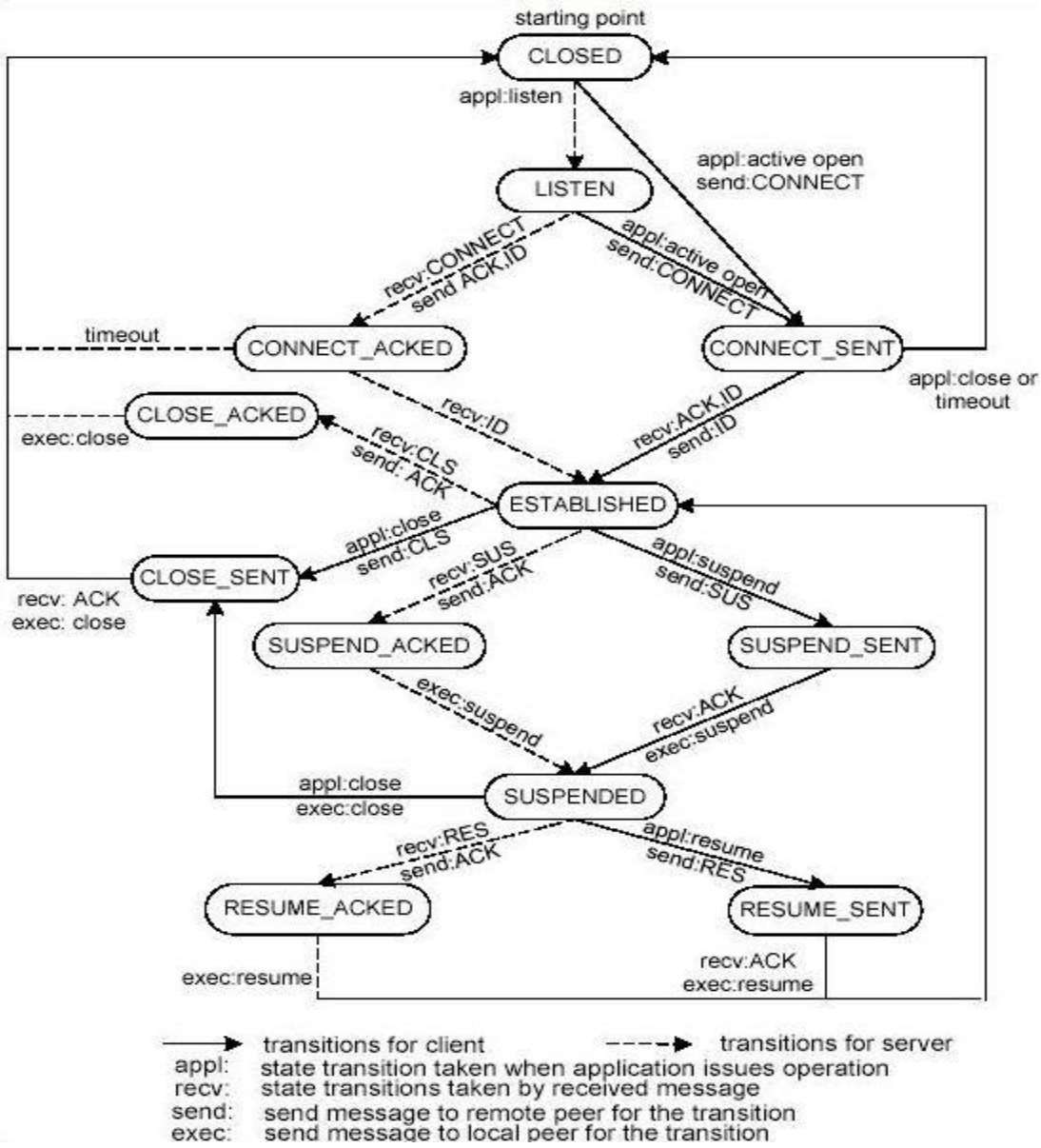


Fig. 2. NapletSocket State Transitions

- *Suspend/Resume a connection*: After a connection is established, either of the two parts may suspend it. The one who issues the request is said to suspend actively and the other is said to suspend passively. At the beginning of active suspending, the one who wants to suspend a connection invokes the suspend interface and a **SUS** message is sent to the peer. If the request is acknowledged, an **ACK** message is sent back and triggers the action of closing underlying input/output streams and data socket. The connection state then switches to **SUSPENDED**.

When the other side of NapletSocket receives the **SUS** message, it sends back an **ACK** message if it agrees to suspend. Then it closes the input/output streams and the data socket under NapletSocket. After that, the state for this peer also changes to **SUSPENDED**. Now connections at both peers are suspended. No data can be exchanged in this state.

At the **SUSPENDED** state, when either of the agents decides to resume the connection, it invokes the resume interface. The resume process first sets up a new connection to the remote redirector and sends a **RES** message. After an **ACK** message is received, it then resumes the connection and the state switches back to **ESTABLISHED**. When resuming the input stream, if there are any data in the buffer received during time of suspending, the input stream of NapletSocket provide data from the buffer. After all data from the buffer are read, it read from input stream of the new connection. For the remote peer in the **SUSPENDED** state, once it receives a resume request, it first sends back an **ACK** message. Then the redirection server hands its connection to the desired NapletSocket and new input/output streams are created. After that, both peers change back to the **ESTABLISHED** state.

- *Close a connection*: In either the **ESTABLISHED** or the **SUSPENDED** state, if an agent decides to close the current connection, it invokes the close interface and the NapletSocket does an active close by sending a **CLS**(CLOSE)} request to the peer. After the request is acknowledged, local data socket is closed. For the other side of the connection, it closes passively after receiving a **CLS** request. It first acknowledges the request and then closes the underlying socket and streams. At the time, data socket at both sides is closed and the state changes to **CLOSED**.

The class diagram of NapletSocket provides detailed information of the NapletSocket structure in Naplet system.

2.9 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 two pairs of input/output endpoints: ServiceReader and ServiceWriter, ServiceInputStream and ServiceOutputStream, to the service, and leave another two pairs of endpoints: NapletReader and NapletWriter, NapletInputStream and NapletOutputStream, to alien naplets. The reader and writer here are designed for character

stream communication, while the inputStream and outputStream for byte stream communication. Besides, for the efficiency of communication, each endpoint of the service channel provides two pairs of buffered stream as wrappers: BufferedServiceReader and BufferedServiceWriter, BufferedServiceInputStream and BufferedServiceOutputStream for the service end, BufferedNapletReader and BufferedNapletWriter, BufferedNapletInputStream and BufferedNapletOutputStream for the alien naplets' end.

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.

Class diagrams of Resource and Service channel offers more detailed information about the implementation of resource management and access control in Naplet system.

2.10 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.

```
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 );
    nap.init();

    try
    {
      nap.onStart();
    }
```

```
      catch ( InterruptedException ie )
      {
        nap.onInterrupt();
      }
    }
  }
```

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.

2.11 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.

2.12 Naplet Security Architecture

The naplet security system is based on the JDK1.2 security architecture. The naplet security behavior is specified by application-specific security policies. A security policy is an access-control matrix that says what system resources can be accessed, in what fashion, and under what circumstances. Specifically, it maps a set of characteristic features of naplets to a set of access permission granted to the naplets. 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.

```
  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 current 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.

3. Naplet application examples

3.1 Overview

This part of the tutorial is concentrated on naplet application example study. It aims to provide readers the basic programmatic skills to develop naplet applications. For those whose primary interest is in developing Naplet system, the tutorial offers another choice: Naplet System

The material in following sections is organized as followed:

1. introduction: a brief introduction of Naplet system
2. hello: a "hello world" example in Naplet system, which mainly shows you how to use application services in servers.
3. rsh: an example simulating "remote share" in mobile agent paradigm. It mainly presents you how to use privileged services in servers.
4. bufferedIO: an example aiming to show the usage of different kinds of IO stream offered by service channel.
5. message: an example aiming to present the post-office messaging service in Naplet system.
6. napletSocket: an example aiming to present the usage of NapletSocket communication in Naplet system.
7. loop: an example aiming to present the Loop itinerary pattern and the concept of conditional itineraries in Naplet system.

In addition, since itinerary is separate from naplets' functionality, itinerary part will be discussed in detail in itinerary section.

We strongly suggest you starting from the beginning, and follow the sections' order while studying, since the knowledge introduced in a new section is, more or less, based on its former ones. By these examples, you are expected to learn the basic programming skills in Naplet system. In other words, how to use the facilities Naplet system provides to develop mobile-agent based applications. As we stated at the beginning, more thorough understanding of Naplet system is out of the discussion scope of these examples. Therefore, after these examples, you will not find answers for questions like how a naplet server operates when a naplet arrives. What's the whole procedure of naplets' launch from server to server? Rather, you will simply taught that Naplet's method "onStart( )" will be executed once naplets arrives in a server, and using method "NapletChannel.launch(naplet)" to launch a naplet. Again, if your interest is in the Naplet system enhancement, please refer to Naplet System, instead.

Please note that the codes to be examined later will involve some advanced features of collections, RMI and networking in Java language. While detailed discussion of these features is beyond this tutorial, they can be found in the official site of Java language www.java.sun.com. Try to use their search service for your convenience.

The examples studied here actually can be found in the distribution of Naplet system. You may run them after you customize the system. As to how to do the customization, please refer to Naplet System Installation Guide.

3.2 Introduction of Naplet System

Naplet system is a Java-based secure and flexible mobile agent framework in support of network-centric distributed applications. It features a structured navigation mechanism, a flexible inter-agent communication facility, and secure interfaces to legacy and privileged Internet services.

Naplet system is centered on two basic concepts: naplet and naplet server. A naplet is an object of class Naplet. Its behaviors are based on the current state of its attributes and in response to the change of its running environment. The primary attributes of a naplet include an identifier, an itinerary, a NapletState object and a NapletContext object. The naplet identifier is system-wide unique such that it represents the naplet in the system. The itinerary specifies a way the servers to be visited. The naplet system provides a structural navigation mechanism so that users can construct a singleton, sequence, parallel, or a composite itinerary easily.

The NapletState object keeps problem-specific intermediate results of the naplet. For example, a naplet for workload information collection records the collected workload in its NapletState object.To protect the NapletState object from unauthorized access by servers, we distinguish the object in three protective types: private, protected, and public. Private state refers to the object that private to naplet only, public state refers to the one that is accessible to any servers; protected state is the information accessible to specific servers.

Naplet runs in a confined naplet server environment, defined by an NapletContext object. This context object provides reference to resources each server offers to facilitate naplet's execution. The resources include message handlers for inter-agent communication, dispatch handlers for naplet migration, and service handlers to services on the server. Since each server may open different resources to naplets, this object is to be set by a resource manager on the arrival of naplets on each server.

Each naplet undergoes a series of stages, from creation through destroy. The behavior of a naplet is defined by methods onStart(), onStop(), onSuspend(), onResume(), and onDestroy(). They are performed in different stages of the life cycle. The naplet behavior can also be controlled by defining a method onInterrupt(). A naplet on the route can be retreated or terminated via this generic method.

A naplet server is an object of class NapletServer. It is a dock of naplets, providing a protected running context for each alien naplet. This context is defined in a NapletContext object, which, in turn, is an attribute of naplets. As mentioned, the context contains references to message handlers for inter-agent communication and dispatch handlers for naplet migration. It also contains handlers to local application services. We distinguish the services in two classes: non-privileged and privileged. The former refers to services that can be invoked by alien naplets directly without special permission grants. The privileged services must be invoked by servers and they cannot be invoked by naplets. The naplet server provides channels for alien naplets to pass arguments to and get results from privileged services.

Communication happens between naplets who know each other. Each naplet maintains an address book which records the IDs of accessible naplets. Messages can be addressed to NapletID. The message handler will check with a Naplet Directory, if exists, to locate the naplet and forward the message to the destination. If no NapletDirectory exists, the message handler will send the message to the server recorded in address book. The naplet server maintains a mailbox for each long-lived naplet for message passing.

3.3 Hello Example

Naplet application examples usually consist of two primary files. One serves as the initiator of the application, which declares and launches naplets in a main program. Another file implements the naplet by extending Naplet class, fueling the naplet functionality and defining a specific itinerary for the naplet. In addition to these two files, depending on whether naplets need to be called back, there will exist the third file, which implements NapletListener interface, for the purpose to transfer information back.

Hello example implements a naplet that travels around the server network, carrying greetings from visited servers. There are three files in this example package: HelloTest.java, HelloListener.java, and HelloNaplet.java. As mentioned above, these three files play initiator, listener, and mobile agent role within the package, respectively.

First, let's take a look at file HelloTest.java, where the main program resides. In class HelloTest, the only method is function main ( ), in which, most of the work are done in the "try" block as shown below.

```
try
{
  HelloListener nal = new HelloListener();
  HelloNaplet na = new HelloNaplet( "HelloNaplet", args, nal );

  NapletChannel.launch( na, 2099 );
}
```

A HelloListener object is first created. And it is binded with a HelloNaplet object as the third parameter of HelloNaplet's constructor. Actually, the first parameter of the constructor is the name of the naplet, and the second one is a list of the names of servers, which the naplet is expected to visit. Please note that the itinerary of the naplet is not specified here. It is done in HelloNaplet.java. After the HelloNaplet object is created, it is launched by the static method "lauch(Naplet, int)" of class NapletChannel. The second int parameter of function "launch( )" is the port number to communicate with the local server for naplet lauch request. Its default value is 2099. Please note that there are a number of exceptions need to be caught after the "try" block.

```
Hashtable messages = ( Hashtable ) result;

for ( Enumeration e = messages.elements(); e.hasMoreElements(); )
{
  String msg = ( String ) e.nextElement();
  System.out.println( msg );
}

try
{
  String host = UnicastRemoteObject.getClientHost();
  InetAddress hostAddress = InetAddress.getByName( host );
  String hostName = hostAddress.getHostName();
  System.out.println( "Message comes from " + hostName );
}
```

Secondly, let's quickly browser file HelloListener.java. Class HelloListener implements interface NapletListener by instantiating method "Report(Object)". In that method, it first prints out the collected information in a for-loop. In the next "try" block, the source of the information is also displayed. Please note that if you are not familiar with those collections, RMI and networking features involved here and you feel these topics are too broad to explore in a short time, simply emulating these codes could temporarily relieve you such that you can continue to the next part, which is also the most important part in this section, HelloNaplet.java.

```
super( name, listener );
try
{
  setNapletState( new ProtectedNapletState() );
  Hashtable messages = new Hashtable( servers.length );
  getNapletState().set( "message", messages );
}
...

setItinerary( new ICItinerary( servers ) );
```

Finally, let's turn to the key component in this example: HelloNaplet.java. By extending the class Naplet, in the constructor, HelloNaplet binds itself a name and a listener, if any. Then it sets a ProtectedNapletState to keep collected information from servers along the path. This information will be stored in a hashtable in the name of "message". And an itinerary is set based on a list of servers the naplet needs to visit.

Now before we go any further, let's first recall an important property of Naplet class: NapletContext. This context object is to be set by a resource manager on the arrival of the naplet on each server. It provides references to server URN, navigator, messenger, service handler for application services, and service channel for privileged services on the server. In the codes to be explored in a moment, we will see how frequently this property is used for those things it refers to. (More detailed information of Naplet class is elaborated in section Architecture of Naplet class in Naplet system part).

```
URN server = getNapletContext().getServerURN();

System.out.println( "Naplet " + this.getNapletID().toString()
                     + " arrives at " + server.toString() );

InfCollection handler = null;
try
{
  handler = ( InfCollection ) getNapletContext().getServiceHandler(
             "serviceImpl.InfCollection" );
  msg = handler.getMsg();
}
```

As an entry point of a naplet on each server, method "onStart( )" defines main functional operations of a naplet. Since HelloNaplet is created for collecting information from visited servers, its "onStart( )" is implemented for this purpose: the several lines before the first "try" block try to display the location information of the naplet along the path. The mechanism behind is very simple. It obtains NapletContext by using the method inherited from Naplet class "getNapletContext( )". Then concatenating it with ".getServerURN( )" will uncover which server the naplet is currently on. Following the same rule, the next "try" block obtains the service handler for application service "InfCollection" by offering the name of the service the naplet is looking for. Please note that the service name actually is a path name, which means that you have to know how to locate the specific service you are looking for in a server while developing the naplet. After that, the naplet obtains the message that the server wants to convey by the service handler. Since the service in this example is an application service, it is allowed to invoke the method "getMsg( )" directly by naplet. For privileged services, however, it is not the case. We will discuss it later in other examples.

```
Hashtable messages
       = ( Hashtable ) getNapletState().get( "message" );
messages.put( server.toString(), msg );
```

After the information is collected, another concern would be how to store this information such that it can be taken back to clients. Remember in the constructor we created a ProtectedNapletState, within which, there is a hashtable named "message" for information storing. Now it comes for service as shown above: information "msg" collected from the server is put in hashtable "message" under the key of server's URN. Since each server has its unique

URN, the hashtable could keep all the information with its associated server in the ProtectedNapletState along the path.

```
getItinerary().travel( this );
```

Once a HelloNaplet obtains and stores information it expects from a server, its task on this server is complete. The next thing to do is to leave for next server according to its itinerary. All these are done by the line shown above. HelloNaplet first obtains its itinerary by using the method "getItinerary( )", which is inherited from Naplet class. Then the method "travel(Naplet)" in Itinerary class will check which server is the next host to visit and launch the naplet to that host somehow.

Upon arriving a new server, the same procedure defined in "onStart( )" repeats again, until the naplet finishes its itinerary.

As you have noticed, except methods, there are also two private classes defined in HelloNaplet: "ResultReport" and "ICItinerary".

```
Hashtable messages
         = ( Hashtable ) nap.getNapletState().get( "message" );
nap.getListener().report( messages );
```

Class "ResultReport" implements the interface Operable, which is designed for post-operations of an itinerary pattern. Simply speaking, the mechanism of such "post-operations" works as followed: each time when naplets finish visiting one server and are about to be launched to next destined server, method "operate(Naplet)" in Operable will be executed. Therefore, by instantiating method "operate(Naplet)", class "ResultReport" defines an operation to be performed at the end of an itinerary: obtain the hashtable "message" which stores the collected information, transfer it back by using method "report(Object)" defined in class HelloNapletListener.

```
Operable act = new ResultReport();

// seq(s0, s1, s2)
setRoute( new SeqPattern( servers, act ) );
```

We kept talking about itinerary. Now let's see how an itinerary is constructed for HelloNaplet. As a private component of HelloNaplet, class "ICItinerary" is an extension of class Itinerary. It inherits every properties and methods from its parent, and defines a constructor of its own. In the constructor, first, an Operable object "act" is created, which is actually "ResultReport" type. A sequential itinerary pattern is created by serving the servers to visit and the action to perform. At last, this sequential pattern is set as the itinerary that class ICItinerary would provide, by using method "setRoute(ItineraryPattern)" inherited from class Itinerary.

To define an operation to be performed at the end of an itinerary actually is not very complex. You just need write a class to implement the interface Operable, instantiating its method "operate(Naplet)" according to the application you are working on. To attach the operation to naplets' itinerary, an "Operable "object needs to be declared in the extended Itinerary class. Setting it as the second parameter when creating basic Itinerary patterns would connect the operation with the itinerary.

Currently, in Naplet system, there are three primitive patterns defined: singleton(SingletonPattern), sequential(SeqPattern), and parallel(ParPattern). Of those, singleton pattern consists of single server to visit. Sequential pattern defines a sequential visit to destined servers, while in parallel pattern, naplets are cloned and dispatched simultaneously to destined servers. More complex patterns could be composed by embedding these three patterns recursively. In this example, only a simple sequential pattern is deployed. More details of itinerary and itinerary patterns are discussed in section Itinerary.

Now we have went through the whole package of "hello". At this moment, after you finished reading all the material above, we assume you can answer the following six questions:

1. How to lauch a naplet? (hint: NapletChannel.lauch(Naplet), getItinerary().travel(Naplet))

2. How to set and get NapletState? (hint: setNapletState(NapletState), getNapletState( ))

3. Which function defines the functional operations of a naplet? (hint: "onStart()")

4. How to get NapletContext? How to use it to obtain the reference of server URN and service handlers for application services? (hint: getNapletContext(), getNapletContext().getServerURN(), getNapletContext().getServiceHandler())

5. How to define an operation to be performed at the end of an itinerary? How to associate it with an itinerary? (hint: implement the interface Operable, and set it as the second parameter when construct the itinerary pattern)

6. How to construct a sequential itinerary for a naplet? (hint: setRoute(new SeqPattern(String[], Operable)))

If you do not have the answers, please go through this section again and try to find them. They will not be repeated in the following examples because there are too many issues in the Naplet system we want to address. Before you go any further, these are the basics we want you to grasp.

However, if you know every answer, congratulations. You are ready to learn more in following sections.

As to how to run the hello example, please refer to the installation section for the details.

3.4 Rsh Example

Before we start this example, we assume you already finished the Hello Example, and mastered those basic knowledge introduced in that example. If not, we strongly suggest you begin from that example, since in this example we only address the different stuff from Hello Example.

Rsh example implements a naplet that asks for the execution results of command "uname -a" from each server. Because this command is only valid in Unix/Linux environment, this example cannot be run in Windows environment.

Structurally, this example is very similar with Hello. There are also three files in the package: RshTest.java, RshListener.java, and RshNaplet.java. They are playing the same roles as they do in Hello: initiator, listener, and mobile agent. Since the architecture of RshTest.java and RshListener.java are almost the same as their counterparts in Hello. We will not go through them here. If you have any question about these two files, please refer to the corresponding parts of Hello Example.

Now Let's go directly to RshNaplet.java. We will pass those constructors since there is nothing new there. However, method "onStart( )" is really worth a close look because it answers the problem we want to discuss in this section: how to access a privileged service in a server? In Hello Example, the service HelloNaplet takes advantage of is an application service. It is obtained through a service handler, and invoked by naplets directly. A privileged service, however, cannot be accessed in such way, since the resources it expose to naplets supposedly are very important for servers. To prevent attacks from malicious naplets, they should be provided in a way that involves servers' supervision. In Naplet system, such supervision is manifested by Service Channel. Simply speaking, service channel acts as an inter media between naplets and privileged services. Programmatically it is implemented as a bi-directional pipe. Whenever a naplet needs to communicate with a service, it informs the service by writing their requests into the pipe, and gets the responce by reading information from the pipe. By contrast, services are always sitting on the other end, trying to read requests, and write the response back after the request is properly handled. Here since we are talking about naplets, how service end operates in service channel is out of our concern. We are only interested in how naplet end operates in service channel.

```
channel = ( ServiceChannel ) getNapletContext().getServiceChannel(
            "serviceImpl.ServiceShell" );
```

As you may see in the codes shown above, similar to obtaining service handlers for application services, service channel is obtained through NapletContext in the same way: get NapletContext by using method "getNapletContext( )", then concatenate it with ".getServiceChannel(String)". Please note that the service name is a path name here.

```
NapletWriter out = channel.getNapletWriter();

char[] cmdChar = CMD.toCharArray();
try
{
  for ( int i = 0; i < cmdChar.length; i++ )
  {
    out.write( cmdChar[i] );
  }
}
finally
{
  out.close();
}
```

After obtaining the service channel of the service "ServiceShell", RshNaplet gets the NapletWriter "out", the writer in the naplet end of the channel, by invoking method "getNapletWriter( )" of the class serviceChannel. Then the command to be executed is written into the channel character by character. Actually, except method "write(char)", NapletWriter offers another method "write(char cbuf[], int off, int len)" for writing massive data at one time, which obviously is more efficient than writing one character at a time. Furthermore, service channel provides buffered IO between naplets and services for even more efficient communication. This will be elaborated in BufferedIO Example. Please note that NapletWriter needs to be explicitly closed after finishing writing. Otherwise, the reader in the service end will be blocked in its "read( )" method, which actually results in a dead lock at last. Here in case any exception happens, which may cause the naplet die before it does the closing, NapletWriter is closed in a "finally" block.

```
NapletReader in = channel.getNapletReader();

int ch;
int i = 0;
while ( ( ch = in.read() ) != -1 )
{
  if ( i < BufSize )
  {
    buffer[i++] = ( char ) ch;
  }
}
```

Following the same procedure, the NapletReader "in" is obtained by invoking method "getNapletReader( )" of the class serviceChannel. RshNaplet then tries to read the responce from the service channel character by character. Similarly, except "read(char)", NapletReader offers another method "read(char cbuf[], int off, int len)" for massive data transmission. Please note that there is no need to close NapletReader after finish reading.

The remaining parts of RshNaplet.java are almost exactly the same as Hello Example. If you have any question, please refer to the corresponding parts there.

As protected resources, privileged services can only be accessed through service channel. Functioning as a pipe between naplets and services, service channel offers each end a reader and a writer for communication purpose. In this example, we talked about at the naplet end, how to obtain the service channel, how to get the reader and writer, and how to write and read data from the channel. By finishing this section, we expect that you have the answer for the following question: how to use privileged services in servers?

Please refer to the installation section for the details about how to run the rsh example.

3.5 BufferedIO Example

By far, we know how to obtain and use NapletReader and NapletWriter to communicate with privileged services. Whereas, as you may notice, the information they read or write are all character-based. For byte-oriented applications, such as image or sound transmission, they are not the best solution. In the light of these cases, service channel offers another choice: NapletInputStream and NapletOutputStream, for byte stream communication. Furthermore, for consideration of efficiency, based on these two pairs of reader and writer, service channel also provides their buffered counterparts: BufferedNapletReader, BufferedNapletWriter and BufferedNapletInputStream, BufferedNapletOutputStream. In this example, we will learn how to use BufferedNapletReader and BufferedNapletWriter. The usage of another set of byte-oriented readers and writers are almost exactly the same, except they are designed for byte stream communication.

BufferedIO example implements a naplet that require privileged services on visited servers by buffered character stream. It aims to test the efficiency of the buffered IO between naplets and servers. Structurally, this example includes three files in the package: BufferedIOTest.java, BufferedIOListener.java, and BufferedIONaplet.java. They played the same roles as they do in Hello Example: initiator, listener, and mobile agent. Since there are nothing new in BufferedIOTest.java and BufferedIOListener.java. We will save our space here to examine BufferedIONaplet.java. If you have any question about these two files, please refer to the corresponding parts of Hello Example.

```
NapletOutputStream outStream
          = channel.getNapletOutputStream();
BufferedNapletOutputStream bOutStream
          = new BufferedNapletOutputStream(outStream);

NapletWriter writer
          = channel.getNapletWriter();
BufferedNapletWriter bWriter
          = new BufferedNapletWriter(writer);

NapletInputStream inStream
          = channel.getNapletInputStream();
BufferedNapletInputStream bInStream
          = new BufferedNapletInputStream(inStream);
```

```
NapletReader reader = channel.getNapletReader();
BufferedNapletReader bReader = new BufferedNapletReader(reader);
```

Now Let's go directly to the new stuff in BufferedIONaplet.java. As shown above, this code patch presents us how to get NapletReader, NapletWriter, NapletInputStream and NapletOutputStream from service channel. Strictly speaking, it is not different from what we have learned in Rsh Example, except that a pair of new reader and writer NapletInputStream and NapletOutputStream is introduced. However, take a close look at four even lines starting with "Buffered". These lines compose of what we will discuss in this section.

If you are familiar with Java I/O, you will also be familiar with the syntax here to create objects of BufferedNapletReader, BufferedNapletWriter, BufferedNapletInputStream, and BufferedNapletOutputStream: simply wrapping a input/output stream into the corresponding buffered stream. Actually buffered I/O is a very common performance optimization. It attaches a memory buffer to the I/O streams such that I/O operations can be done on more than one byte/character at a time, hence increasing performance. Service channel provides buffered streams for naplets in case there is a large amount of communication need to be done. The usage of these buffered streams are the same with what we have learned in last section: invoking one of the methods "read(char)", "read(byte)", "read(char[], int, int)", "read(byte[], int, int)", "write(char)", "write(byte)", "write(char[], int, int)" and "write(byte[], int, int)" will help naplets read or write from service channel.

The remaining parts of BufferedIONaplet.java are almost exactly the same as Hello Example. If you have any question, please refer to the corresponding parts there.

For byte-oriented and character-oriented applications, the service channel offers two pairs of I/O streams: NapletInputStream and NapletOutputStream, NapletReader and NapletWriter. Besides, for the consideration of performance, the service channel provides two pairs of buffered I/O streams: BufferedNapletInputStream, BufferedNapletOutputStream, BufferedNapletReader, and BufferedNapletWriter, as the wrappers for the corresponding streams. For the comparison of the performance between buffered streams and unbuffered streams, you may try to run two examples packed with the distribution of Naplet system: "bufferedIO" and "unbufferedIO" in the directory of "examples/naplet-client/".

Now the discussion of service channel part has been finished. By far, you are expected to be capable of answering following two questions:

1. What facilities service channel offers for communication between naplets and privileged services in Naplet system?
   (hint: NapletInputStream, NapletOutputStream, NapletReader, NapletWriter, BufferedNapletInputStream, BufferedNapletOutputStream, BufferedNapletReader, and BufferedNapletWriter)

2. How to use them? (hint: different versions of "read( )" and "write( )" methods)

Please refer to the installation section for the details about how to run the bufferedIO example.

3.6 Message Example

For some applications, single naplet may not be capable of handling the whole situation. Instead, more than one naplets need to cooperate with each other to complete the task. As one of the necessities for such cooperation, communication between naplets becomes very important. In Naplet system, naplets are provided a mechanism for persistent asynchronous communication. It is based on a post-office protocol: on arriving a naplet server, each naplet is associated with a mail box, which will store all the mails for this naplet when it stays. This mail box will be checked for new mails from time to time. Meanwhile, a naplet can contact the messenger of the current server it stays in to send mails to other naplets whose information are presented in the sender's AddressBook. The messenger functions as a postman in our real life, forwarding the mail to the target naplet's corresponding mail box. Details of the post-office protocol are discussed in section Post-Office Messaging Service in Naplet system part.

To explore the programming skills of such asynchronous communication, message example presents us a case that two naplets are dispatched in parallel and they communicate with each other by the post-office messaging service. Structurally, message example consists of two files: MessageTest.java and MessageNaplet.java. Since most of the codes in this example belong to the basic programming techniques that we discussed in former examples, we intent to elide them. Our major concern here would be naplets' communication part.

Passing those knowledge we already knew, let's go directly to the method "operate(Naplet)" of class "Barrier" in MessageNaplet.java.

```
NapletID myID = nap.getNapletID();
System.out.println( "My NapletID =" + myID.toString() );

Messenger postman = nap.getNapletContext().getMessenger();

// Address book of a naplet contains all the NapletID of its
// sibling naplets (due to cloning), as well as the address
// information inherited from its parent (or creator).
AddressBook aBook = nap.getAddressBook();
Iterator iter = aBook.iterator();
...
AddressEntry entry = ( AddressEntry ) iter.next();
NapletID nid = entry.getNapletID();
try
{
  if ( !nid.equals( aBook.creator() )
       && !nid.equals( myID ) )
  {
    Message msg = new Message( myID, nid,
                     "Helloooo from " + myID.toString() );

    postman.send( myID, entry.getServerURN(), msg );
    System.out.println( "Sending message to " +
                        entry.getServerURN().toString() );
  }
}
```

As shown above, these codes present the procedure a naplet sends a mail to another naplet whose information is recorded in sender's AddressBook. Following the flow of the codes above, first the naplet gets its NapletID by method "getNapletID( )" inherited from class Naplet, then it obtains the Messenger of the server it currently visits through NapletContext, and AddressBook by method "getAddressBook( )" from class Naplet. As its name states, AddressBook contains address information of related naplets. In the book, each naplet has an AddressEntry. Each entry is composed of information like the related naplet's NapletID, the URN of the server this related naplet is currently in or visited. AddressEntry is accessible through the iterator of AddressBook. The NapletID of the naplet corresponding to each entry can be obtained by method "getNapletID( )" of class AddressEntry. To avoid self-sent mails, the NapletID is checked to see whether it is the same as the sender's NapletID. If not, a Message is constructed by offering the sender's NapletID, the receiver's NapletID, and the string message to send. Messenger "postman" is responsible for mailing this message to the target naplet. Here please pay attention to three arguments of method "send(NapletID, URN, Message)" from class Messenger. They correspond to the sender's NapletID, the URN of the destination server, where the target naplet is expected to currently be, and the Message to be mailed, respectively. The second argument is gotten from the target naplet's AddressEntry in the AddressBook. It is the newest updated server URN information of the target naplet, which suggests the possible location the target naplet currently may be. Notice that we use "possible location" here. Unlike stationary objects, naplets are mobile in naplet servers' network. It is very difficult to precisely locate where naplets are. The server currently hosting naplets could become past tense in any minute. Naturally, problems will arise if the receiver left that server, without updating its entry on time, or the receiver does update its entry, while it is blocked somewhere without reaching the destination as expected. However, since the major concern of this example part of the tutorial is programming skills, we will simply teach you that, using method "getServerURN( )" from class AddressEntry of corresponding target naplet to provide mails' destination information will be enough. Naplet system will handle the remaining. Whereas, if you are interested in how these problems get worked out, please refer to section Post-Office Messaging Service.

```
for ( int i = 0; i < aBook.size() - 2; i++ )
{
  Message msg = postman.checkMailBox( myID );
  if ( msg != null )
  {
    System.out.println( msg.getMessage() );
  }
  else
  {
    System.out.println( "No new message" );
  }
}
```

As said at the beginning, naplets will check their mail boxes from time to time for new messages. The checking is done by invoking the method "checkMailBox(NapletID)" from class Messenger,

as shown above. The string-formated message can be obtained by method "getMessage( )" of class Message.

The itinerary of this example is constructed based on the parallel pattern. The details will be discussed in section Itinerary.

Naplets need communicate with each other for cooperation. For this purpose, Naplet system provides a post-office messaging service for persistent asynchronous communication between naplets. To take advantage of this service, the major programming skills needed to know are how to get the Messenger and use it to send and check mails; how to get AddressEntry in AddressBook to obtain information of related naplets. If you have answers to these two questions, basically you grasp the knowledge introduced in this example, and are ready to use the post-office service in Naplet system.

Please refer to the installation section for the details about how to run the message example.

3.7 NapletSocket Communication Example

Unlike the asynchronous persistent post-office messaging service, which adopts stationary mail boxes for naplets' communication, NapletSocket connection offers synchronous transient communication between naplets. As its name states, such communication is based on conventional Socket and TCP/IP protocol. Further, it is adapted to the fact that naplets are always migrating among the naplet server network, such that once a NapletSocket connection is established, it will migrate with the owner naplets. The connection migration is transparently performed by an underlying component called *SocketController*. Therefore, in the application layer, the NapletSocket connection works the same as the conventional stationary Socket connection. The detailed information of the NapletSocket connection migration is addressed in section NapletSocket Connection Migration Mechanism in Naplet System part.

Resembling Java Socket API, NapletSocket package interface is composed of two classes: *NapletSocket* and *NapletServerSocket*. The usage of these two classes is very similar to Java Socket. Basic techniques are presented in example *napletSocket*.

The napletSocket example implements a naplet application that two naplets are dispatched in parallel and they communicate with each other by a NapletSocket connection, while migrating over the network. These two naplets moves concurrently in a sequential itinerary formed by the same number of hosts. The naplet with lower version number will act as the server, and another will act as the client. The connection is established when the naplets land in the first host they visit. They keep communicating with each other by the established NapletSocket channel while migrating.

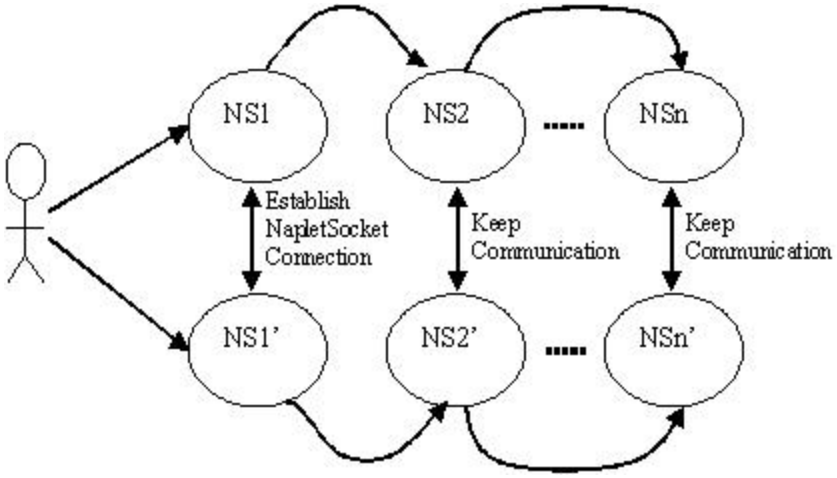


Fig. 3. The SocketTalkNaplet

Structurally, napletSocket example consists of two files: SocketTalkTest.java and SocketTalkNaplet.java. Since at this moment, we have experienced many naplet examples, our major concern here will be the techniques as to how to use the NapletSocket to communicate.

```
// set server name; Here the assumption is
// the 1st machine has the SocketServer
setServer( servers[0] );
setServerID( getNapletID().toString() + ".1" );
```

In the class SocketTalkNaplet, first we noticed that a NapletSocket attribute "napletSocket" is declared. This attribute functions as the NapletSocket connection that the naplets deploy for synchronous communication. We will see its acting later in the program. As shown above, in the constructor *SocketTalkNaplet(String, String[], NapletListener)*, a server that the naplet is to visit "servers[0]" is selected as the one where the NapletServerSocket is set up listening to the NapletSocket connection request. The corresponding NapletID which serves as the server in the NapletSocket connection is set as the original naplet id concatenating with the version number ".1".

```
String rank = getNapletID().getVersion();

if ( rank.equals( ".1" ) )
{ // server
   if ( napletSocket == null )
   { // first time start a server and accept
     // nsocket
     try
     {
       NapletServerSocket nss
           = new NapletServerSocket( getNapletID() );
```

```
        napletSocket = nss.accept();
      }
      catch ( Exception e )
      {
        e.printStackTrace();
      }
    } // end of first time for server

    ...

    try
    {
       PrintWriter out = new PrintWriter(
             napletSocket.getOutputStream(), true );
       BufferedReader in
             = new BufferedReader( new InputStreamReader(
                     napletSocket.getInputStream() ) );

       String request = null;
       request = in.readLine();
       System.out.println( "get a request:" + request );

       String response = null;
       response = "echo:" + request;
       out.println( response );
    }
    catch ( Exception e )
    {
      e.printStackTrace();
    }
} // end of server
```

In the method onStart( ), we will see how the server and client roles are differentiated between naplets. As shown above, first the NapletID version is fetched and assigned to a String variable "rank". It is then compared with ".1", which is selected as the naplet's version number that acts as the server role in the constructor of SocketTalkNaplet, as we discussed before. If the value of "rank" equals to ".1", this naplet is the server naplet. It immediately checks whether there is any existing NapletSocket connection, by comparing the value of "napletSocket" with "null". If the connection has not established yet, a NapletServerSocket object "nss" is created. Please notice that the constructor of NapletServerSocket takes the NapletID of the naplet which creates the NapletServerSocket as the parameter. The NapletServerSocket "nss" begins listening to client's request. Once one NapletSocket connection request is detected and accepted by the server, "napletSocket" is populated and ready to use.

After the connection is set up, input and output stream can be obtained from NapletSocket by methods "getInputStream()" and "getOutputStream()". The communication is accomplished by reading from and writing into the output and input stream. In the example, the server naplet first reads a request from the client naplet, then the request is written back as part of the response to the client naplet.

```
else
{ // all others are client
  try
  {
    // if first time, setup socket with the server
    if ( napletSocket == null )
    {
      // delay some time in case server needs more
      // time to set up.
      Thread.currentThread().sleep( 3000 );

      // first time for open napletsocket
      NapletID mynid = getNapletID();
      NapletID destnid = ( NapletID ) mynid.clone();

      destnid.setVersion( ".1" );

      napletSocket = new NapletSocket( mynid, destnid );
    } // end of first time for client

    PrintWriter out = new PrintWriter(
            napletSocket.getOutputStream(), true );
    BufferedReader in
           = new BufferedReader( new InputStreamReader(
                   napletSocket.getInputStream() ) );

    String request = null;
    request = "request from:" +
          InetAddress.getLocalHost().getHostName();
    out.println( request );

    String response = null;
    response = in.readLine();
    System.out.println(
            "**response from server is:" + response );
  } //end of client
  catch ( Exception e )
  {
    e.printStackTrace();
  }
}
```

If the NapletID version of the naplet does not equal to ".1", this naplet will play the client role. As shown above, the client naplet first checks whether NapletSocket connection exists. If not, it sleeps for a while, waiting for the server naplet landing. To set up a NapletSocket connection, the client naplet creates an object of NapletSocket, by providing its own NapletID, as well as the NapletID of the server naplet. In this example, since the client and server naplets are both cloned by one naplet in a parallel itinerary, their NapletIDs are the same, except that the NapletID version of the server naplet is ".1". Once this object is created, the NapletSocket connection between the server and client is established. As mentioned before, the client naplet can obtain input and output stream from the connection to communicate with the server.

After sending/reading the request, reading/sending the response, both server naplet and client naplet will travel to the next host, according to their itineraries. In the following hosts, however, the naplets will go directly to grab the input and output stream, since the connection has already been established in the first host naplets visit and the connection is kept with the naplets while they migrating. In Naplet system, the connection migration is handled by the underlying component in a transparent manner, such that in the application layer, the connection appears to be location-independent and works in very similar way that JAVA Socket does.

The itinerary of this example is constructed based on the parallel pattern. The details is discussed in section Itinerary.

There are some special requirements to run the NapletSocket-related examples. Please refer to the installation section for the details.

3.8 Loop Example

The Loop example implements a naplet that travels around the server network in a Loop itinerary. It includes two files: LoopTest.java and LoopNaplet.java, of which, LoopTest.java has almost the same structure as its counterparts of the former examples we studied. LoopNaplet.java, however, has something new shining: itinerary pattern Loop and the concept of conditional itineraries. In this section, our discussion will center on these two issues.

Now, let's move to the file LoopNaplet.java for a closer look of these two issues. Before the codes' analysis begins, we ask you to pay attention to one variable named "count" as we go along the whole file, because it actually plays a very important role in this example.

```
getNapletState().set( "count", new Integer( 5 ) );
```

As shown above, in the second constructor of class LoopNaplet, we notice that the variable "count" is created. It is set as Integer "5" and saved as a property of NapletState. At this moment, we totally have no clue what this variable is used for, except that it is part of the NapletState and its value is "5". Keeping these in mind, let's continue our digging.

```
try
{
  Integer count = ( Integer ) getNapletState().get( "count" );
  int c = count.intValue();

  System.out.println( "Loop Count = " + c );
  getNapletState().set( "count", new Integer( c - 1 ) );
  ...
}
```

As we go along, in method "onStart( )", the variable "count" appears again. Here it is first fetched from NapletState. Its value is checked and displayed, then is reset a new value which is one less than the original. Since "onStart( )" will be executed each time naplets arrive in a server, the value of "count" will be updated by lessening one for each visit, which, in some sense, could make the variable "count" a very good down counter for naplets' server-visit times. However, no more evidence could be found in method "onStart( )" to testify our guess. We decide to keep going for new information.

Till now, the examination of class "LoopNaplet" is complete. Two private classes: "Guardian" and "ICItinerary", remains for inspections.

```
private class Guardian
      implements Checkable
{
  public boolean check( Naplet nap )
  {
    boolean going = false;
    try
    {
      Integer count = ( Integer ) getNapletState().get( "count" );
      if ( count.intValue() > 0 )
      {
        going = true;
      }
    }
    catch ( NoSuchFieldException nsfe )
    {}

    return going;
  }
}
```

Class "Guardian" implements the interface Checkable, which is designed for pre-condition checking of each single itinerary pattern. Simply speaking, the mechanism of such "pre-condition checking" works as followed: each time before naplets are about to be launched to next destined server, method "check(Naplet)" in Checkable is executed. It returns a boolean value, which is inspected subsequently. If it is "true", launch is performed as usual. If it is "false", however, current itinerary pattern will be skipped. As a result, naplets either finish their itinerary if there is no more pattern to perform in the itinerary stack, or continue next itinerary pattern popped from the itinerary stack (As to the information of itinerary stack and relationship between itinerary and itinerary pattern, we will have detailed discussion in next section Itinerary).Therefore, if a class implements interface Checkable, and is binded with an itinerary pattern, this pattern becomes a conditional one: method "check(Naplet)" would determine whether this pattern needs to go on or to be terminated.

Back to class "Guardian". In method "check(Naplet)", "Guardian" declares a boolean variable "going" and set it "false". Then the variable "count" is obtained from NapletState and compared

with zero. If the value of "count" is greater than zero, "going" is reset to "true". The value of "going" is returned at last.

Given the operations on "count" defined in class "LoopNaplet", the operations here seem support our "down counter" guess. To make the image clearer, let's piece together all information of "count" from the beginning. First, "count" is initialized as "5". When a naplet arrives in a server, the value of "count" is lessened by one. Then before the naplet is launched to next server, the value of "count" is compared with zero. If greater than zero, launch is performed, and the lessening and comparing steps repeat in the new server. Otherwise, the current itinerary pattern is finished, since "count" less than zero will result a "false" returned from method "check(Naplet)". A typical count down behavior, isn't it? Based on the analysis above, now we can conclude that, according to the initial value of "count", in this example, naplets are limited to visit 5 servers all together. Naturally, following this conclusion, one subsequent question would arise: how these five servers are visited. In this regard, class "ICItinerary" could provide us the answer.

```
Checkable guard = new Guardian();

// Loop itinerary
ItineraryPattern ip = new SeqPattern( servers );
setRoute( new Loop( guard, ip ) );
```

Class "ICItinerary" extends class Itinerary, providing a constructor of its own. As shown above, in the constructor a Checkable object "guard" is created, which is actually "Guardian" type. Then a sequential itinerary pattern "ip" is created. As arguments, these two objects are passed to a Loop constructor to create a loop itinerary pattern, which, in turn, is set as the current itinerary pattern of "ICItinerary".

Same as singleton, sequential, and parallel patterns, Loop is also an itinerary pattern defined in Naplet system. It works very much like a for-loop or while-loop in most programming languages. Naplets in such pattern will travel around the destined servers iteratively, until certain condition is satisfied. Fig. 4 gives some visual hints of Loop itinerary.

Recall that in former examples, when creating itineraries, no Checkable argument is involved. Actually for patterns like singleton, sequential, and parallel, it is optional to include pre-conditions. Whereas, in Loop pattern, it is mandatory, since there has to be a condition to terminate the loop (we assume nobody wants any format of forever loops). Another distinction of Loop is that, for simplicity, currently it is restricted to be loop over singleton or sequential patterns only, which implies that naplets in loop pattern could either repetitively visit one server, or iterates over multiple servers sequentially, until the termination condition is satisfies.

Since this example involves too much knowledge of itinerary and itinerary patterns, which are not adequately explored in former sections, it would be very normal if you still have confusion about Loop. However, you are expected to understand the basic concept of conditional itineraries: the Checkable part, because it will be deployed in the next section Itinerary. If you are clear about this concept, your confusion here could be solved in Itinerary since we will have detailed discussion of itinerary and itinerary patterns there.

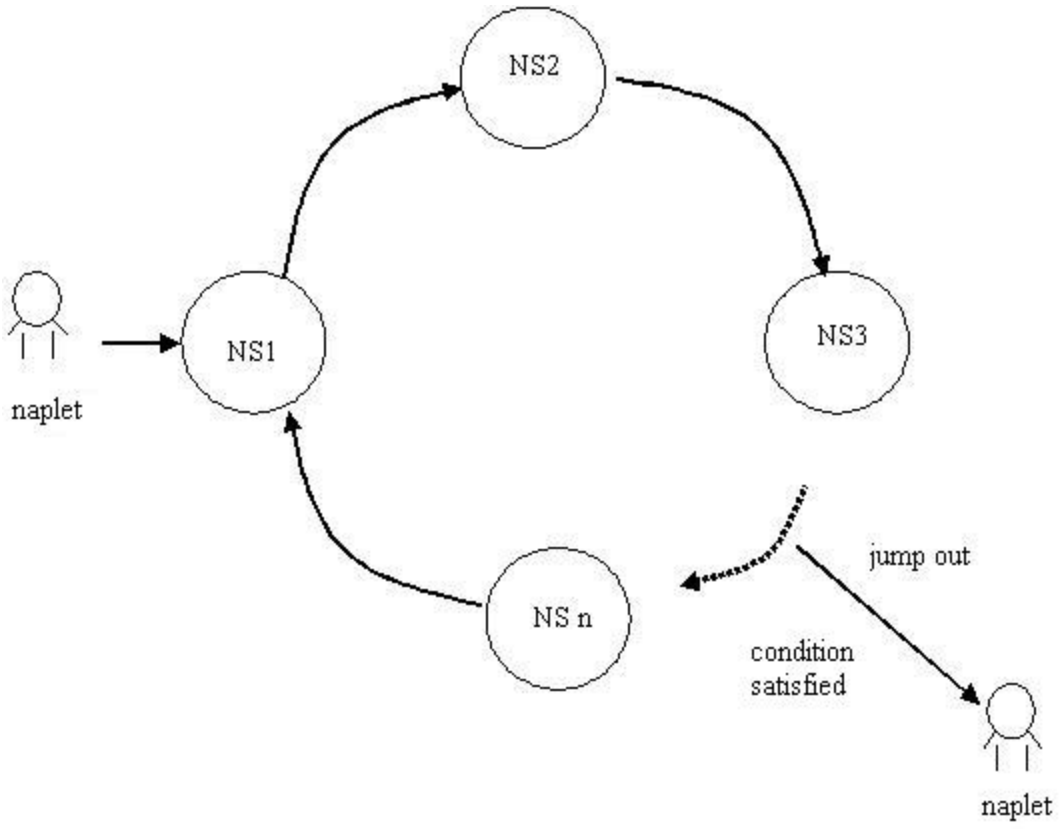


Fig. 4: Loop Itinerary

3.9 Itinerary Example

Except functionality, another important feature of a naplet is its itinerary. Itinerary of a naplet, as the name states, specifies the way the naplet travels in the network of naplet servers. Usually, it is defined as an extension of class Itinerary, whose operations, in turn, revolve around another important class in itinerary: ItineraryPattern. The relationship between ItineraryPattern and Itinerary is like content to container. Itinerary provides an interface for naplets to store and obtain their specific ItineraryPattern. Whereas, as the backbone of Itinerary, ItineraryPattern defines the pattern a naplet visits servers.

In Naplet system, Itinerary is designed in a way such that one itinerary could possibly hold multiple ItineraryPatterns. As shown in Fig. 5, according to the order by which the itinerary patterns are placed into an itinerary, one will be set as the current pattern for naplets to use, while the others have to be pushed into a stack in Itinerary for future use. Once the current one is

complete, itinerary will pop another one from the stack as the current. Until the stack is empty, the itinerary is not finished.

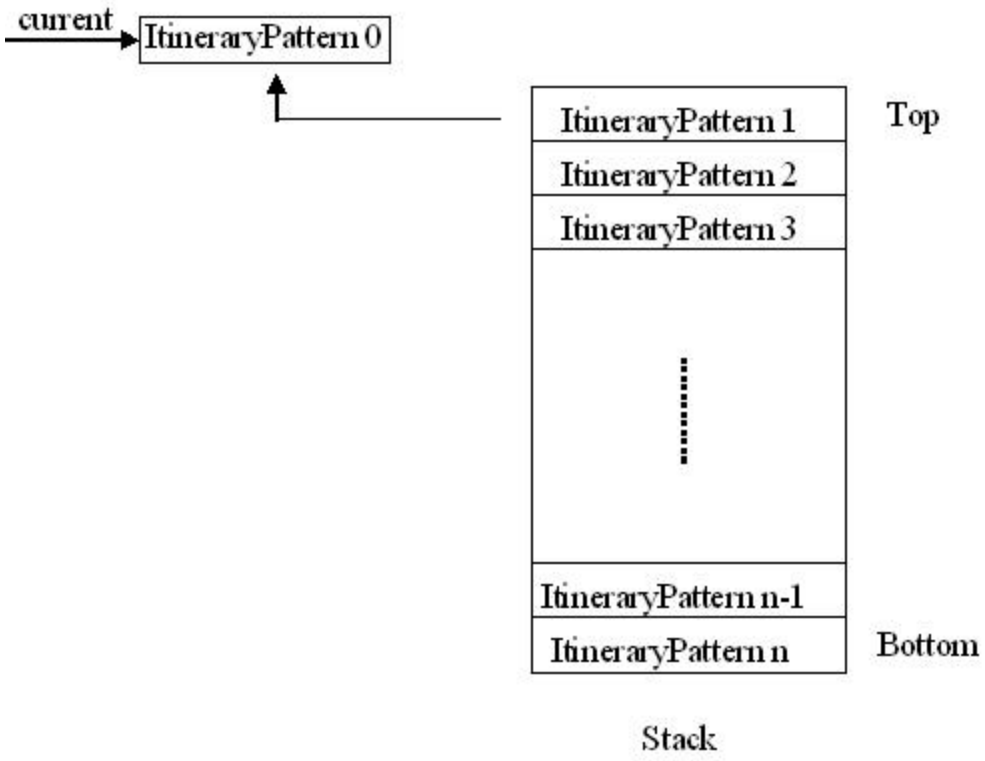
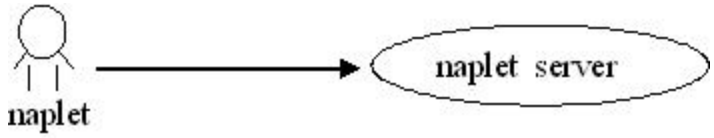
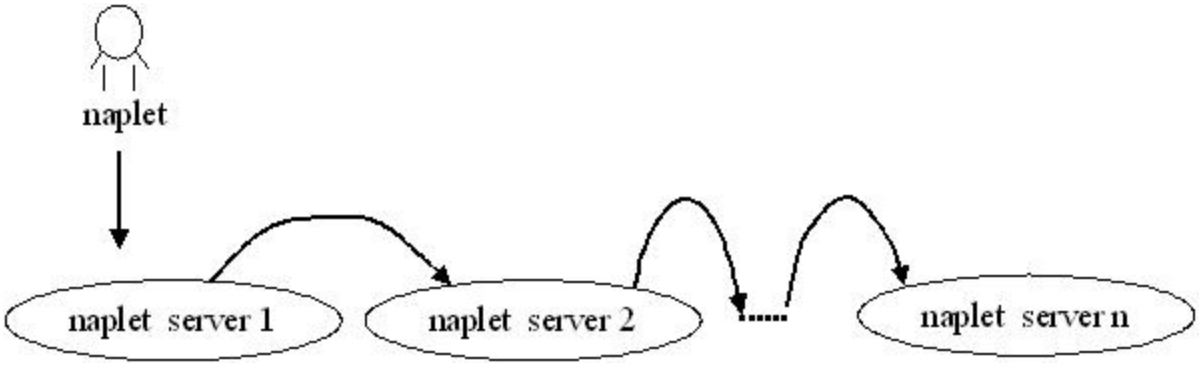


Fig. 5: Itinerary structure

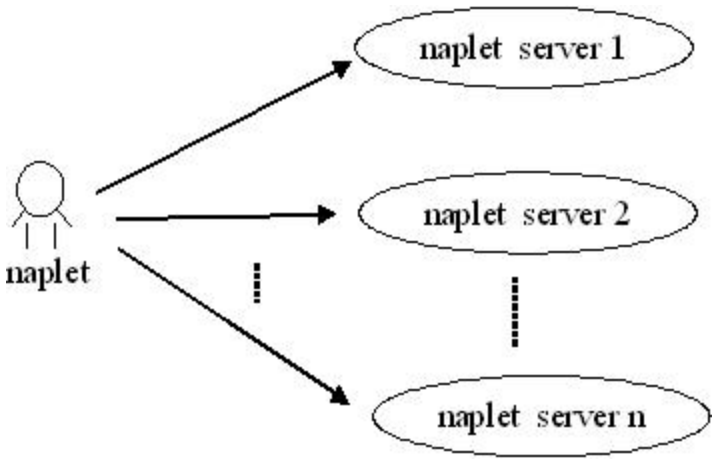
Currently, in Naplet system, there are three primitive patterns defined: singleton(SingletonPattern), sequential(SeqPattern), and parallel(ParPattern). Of those, singleton pattern consists of single server to visit. Sequential pattern defines a sequential visit to destined servers, while in parallel pattern, naplets are cloned and dispatched simultaneously to destined servers. Besides, Naplet system also defines a composite itinerary pattern: loop(Loop), which specifies a conditional iterative visit to target servers. Fig. 6 illustrates these four cases. More complex patterns can be constructed by recursively combining these ones. We will have examples for corresponding techniques later.



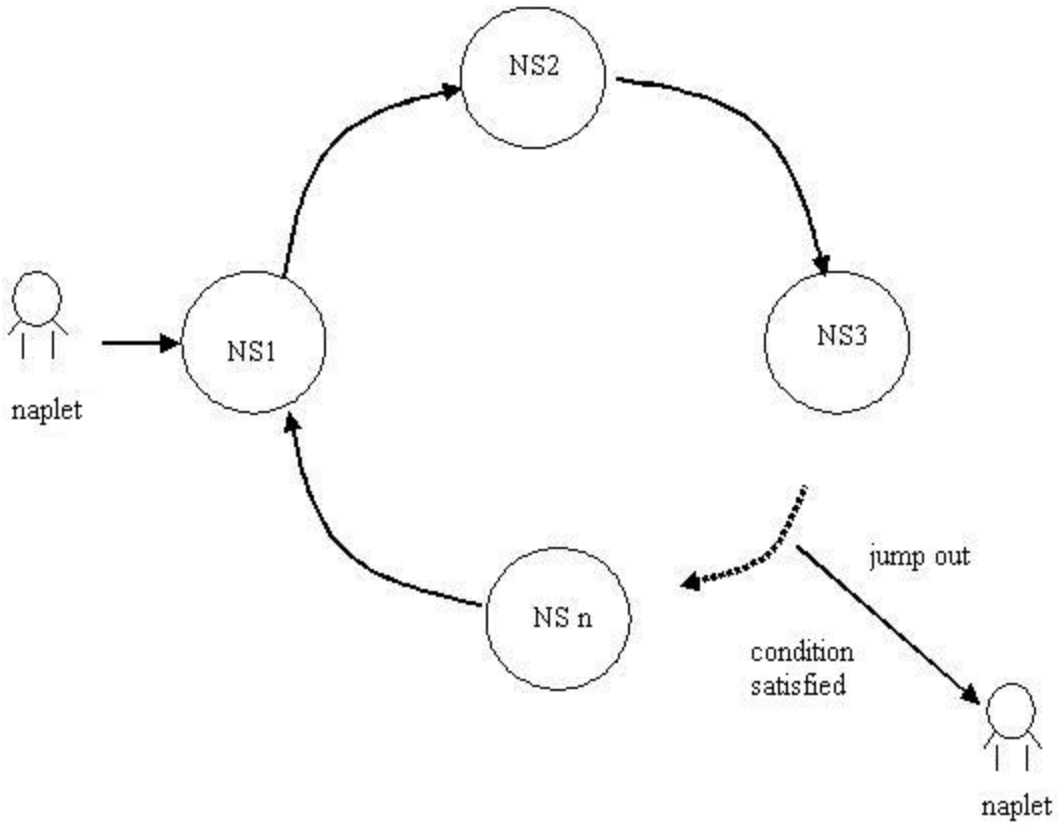
Singleton Pattern



Sequential Pattern



Parallel Pattern



Loop Pattern

Fig. 6: Itinerary patterns

Before we go any further, two itinerary-related interfaces: Operable and Checkable, need a few more words here. Operable defines an interface of operations to be performed at the end of an itinerary, while Checkable defines an interface of guardian operations to be performed as a pre-condition of each single itinerary pattern. Recall in Hello example, you learned how to implement and relate Operable to a sequential itinerary pattern. Loop example discusses how to apply Checkable to construct a loop itinerary. These two concepts will also be deployed in the following examples. However, since they are already discussed in former sections, we will not spare space to detail their usage. If you need a review, please refer to the corresponding parts of Hello example and Loop example. The emphasis in this section is on the construction of different kinds of ItineraryPatterns.

```
public SeqPattern( String[] hosts )
public SeqPattern( String[] hosts, Operable act )
public SeqPattern( Checkable guard, String[] hosts )
public SeqPattern( Checkable guard, String[] hosts, Operable act )
```

```
public SeqPattern( ItineraryPattern[] itin )
public SeqPattern( ItineraryPattern[] itin, Operable act )
public SeqPattern( Checkable guard, ItineraryPattern[] itin )
public SeqPattern( Checkable guard, ItineraryPattern[] itin, Operable act )
```

Fig. 7: Constructors of sequential itiernerary pattern.

Now let's look at the ways to create primitive itinerary patterns: the constructors. Fig. 7 shows all the constructors of sequential itinerary pattern. Actually, there are exactly the same counterparts in parallel pattern. By saying the same, we mean the same signatures of each constructor. To avoid redundancy, only sequential case is explored here. However, the analysis applies to the parallel case, too.

To create a sequential itinerary pattern, you have eight choices. By referring to Fig. 7, let's address them one by one. In the first case, sequential pattern is created by offering an array of string, which specifies the names of the servers to visit. Please note that there are neither pre-condition nor post-operations related to each visit in this constructor, which implies that in a sequential itinerary pattern created like this, naplets simply perform their functionality defined in the methods of extended Naplet classes, server after server. For applications where naplets do not need any operations before they carry out or after they complete their tasks, this constructor would be a good choice. However, in case we need some pre-condition or post-operations in a visit, sequential pattern offers other solutions. In the second constructor, post-operations are added as the second parameter of the constructor. Actually, this constructor is not new for us. We used it all the way in Hello example, Rsh example, BufferedIO example and Message example. As a very loose term, post-operations could be used for different purposes. For instance, in Hello example, Rsh example and BufferedIO example, the post-operations are all related to reporting results through NapletListener. Whereas, in Message example, they are designed to send messages to other naplets and check the mail box before naplets depart from a server. Although there is a huge space to define application-specific post-operations, all of them share the same interface Operable. Similarly, the third constructor has a parameter for pre-condition. Actually, we have seen it in Loop example, in which, it is deployed as a termination condition for the iterative loop pattern, by checking the value of a counter. Same as post-operations, the design of pre-conditions is application-specific, but they all share one interface Checkable. The fourth constructor in Fig. 7 is a combination of last two. Except it includes both pre-condition and post-operations, the usage of this constructor is the same as last ones.

Remaining four constructors look very similar to the four we have discussed, except that they use an array of ItineraryPattern, instead of an array of string, to represent the servers to visit. Please recall that all three primitive patterns are extensions of ItineraryPattern, which implies that they are all qualified candidates for this parameter. Based on this implication, if we step further, we would dare to conclude that these four constructors provide a mechanism for the user to construct complex itinerary patterns by embedding a primitive one into another. Since the patterns constructed after such embedment are still extension of ItineraryPattern, no matter how complex they are, this kind of embedment could be done recursively, and endlessly, if you want. By doing so, very complex itinerary patterns can be composed. We will see examples next.

```
  public SingletonPattern( String urnStr )
  public SingletonPattern( String urnStr, Operable act )
  public SingletonPattern( Checkable guard, String urnStr )
  public SingletonPattern( Checkable guard, String urnStr, Operable act )
```

Fig. 8: Constructors of singleton itinerary pattern

Different from sequential and parallel patterns, singleton pattern only have one server to visit. Therefore, it is not necessary to provide constructors similar to the last four in sequential and parallel cases. Except that, all the analysis above applies to singleton pattern, too.

```
  public Loop( Checkable guard, ItineraryPattern itin )
```

Fig. 9: Constructor of loop itinerary pattern

As you may notice, to construct patterns we discussed above, the pre-conditions are actually optional. Loop pattern, however, is not the case. Since it is in a "loop" form, there has to be some condition to make the loop finite (we assume that nobody wants forever loop). Therefore, as shown in Fig. 9, the single constructor of loop pattern mandates a pre-condition. Another parameter of the constructor is for itinerary pattern. Currently, for simplicity reason, Naplet system restricts loop pattern only over sequential or singleton pattern. We have more detailed discussion of Loop pattern in Loop example.

```
class ResultReport
   implements Operable
{
   public void operate( Naplet nap )
   {
    ...
   }
}

class Guardian
   implements Checkable
{
  public boolean check( Naplet nap )
  {
   ...
  }
}

class ICItinerary
   extends Itinerary
 {
   public ICItinerary( String[] servers )
     throws InvalidItineraryException
   {
     super( );

     Operable act = new ResultReport();
     Checkable guard = new Guardian();
```

```
    // Singleton(s0)
    setRoute( new SingletonPattern( servers[0]) );

    // seq(s0, s1, ..., sn)
    pushRoute( new SeqPattern( guard, servers ) );

    // par(s0, s1, ..., sn)
    pushRoute( new ParPattern( guard, servers, act ) );

    //  broadcast
    ItineraryPattern[] ip = new ItineraryPattern[servers.length];
    for (int i=0; i<servers.length; i++)
    {
      ip[i] = new SingletonPattern( guard, servers[i], act );
    }
    pushRoute( new ParPattern( ip ) );

    // par(seq(s2, s0),seq(s1, s0))
    String[] path0 = { servers[1], servers[0] };
    String[] path1 = { servers[2], servers[0] };
    ItineraryPattern[] ip = new ItineraryPattern[2];
    ip[0] = new SeqPattern( path0 );
    ip[1] = new SeqPattern( path1 );
    pushRoute( new ParPattern( guard, ip, act ) );

    // par(seq(s2, s1, s0))
    String[] path = { servers[2], servers[1], servers[0] };
    ItineraryPattern[] ip = new ItineraryPattern[1];
    ip[0] = new SeqPattern( path );
    pushRoute( new ParPattern( guard, ip ) );

    // seq(s1; par(s2, s3))
    ItineraryPattern[] ip = new ItineraryPattern[2];
    ip[0] = new SingletonPattern( servers[2] );
    String[] s = { servers[1], servers[0] };
    ip[1] = new ParPattern( s, act );
    pushRoute( new SeqPattern( ip ) );

    // par(s1; seq(s2, s3))
    ItineraryPattern[] ip = new ItineraryPattern[2];
    ip[0] = new SingletonPattern( servers[0] );
    String[] s = { servers[1], servers[2] };
    ip[1] = new SeqPattern( s, act );
    pushRoute( new ParPattern( guard, ip, act ) );

    System.out.println( "ICItinerary = " + getRoute().toString() );
  }
}
```

Fig. 10: Examples of Construction of Itinerary

After inspecting the constructors, let's apply the knowledge we learned above in some examples for better understanding. In Fig. 10, we construct an environment to define an itinerary by

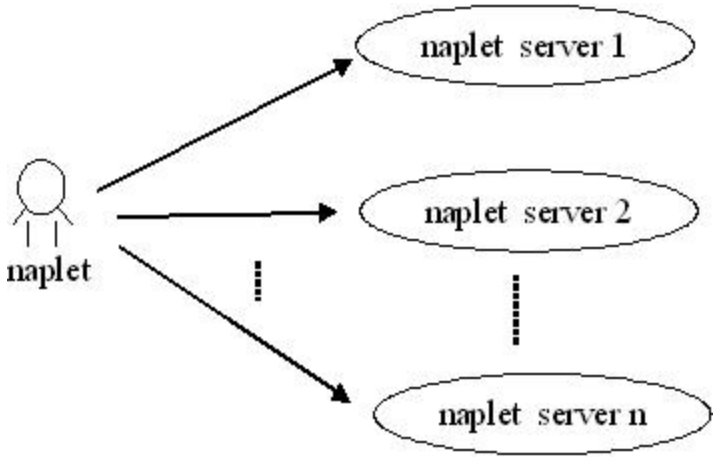
different kinds of itinerary patterns. Since one itinerary can only has one current pattern, all the itinerary patterns defined after the first one are pushed into the stack for future use.

As shown in Fig. 10, class ICItinerary specifies an itinerary by extending class Itinerary. Inheriting every property and method from its parent, ICItinerary defines a constructor of its own. In the constructor, method "setRoute(ItineraryPattern)" of class Itinerary is deployed to initialize the current itinerary pattern for class ICItinerary, meanwhile method "pushRoute(ItineraryPattern)" is responsible for depositing the remaining itinerary patterns into a stack. Please note that how many patterns one itinerary contains and how these patterns related with each other are application-specific. Our example is solely for conceptual display purpose, without any application background. It is not promised to work in reality. Actually we do not recommend readers to use any meaningless itinerary in real life, either. Always remember to stick to the application requirements.

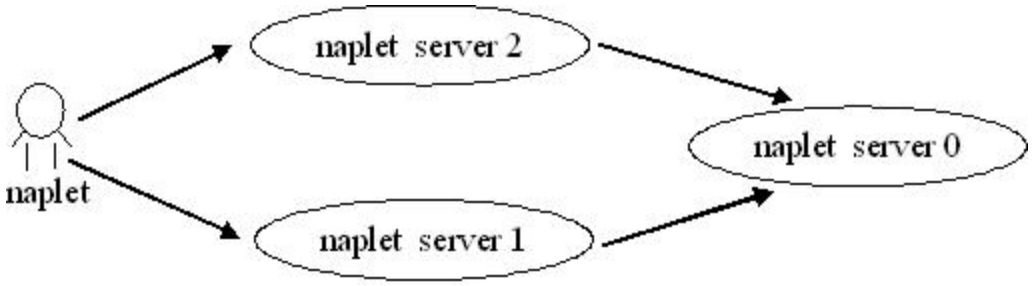
Now let's back to the topic, looking at those patterns defined in Fig. 10 one by one. The first pattern is a very simple one: a singleton pattern without any pre-condition or post-operations. The second and the third ones actually are also very simple: basic sequential pattern with pre-condition and parallel pattern with both pre-condition and post-operations. Since they are very clearly self-expressed, we do not offer any further explanations here.

As to the fourth one, it utilizes the itinerary patterns' embedment we discussed above. First it constructs an array of ItineraryPattern. The array then filled with homogeneous SingletonPatterns with both pre-condition and post-operations within a for-loop. These SingletonPatterns are embedded into a ParPattern by using the fifth constructor in Fig. 7, except it is parallel pattern, instead of the sequential. This form of itinerary could serve as broadcast, because of its homogeneity in parallel pattern. The fifth one in Fig. 10 distinguishes from the fourth in that its embedded two patterns are constructed in sequential form. The sixth is a parallel pattern enclosing a sequential pattern, while the seventh is a sequential pattern enclosing two different patterns: one is singleton, and the other is a parallel consisting of two servers. By contrast, the eighth is a parallel pattern enclosing one singleton and one sequential consisting of two servers.

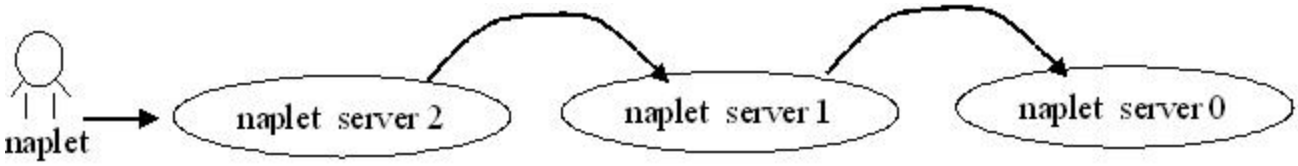
Please note that the pattern embedment in cases above share the same procedure: first create an array of ItineraryPatterns, then fill each element in the array the pattern you want, pass the array as an argument of the corresponding constructor of the final pattern you choose. Actually, this procedure could be repeated recursively, thus resulting more complex itinerary patterns. Fig. 11 offers some visual hints for above discussed composite patterns.



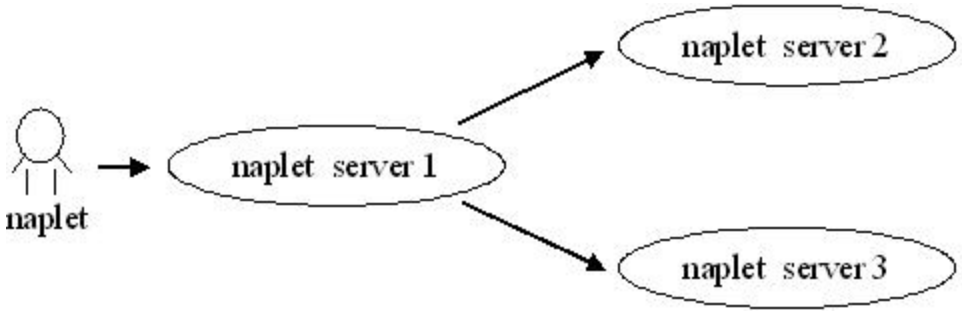
Fourth case



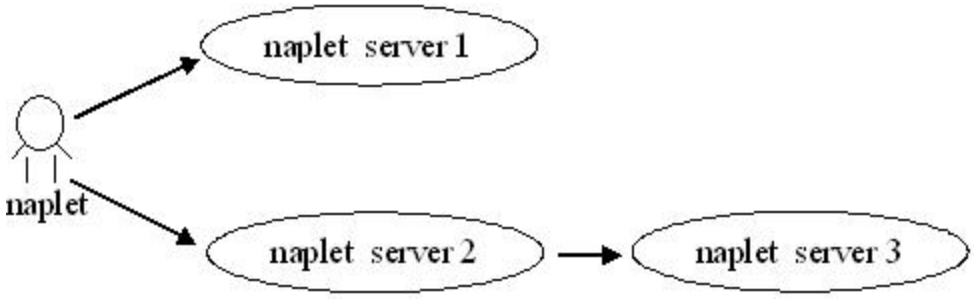
Fifth case



Sixth case



Seventh case



Eighth case

Fig. 11: Composite itinerary patterns according to Fig. 10

Assuming one naplet follows ICItinerary defined above (although we do not recommend doing so, let's assuming), theoretically it will execute all eight itinerary patterns one by one. Please note that, the "push" action is not confined to the constructors of itinerary. Itinerary patterns could be pushed into a naplet's itinerary at any point during its life cycle. By contrast, in the constructor, one pattern has to be set as the current one, for immediate use by naplets. Recall that, itineraries in examples we studied in former sections, all have only one pattern defined for naplets.

By finishing this section, we hope you master the techniques to construct itinerary and itinerary pattern. Actually, the itinerary part is one of the most difficult parts in Naplet system. It is full of variations. Practice is recommended for better understanding the knowledge taught here.

Now our example part of the tutorial is complete. If you understand everything we discussed in this part, you should be capable of developing your own applications based on Naplet system right now. However, if you think this part is not adequate for your situation, you can continue your journey in Naplet system part. There you may find more advanced details about the whole Naplet system.