Communication between Client and Server in the Distributed Object-oriented Repository Network Management System

By

Shan Jin
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180-3590

Computer Science Masters Project

Content Table

1. Introduction …….. 3

2. Design and Implementation ……. 4

2.1.Connection between the client and the repository ……4

2.2.Client interface ……5

2.3. Repository……6

3. Experimental result……..8

4. System requirements and running procedure ……10

4.1. System requirements ……10

4.2. Procedure to run the whole program ……10

5. Future work ……13

6. Summary ……..13

Reference ……14

Appendix a: snmpCB.idl source code……..15

Appendix b: ProNet_clientCB.java source code ……16

Appendix c: ProNet_serverCB.java source code ……29

1. Introduction

The Distributed Object-oriented Repository System (DOORS) constitutes a middle ware between independent network managers and network routers. It provides a secure, efficient, and fault tolerant method for the acquisition, management, manipulation, aggregation, and caching of network data and objects [1]. It requires the use of mobile agents and a dynamic interface to support management and collection protocols such as SNMP, LDAP or NDS [1].

The architecture of DOORS (involving only one repository) is given in Figure 1.

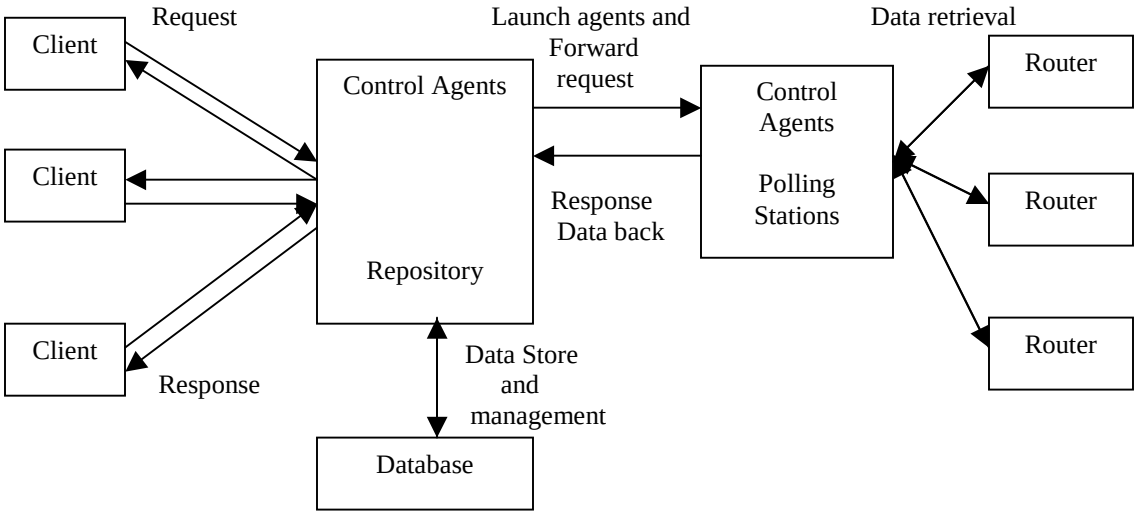


Figure 1: DOORS Architecture

Each DOORS repository controls its mobile agents and coordinates requests received from different clients. Upon receiving a request, the repository determines if a recent copy of the requested data exists in the storage system; if it does, the data are retrieved from there to reduce network traffic, round trip time, and router load. Otherwise, the request is sent to the router through mobile agents [2].

This report is concentrated on the dynamic interface between the client and the repository server. we choose a CORBA object model as the primary vehicle for communication and management of distributed clients and repository.

The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object management Group (OMG), which provides a model for interactions between objects in a distributed environment [3]. Network management is in essence a distributed

application. From this point of view, CORBA based network management is certainly an attractive approach.

Currently we are using sun's Java IDL version of CORBA, called idltojava. Java IDL adds CORBA capability to the Java 2 plat form, providing standards-based interoperability and connectivity. The most important advantage to choose idltojava CORBA object model is its compatibility between each new version and old version, which makes the software easy to maintain. Also, the idltojava CORBA object model provides call back object, which is important in the two-way communication between the client and the repository. The other advantage to choose Java as the primary development language is the need for portability of code, and its wide use in agent systems that we need for communication between repository and network routers.

Part two described the design and implementation details for communication and management between the client and the repository. Part three described the experimental results in current stage. Part four introduced the system requirement and the procedure to run the whole DOORS system. Part five introduced some points to be considered in future work.

2. Design and Implementation

The communication between the client and the server is a two-way communication. The client sends the snmp requests to the repository. The repository forwards the snmp request to the polling station through the mobile agents created by DOORS system. When the polling station gets the snmp responses, it sends the responses back to the repository through the mobile agents, then the repository sends the snmp responses back to the corresponding client.

Since each client can send snmp requests to multiple routers, and each repository needs to communicate with multiple clients at the same time, there should be some mechanisms to handle the mapping between each snmp request and each snmp response, and the mapping between each snmp response and each client.

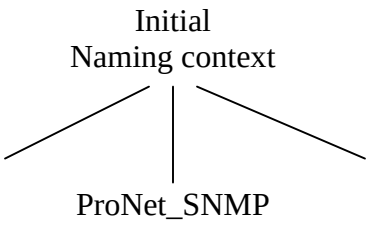
2.1. Connection between the client and the repository

The connection between the client and the repository is using CORBA Naming Service. The Naming Service provides a tree-like directory for object references. Object references are stored in the namespace by name and each object reference-name pair is called a name binding. Name bindings may be organized under naming contexts. Naming contexts serve the same organizational function as a file system subdirectory. All bindings are stored under the initial naming context.

In the repository server, the server class first creates an ORB instance. The server class then creates a servant instance and connects the ORB instance with the servant instance. The servant implements all repository interfaces provided to the client. After that, the

sever class registers the ORB instance in the naming context under the name "ProNet_SNMP". Then, the server just waits for invocations from the clients.

The naming tree is as below. Because, currently, the clients communicate with only one local repository, there is only one name binding in the tree.



Each client creates an ORB instance, looks up "ProNet_SNMP" in the naming context, and then receives a reference to that CORBA object. Then, the client can connect with the repository server and invoke the interfaces provided by that server.

2.2. Client interface

The client interface is a device used to communicate with the repository. It was designed to be simple, so it could be both versatile, and easy to develop. It needs to formulate and send a snmp request in a form recognizable to the repository.

Each client has a call back object supported by idltojava CORBA object model. The client ORB instance is connected with its call back object. The call back object class provides the interface that could be invoked by the repository. When the repository gets the snmp responses from the polling station, it can invoke the CallBack method in the client call back object class, send the data back to the client directly.

Each client can sent out snmp requests to multiple routers. For each snmp request, there is a RequestHandler class to handle the formulation of the request and sending the request. The RequestHandler contains the information of the snmp request, including the name of the router, the name of the output data file, the polling interval, the total polling time and the snmp oids. All information is read by the client from the configure files specified by the users.
In current implementation, the DOORS system requires decimal format snmp oids in the configure files.

Besides the polling information, each snmp request also has a request id, initialized by the repository. Each request id is related with a client call back object. The mapping information is stored at the repository side. When the repository tries to send the snmp response to the client, it can reach the correct client based on the mapping information.

Whenever the client tries to send a snmp request to the repository, it has to tell the repository the request id for that snmp request and the time to send the request. The format of the snmp request is simple:

<router name> <interval> <polling time> <snmp oid> <snmp oid> …

In order to avoid letting the client hanging too long time for one request, timeout is set at the client side. Based on the time spending during the round trip between the repository and the polling station, timeout is currently set as five polling intervals. After sending the snmp request to the repository, if after five intervals, there is still no snmp response back, the client will give up for that request and inform the users about the timeout. If during the five intervals, there is snmp response back, the client will reset the time stamp and continue to wait for next polling result for that snmp request in next five intervals. Thus, the longest waiting time for one specific request is the sum of polling time and five intervals.

When the repository gets the snmp response from the polling station, it calls the CallBack method in the call back object and pushes the snmp data back to the correct client. The CallBack method simply prints the snmp data into the specific output data file. The name of the output data file is send back with the snmp response by the repository. If no specified output data file provided by the user, all response data will be print out on the screen.

In short, we hope the client interface as simple as possible, just sending requests and print out responses. We use the repository to manage the complication during the communication.

2.3. Repository

The repository is responsible for the controlling of agents and coordinating requests from different clients. The basic functionality of the Repository is getting the snmp requests from the clients, creating the mobile agent, forwarding the requests to the polling station through the mobile agent, getting the snmp responses from the polling station through the mobile agent again, and then sending back the responses to the corresponding clients.

In current design, we didn’t consider the local storage system in the repository. Whenever a snmp request coming, it is forwarded to the polling station through the mobile agents.

In order to handle the transfer of requests and responses between the clients and the repository, each repository contains four hash tables in its servant class. All hash tables use request id as the key.

Hash table <idToCallBack>: contains the client call back objects related to each request id. When the repository tries to send the snmp response back to the client, it finds out the call back object of the client according to the request id. In this way, the repository can handle multiple clients and multiple snmp requests synchronously.

Hash table <DataFiles>: contains the names of output data files for the requests. Since the snmp requests could be send out synchronously, and the snmp responses could be send back out of order, the repository should tell the client which output file should contain the corresponding snmp results. This hash table provides the mapping between the request id and the output file name.

Hash table <DataReadyFlagTable>: contains the flags to check if the polling result of the request has been pushed back. The flag is used while the client tries to determine if turning on the timeout. Initially, for all request id, the flags are set to "notready", which means that no polling result sent back. Whenever one polling result of one snmp request is sent back to the client, the flag is set to "ready". The client will check the flag in each polling interval while waiting for the snmp response. If after five intervals, the flag has never been set to "ready", the client turns on the timeout message, and give up for the corresponding snmp request. If during five intervals, the flag has been set to "ready", the client knows that one polling result of the request has been sent back. The client, then, will continue to wait for the next polling result of that request if there are more results coming. At the mean time, if there are more polling results coming for the request, the flag is reset to "notready" after the client check it. Then, in next five intervals, the client will check the flag again to determining the timeout. Therefore, the flag of each request id needs to be adjusted several times during the waiting period.

Hash table <PollCounts>: contains the number of the results for each snmp request. It is the same as the number of the times to poll, determined by the polling interval and the polling time.

$$\text{Count} = (\text{polling time}) / (\text{polling interval}).$$

The reason to keep the mapping information between polling count and each request id is that each request needs to poll the router one or several times. Both the repository and the client need to know if all results for the specific request have been sent back. Then, they can determine when to stop the communication.

The request id is set with sequential number started from zero while starting the repository server. If the repository is always alive after starting, the value of the request id will increase accumulating. If the repository is restarted, the value of the request id will start from zero again.

In order to keep the space of all hash tables as small as possible, whenever all polling results of a snmp request have been sent back to the client or timeout is turned on for one request at the client side, the contents of corresponding request id are removed from all hash tables.

After the repository get the snmp request from the client, it creates the mobile agent, and forwards the request to the polling station through the mobile agent. The method

implements this is collect() in the servant class. In current implementation, we assume that only one polling station connected with the repository.

The request forwarded to the polling station contains the information of request id, the name of the router, snmp oids, polling interval and polling time. The protocol between the repository and the polling station is as below:

-I <request id>
-r <router name>
-o <snmp oid> <snmp oid> …
-i <polling interval>
-t <polling time>
-c <number of times to poll> # optional, if polling time provided.

After forwarding the snmp request, the repository will listen to the action of the mobile agent. As soon as the mobile agent gets the snmp response from the polling station, it will automatically tell the repository that the response is coming. A method called processMessage() in servant class implements this functionality. The method is invoked automatically by the mobile agent when the event occurs (In this case, the message is sent back.). Then, the message is parsed, if it is the snmp results needed by the client, it is sent back to the client.

To push the snmp result back to the client, the repository needs to know which client the data will be sent to and which output data file the client will write the data to. The method implements this function is called PushdataToClient() in the servant class. The message pushed back to the client by the method contains the information of request id, polling time, router name, snmp oid, and snmp value. Based on the request id, the repository finds out the corresponding client call back object and the output file name, then send out the massage and output file name to the client directly.

3. Experiment result

The whole program consist three components, client and server part, mobile agent part, and snmp polling station part.

We test the functionality of the whole system by combining the three parts together. As the previously stated, we retrieved SNMP data from the routers being used as targets. The router in our tests was a SPARC Ultra 10 workstation (cobol.cs.rpi.edu). The polling station and the repository were one SPARC Ultra 10 workstation (apl.cs.rpi.edu). Cobol and Apl are in the same subnet. The user terminals were also SPARC Ultra 10 workstations in the same subnet.

The input is the configure file created by the users. The format of the sample of the configure file is as below:

SEGMENT

```
cobol.cs.rpi.edu                    # router name
cobol                               # output data file name
10                                  # polling interval (second)
20                                  # polling time (second)
2.2.1.10.2                          # snmp oids
2.2.1.16.2
4.3.0
4.9.0
4.10.0
END
```

In practice, each configure file can contains several segments with the same format, each start as “SEGMENT” and end as “END”. Each segment will be constructed to one snmp request.

The output file contains the router name, client request time (system time at the client side), polling time (system time at the polling station side), snmp oids and snmp values. The format of the sample of the output is as below:

```
Request time:   976779125481            # system time of sending the request
cobol.cs.rpi.edu    976779145953        # <router name> <system time to poll first time>
2.2.1.10.2      3768518045              # <snmp oid> <oid value in the first poll>
2.2.1.16.2      2173943303
4.3.0   12673183
4.9.0   13933169
4.10.0  18383281


cobol.cs.rpi.edu    976770155971    # <router name> <system time to poll second time>
2.2.1.10.2      3768518045          # <snmp oid> <oid value in the second poll>
2.2.1.16.2      2173943303
4.3.0   12673183
4.9.0   13933169
4.10.0  18383281
```

From the basic experiment, we can get the conclusion that the current implementation version of DOORS system can have the functionality to send snmp requests to the polling station and get back the snmp response from the polling station through the mobile agent. The communication between the clients and the repository is implemented and managed correctly. But there is still one problem in the whole DOORS system. Whenever the mobile agent takes back the snmp message to the repository, the servers in the mobile agent part need to be restart before the new snmp request sent out to the repository from the client, otherwise, there are many extra snmp responses will be sent back.

4. System requirements and running procedure

4.1. System requirements

The whole program needs to be compiled and run on the Solaris 2.8 platform.

The hosts are SPARC Ultra 10 workstations. The repository server and the polling station should be the same host.

The compiler for the CORBA idl file (in client and repository server part) is idltojava, provided by SUN Java IDL.

The compiler for all java files (in all three parts) is javac, provided by SUN Java version 1.2.2.

4.2. Procedure to run the whole program

To make the menu easily to understand, the description format is set:
Italic format is the command to run the program.

Put the files to implement the dynamic interface between the client and the repository under same directory. For example, at the directory of:
/cs/jins/Master/mycode/.

There are three files.

snmpCB.idl: The CORBA idl file, containing the interfaces needed in the communication between the client and the repository.
ProNet_clientCB.java: The client application file.
ProNet_serverCB.java: The servant file of the repository server.

Copy the CORBA idl compiler “idltojava” to the same directory.

Put all files from other two parts (mobile agent part and the polling station part) in the same directory. For example, at the directory of:
/cs/jins/Master/agentcode/code/.

Create subdirectory in the agentcode directory to contain the class files. The path of the subdirectory is : /cs/jins/Master/agentcode/class/bin/.

Copy the snmp library class files into the bin/ subdirectory. The path is:
/cs/jins/Master/agentcode/class/bin/AdventNetSnmpV3/.

Modified the file .bashrc in the home directory to set the CLASSPATH. Set the CLASSPATH as below in .bashrc file:

CLASSPATH="/cs/jins/Master/agentcode/class/bin:$CLASSPATH"
SNMPPATH=$HOME/Master/agentcode/class/bin/AdventNetSnmpV3

CLASSPATH=.:$CLASSPATH:$SNMPPATH/jdk1.2_classes:$SNMPPATH/classes:$HOME/Master/agentcode/class/bin
export CLASSPATH

Now to compile and run and program.

First, compile the files from the mobile agent and polling station parts, start the servers in mobile agent part.

Go to the directory: /cs/jins/Master/agentcode/class/bin/, compile the java files.

Javac –d ./ ../../code/.java*

Copy all class files from polling station parts (Snmp*.class) into the directory that contains the files from the client and the repository part. In this case, copy Snmp*.class into the directory: /cs/jins/Master/mycode/.

On the host “apl.cs.rpi.edu”, open a console, start the lookup server for mobile agent part. Lookup server tries to find out which polling station the snmp request needs to go. In our implementation, only one polling station is considered in the mobile agent part.

java msu.magent.lookup.SuperServer 9000

(9000 is the default port number for lookup server. Or choose 8000 as port number.)

On the host “apl.cs.rpi.edu”, open another console, start the local server for mobile agent. Go to the directory: /cs/jins/Master/agentcode/, create a subdirectory named “pollstation”, go to the directory “pollstation”, and create a subdirectory named “classdir”. Now the path is: /cs/jins/Master/agentcode/pollstation/classdir/.

Go to the directory: /cs/jins/Master/agentcode/pollstation/, start the local server.

java msu.magent.LocalServer.Server 8040

(8040 is the default port number for local server. Different number could be chosen, such as 8050, 8060…)

Second, compile the files from client and repository server part, and run the program.

Go to the directory: /cs/jins/Master/mycode/.

Compile the CORBA idl file.

idltojava snmpCB.idl

The idltojava compiler will create a subdirectory named “snmpCBApp/”, which contains the java files generated by idltojava at the same time. Those files are necessary while implementing the interfaces of idl file.

Compile the java files at the same directory.

*javac * **.java snmpCBApp/***.java*

Start the naming server.

On the host “apl.cs.rpi.edu”, open a new console, go to the directory: /cs/jins/Master/mycode/, start the naming server in background.

tnameserv –ORBInitialPort 1050 > namingfile &

(1050 is the default port number for name server. It could be chosen as different number, such as 1060, 1070, …. The namingfile contains the information of IP address and port number of the naming server. Those information is required while running the client application. When the name server is started on one host, it doesn’t need to restart again, unless the user exits from that host.)

At the same console and same directory, start the repository server.

java ProNet_serverCB –ORBInitialPort 1050

Start the client application.

On the host “rexx.cs.rpi.edu”, go to the directory: /cs/jins/Master/mycode/, run the client program.

java ProNet_clientCB –n config6 namingfile

(“-n” means no community string required, config6 is the name of the configure file, namingfile is the name of the file that containing port number and IP address of the naming server.)

At the same directory, an output data file named “cobol” will be generated by the program. The file contains the information of router name, request time, polling time, snmp oids, and snmp values. The sample is shown at part 3.

5. Future work

There are lots of points need to be considered in the future work on DOORS system. In the part of communication between the client and the repository, we assumed that the clients only communicate with one local repository server, and only one polling station is connected with the repository. But, in the real network, there should be many repositories. Each repository can manage some routers. Similar, each repository can connect with

several polling stations. Therefore, during the communication between the clients and the repository, the mapping information between the repository and the router, and between the repository and the polling stations, is necessary to be figured out and managed. However, the mechanism to manage the information is not mature.

For example, for the mapping information between the repository and the router, who needs to store the information? If let the client store the information, the client will choose the correct repository based on the mapping information, and send the snmp requests to that repository. But, the drawback is that it is hard to keep the mapping information consist in different client, since there are thousands of clients, and each of them communicate with the repository dynamically.

If put the mapping information at the repository side, there is still consist problem in different repository. One way to deal with this is putting the mapping information at one repository. All clients communicate with that initial repository first, finds out which repository the requests need to be sent to, then the initial repository will send out the snmp requests to the correct repository. In this case, the communication between two repositories need to consider very carefully.

Similar, for the mapping information between the repositories and the polling stations, the consist issue is also needed to consider. Meanwhile, since the repository should tell the mobile agent which polling station it should go, the mobile agent should have the capability to find out the addresses of different polling station. How to provide the mapping information to the mobile agent is another issue need to consider while integrate the repository part with the mobile agent part.

Another point need to consider is that the number of mobile agents created by the repository. In current integration between the client-server part and the mobile agent part, only one agent is created to forward the snmp requests to the polling station. If one repository could map to several polling stations, then multiple mobile agents needed to be created in one repository. How to handle those agents in the repository is a big issue to consider.

6. Summary

DOORS system provides a middleware between independent network managers and network routers. To support the network management, DOORS system needs to use a dynamic interface between the client and the server and mobile agents. This report described the design and the implementation of the communication between the clients and the repository server in DOORS system. The communication is a two-way connection. The clients send multiple requests to the repository synchronously, the repository server forwards the request to the polling station through the mobile agent. When the mobile agent take the response back to the repository, the repository sends the data back to the correct client.

The implementation of the communication used the CORBA object model, and java programming language.

In the experiments, we integrated three components (communication part between the client and the server, mobile agent part and polling station part) together. The DOORS system can perform basic function to handle the management of the network. But there is still problems need to consider and deal with.

References

[1] Alan Bivens, Li Gao, Mark F.Hulber and Boleslaw K.Szymanski . Agent-Based Network Monitoring. *Rensselaer Polytechnic Institute, Troy, NY.*

[2] Alan Bivens, Patrick Fry, Li Gao, Mark F. Hulber, Quifen Zhang and Boleslaw K.Szymanski. Distributed Object-Oriented Repositories for Network Management. *Rensselaer Polytechnic Institute, Troy, NY.*

[3] Juan Pavon. Building Telecommunications management Applications with CORBA. *Universidad Complutense de Madrid.*

Appendix a: snmpCB.idl source code

```
module snmpCBApp
{
  interface snmpCallback {
```

```
    void CallBack (in string data, in string outputfile);
  };

  interface ProNet_SNMP {
    void collect(in string snmp_command, in string request_id,
                 in long long request_time);
    void PushdataToClient(in string callbackdata);
    string initId(in snmpCallback clientref, in string dataFile);
    string getDataReadyFlag(in string request_id);
    boolean getPollCounter(in string request_id);
    void removeCallBack(in string request_id);
    void removeDataFlag(in string request_id);
  };
};
```

Appendix b: ProNet_clientCB.java

```
/*****************************************************************************
****
```

```
**********************************************************************
*****
**
** ProNet_clientCB.java
** created by Shan Jin
** December 12, 2000
**
** ProNet_clientCB.java provides the interfaces to users to access the
** interfaces provided by the repository server. The client (user) sends
** snmp requests to the repository server, then waits for the responses from
** the server. The snmp request information is specified in the configure
** files provided by the user. The configure files include the information of
** router name, output data file name, the interval time to poll, the total
** polling time and the snmp oids.
**
** Each client can send out multiple snmp requests, each request has an
** identification id related with the client. This identification id is used
** to wake up the correct client while snmp responses sent back from the server.
** Also, the multiple snmp requests could be sent out synchronously.
**
** To avoid letting the client wait for too long time, there is a timeout
** mechanism that setting timeout time for each request to five-interval time.
**
**********************************************************************
******
**********************************************************************
*****/


import snmpCBApp.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

import java.lang.*;
import java.io.*;
import java.util.*;


/**********************************************************************
*****
*
* snmpCallbackServant class implements the CallBack function of snmpCallback
* interface in snmpCB CORBA idl file.
*
* method: void CallBack(String data, String dataFile);
*
```

```
**********************************************************************
*****/

class snmpCallbackServant extends _snmpCallbackImplBase
{
   /****
    To print out the snmp response into the corresponding output file. The name
    of the output file is specified by the user.
    ****/

   public void CallBack (String data, String dataFile)
   {
         /* Write the push back data into the output file if the name of the
           file provided by the user
         */
         if (dataFile != null) {
            PrintWriter outFile = null;
            try {
                  outFile = new PrintWriter
                     (new BufferedWriter
                           (new FileWriter(dataFile, true)));
            } catch (Exception e) {
                  System.out.println("Error in open " + dataFile);
                  e.printStackTrace();
            }

            outFile.println(data);

            try {
                  outFile.close();
            } catch (Exception e) {
                  System.out.println("error in close " + dataFile);
                  e.printStackTrace();
            }
         }

      /* if no output file name provided, print out the snmp result on the
        screen.
      */
         else
            System.out.println(data);
   }
}
```

```
/**************************************************************************
****
*
* class RequestHandler deals with one request each time. It collects the
* information on one router for many snmp variables, buffers them, then
* sends the request out to the repository server.
*
* RequestHandler also deals with timeout issue for each snmp request. The
* client can handle multiple snmp requests synchronously through
* RequestHanler.
*
* Data member: strRouter, strRepository, strVars, strDataFile,
*              nInterval, nRunTime, RequestTime, corbaServer,
*              request_id, snmpCallbackRef, ifDataReady,
*              client_orb, namingcontext.
*
* Methods: boolean startServer();
*          synchronized void run();
*
**************************************************************************
***/

class RequestHandler extends Thread
{
   String strRouter = null;          // router name
   String strRepository = null;      // repository name
   String strVars = null;           // all variable names
   String strDataFile = null;      // output data file name
   int nInterval;                 // polling interval
   int nRunTime;                    //  polling time
   long RequestTime;

   ProNet_SNMP corbaServer = null;    // repository server

   // request id for each client. (Note: each client can have several
   // request id.)
   String request_id;
   snmpCallbackServant snmpCallbackRef;      // callback object for client
   String ifDataReady;                     // flag to check if the snmp
                                  // response has been pushed back
                                 // to the client
   ORB client_orb;              // ORB object for the client
   String namingcontext;       // naming server that the client needs to
                          // connect with
```

```
   RequestHandler()
   {
      nInterval = 0;
      nRunTime = 0;
         RequestTime = 0;
         ifDataReady = "notready";
   }

   RequestHandler(String strRo, String strRe, String strV, String strF, int nInt, int nRun,
ORB orb, String namingstr)
   {
      strRouter = new String(strRo);
      strRepository = new String(strRe);
      strVars = new String(strV);
      strDataFile = new String(strF);

      nInterval = nInt;
      nRunTime = nRun;
         RequestTime = 0;
         ifDataReady = "notready";

         client_orb = orb;
         namingcontext = namingstr;
   }


   /****
    Connect  the client object with the naming server that registered by the
    repository server.

    Get the request id from the repository server, buffers the snmp request,
    then sends out the snmp request to the server.
   ****/

   public boolean startServer()
   {
         if ((strRouter==null) || (strVars==null) || (strRepository==null) ||
            (nInterval==0) || (nRunTime==0))
         {
             System.out.println("Error in start CORBA server in short of parameters.");
             return (false);
         }

         // Initial naming context
         try {
            org.omg.CORBA.Object client_objRef =
```

```
                    client_orb.string_to_object(namingcontext);
            NamingContext client_ncRef =
                    NamingContextHelper.narrow(client_objRef);

            // resolve the object reference in Naming
            NameComponent nc = new NameComponent("ProNet_SNMP", "");
            NameComponent path[] = {nc};
            corbaServer = ProNet_SNMPHelper.narrow(client_ncRef.resolve(path));

            // connect to callback object
            snmpCallbackRef = new snmpCallbackServant();
            client_orb.connect(snmpCallbackRef);
          } catch (Exception ex) {
            System.out.println("Exception during initialing naming and connecting with
callback object");
            ex.printStackTrace();
            return(false);
          }

          /* Initialize the request id, each request id is related with one
          corresponding client callback object.
        */
          request_id = corbaServer.initId(snmpCallbackRef, strDataFile);

          /* Create the snmp command string */
          StringBuffer strComm = new StringBuffer(strRouter);
          strComm.append(" ").append((new Integer(nInterval)).toString());
          strComm.append(" ").append((new Integer(nRunTime)).toString());
          strComm.append(" ").append(strVars);

          // call collect, send out the snmp request
          try {
          // time to send out the snmp request
            RequestTime = System.currentTimeMillis();

            // Write the request time into the output file if the name of the
            // file provided by the user
            if (strDataFile != null) {
                PrintWriter outFile = null;
                try {
                  outFile = new PrintWriter
                        (new BufferedWriter
                          (new FileWriter(strDataFile, false)));
                } catch (Exception e) {
                  System.out.println("Error in open " + strDataFile);
                  e.printStackTrace();
```

```
                }

                outFile.println("Request time:\t" + RequestTime);

                try {
                   outFile.close();
                } catch (Exception e) {
                   System.out.println("error in close " + strDataFile);
                   e.printStackTrace();
                }
             }
          // if no data file name provided, print the infor to the screen.
             else
                System.out.println("Request time:\t" + RequestTime);

          // Send out the snmp request
             corbaServer.collect (strComm.toString(), request_id, RequestTime);
          } catch (Exception e) {
             e.printStackTrace();
             return (false);
          }

          return (true);
       }


       /****
        run() synchronuously send out snmp request. For each snmp request, it
        sets the timeout mechanism. If during five-interval time, the client
        doesn't get any response from the server, the client will abandon the
        delayed snmp request.
       ****/

       public synchronized void run()
       {
          long nSleepMillis = nInterval * 1000;
          long timeStampH = 0;
          long timeStampE = 0;
          long sleepmillis = 0;

          // the system time to receive a polling result.
          long ReceiveTime = 0;

          // The time to stop waiting for the polling result
          long StopTime = 0;
```

```
// flag to check if more snmp responses will come for one particular
// snmp request
    boolean countValid;

    // Send out the snmp request
    if (!startServer()) {
            System.out.println("Error in start repository server");
            return;
       }

    StopTime = RequestTime + nRunTime*1000 + nInterval*5*1000;
    countValid = corbaServer.getPollCounter(request_id);

/* If there is more time to wait and more reponses to come,
   continue to wait.
*/
while (ReceiveTime <= StopTime && countValid) {

      /* wait 5-interval time for the push back data.
      If time out then give up the request, and inform the server to
       remove the requeatId and the callback object from the
      hashtable.
      */
     for (int i=0; i<5; i++) {
            timeStampH = System.currentTimeMillis();

            try {
               ifDataReady = corbaServer.getDataReadyFlag(request_id);

               if (ifDataReady == null) {
                     System.out.println("no flag matches to " + request_id);
                     return;
               }
               else {
                     // Inform the client if the data has been pushed back
                     if(ifDataReady.equals("ready")) {
                        ReceiveTime = System.currentTimeMillis();
                        break;
                     }
               }
            } catch (Exception e) {
               System.out.println("error in getDataReadyflag");
               e.printStackTrace();
               return;
            }
```

```
        timeStampE = System.currentTimeMillis();
        sleepmillis = timeStampE - timeStampH;
        sleepmillis = nSleepMillis - sleepmillis;

        try {
           if (sleepmillis > 0) {
                 sleep(sleepmillis);
           }
        } catch (Exception e) {
           System.out.println("error in sleep function");
           e.printStackTrace();
           return;
        }
}

/* check if timeout */

// not timeout
if (ifDataReady.equals("ready")) {

/* data has been pushed back, remove the flag from
        the hash table or adjust the status of the flag
        depending upon if all snmp results for this
        request id have been sending back.
      */
      corbaServer.removeDataFlag(request_id);
}

// timeout
else {

      /* Write timeout message into the output file if the name of
        the file provided by the user
      */
      if (strDataFile != null) {
         PrintWriter outFile = null;
         try {
                outFile = new PrintWriter
                   (new BufferedWriter
                         (new FileWriter(strDataFile, true)));
         } catch (Exception e) {
                System.out.println("Error in open " + strDataFile);
                e.printStackTrace();
         }

         outFile.println("Timeout for the request");
```

```
                    try {
                        outFile.close();
                    } catch (Exception e) {
                        System.out.println("error in close " + strDataFile);
                        e.printStackTrace();
                    }
                }
                else
                    System.out.println("Time out for the request");

                /* timeout for this request. Remove the callback object,
                   outputdata file name, dataready flag and snmp polling
                   counter from the corresponding hashtable.
                */
                try {
                    corbaServer.removeCallBack (request_id);
                    break;
                } catch (Exception e) {
                    System.out.println ("error in removeCallBack");
                    e.printStackTrace();
                    return;
                }
            }

            countValid = corbaServer.getPollCounter(request_id);
        }
    }

}       // end of class RequestHandler


/**************************************************************************
******
*
* class ProNet_clientCB reads in naming server information file and router
* configure files, then let each RequestHandler to deal with snmp requests.
*
**************************************************************************
*****/

public class ProNet_clientCB {
    public static void main(String args[]) {

        String usage = "java ProNet_clientCB <community group, -n for none> <configer
file> <naming server file>";
```

```
if (args.length < 3) {
   System.out.println("Usage: " + usage);
   System.exit(0);
}

try {
   ORB orb = ORB.init(args, null);

   // Read in naming server file first
   String namingCntx = null;
   StringTokenizer tokenNM = null;

   BufferedReader fin = null;
   try {
         fin = new BufferedReader
            (new FileReader(args[2]));
   } catch (Exception e) {
         System.out.println("Error in open naming file");
   }

   do {
         try {
            namingCntx = fin.readLine();
         } catch (Exception e) {
            e.printStackTrace();
         }

         if (namingCntx != null)
            tokenNM = new StringTokenizer(namingCntx, ":");
   }
   while ((namingCntx != null) && !(tokenNM.nextToken().equals("IOR")));

   try {
         fin.close();
   } catch (Exception e) {
         e.printStackTrace();
   }

   if (namingCntx == null) {
         System.out.println("cannot fine the naming context");
         System.exit(0);
   }

   // Read config file, create RequestHandler for each request
   BufferedReader configFileReader = null;
```

```
        Vector vecHandler = new Vector();

        String strRouter = null;
        String strFile = null;
        int nInterval = 0;
        int nRunTime = 0;
        StringBuffer strV = null;
        String strRepository = null;

        String strConfigLine = null;

        try {
            configFileReader = new BufferedReader
              (new FileReader (args[1]));
        } catch (Exception ce) {
            System.out.println("Error in open config file " + args[1]);
        }

        while (true) {
            // read beginning of a segment
            do {
              try {
                  strConfigLine = configFileReader.readLine();
              } catch (Exception e) {
                  e.printStackTrace();
              }
            }
            while ((strConfigLine != null)
&& !(strConfigLine.trim().equals("SEGMENT")));

            if (strConfigLine == null)      // end of file
              break;

            int countline = 0;
            while ((strConfigLine != null) && (countline <= 4)) {

              try {
                  strConfigLine = configFileReader.readLine();
              } catch (Exception e) {
                  e.printStackTrace();
              }

              if (strConfigLine == null)
                  break;

              countline++;
```

```
                switch (countline) {
                case 1:       // read router name
                      strRouter = new String(strConfigLine.trim());
                      break;
                case 2:       // read data file name
                      strFile = new String(strConfigLine.trim());
                      break;
                case 3:       // read polling interval
                      nInterval = (new Integer(strConfigLine.trim())).intValue();
                      break;
                case 4:       // read run time
                      nRunTime = (new Integer(strConfigLine.trim())).intValue();
                      break;
                default:
                }
             }

             if (strConfigLine == null)
                break;

             strV = new StringBuffer();

             while ((strConfigLine != null) && !(strConfigLine.trim().equals("END")))
{
                strV.append(strConfigLine.trim() + " ");

                try {
                      strConfigLine = configFileReader.readLine();
                } catch (Exception e) {
                      e.printStackTrace();
                }
             }

             RequestHandler rhNew = new RequestHandler
                (strRouter, "repository", strV.toString(), strFile,
                 nInterval, nRunTime, orb, namingCntx);
             vecHandler.addElement(rhNew);
          }   // end of read config file

          try {
             configFileReader.close();
          } catch (Exception e) {
             e.printStackTrace();
          }
```

```
            // start all RequestHandler
            int nRounters = vecHandler.size();
            for (int k=0; k<nRounters; k++) {
                ((RequestHandler)vecHandler.elementAt(k)).start();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }   // end of main()
}       // end of class ProNet_clientCB
```

Appendix c: ProNet_serverCB.java

```
/**************************************************************************
*******
**************************************************************************
*******
**
** ProNet_serverCB.java
** created by Shan Jin
** December 12, 2000
**
** ProNet_serverCB.java implements the functionalities of the repository server
** to get the snmp requests from the clients, forward the snmp requests to the
** polling station through mobile agents, get the snmp responses from the
** mobile agents, then send back the snmp responses to the clients.
**
** Each repository server can handle multiple clients synchronuously.
**
**************************************************************************
*******
**************************************************************************
******/


import snmpCBApp.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

import java.lang.*;
import java.io.*;
import java.util.*;

import msu.magent.lookup.*;
import msu.magent.io.*;
import msu.magent.Agent.*;
import msu.magent.LocalServer.*;


/**************************************************************************
*****
*
* class ProNet_SNMPServant implements the interfaces of ProNet_SNMP in snmp
* CORBA idl file. The communications between the clients and the repository
* server and the combination between the repository server with the mobile
* agent are implemented. The interfaces are provided to the clients.
*
* Data memebers: idToCallBack, DataFiles, DataReadyFlagTable, PollCounts,
*                client_id_num, agent, as.
```

```
*
* Methods: String initId(snmpCallback callbackobj, String datafile);
*          void collect(String snmp_command, String requestid, long requesttime);
*          void processMessage();
*          void PushdataToClient(string data);
*          String getDataReadyFlag(String id);
*          boolean getPollCounter(String id);
*          void removeCallBack(String id);
*          void removeDataFlag(String id);
*
**********************************************************************
******/

class ProNet_SNMPServant extends _ProNet_SNMPImplBase implements
   AgentMessageListener
{
   // All hashtables use request_id as the keys.

   // contains client call back objects
   Hashtable idToCallBack = new Hashtable();

   // contains output data file name for each snmp request
   Hashtable DataFiles = new Hashtable();

   /* contains flags to check if the corresponding request has been pushed
      back, if yes, the flag is set to "ready", if not, the flag is set to
      "notready".
   */
   Hashtable DataReadyFlagTable = new Hashtable();

   /* contains the number of snmp results for one snmp request, determined
      by polling interval time and total polling time.
      count = (polling time) / (interval);
   */
   Hashtable PollCounts = new Hashtable();

   int client_id_num = 0;      // used to calculate the request id

   // Each repository server now can only handle one send agent.
   SnmpAgent agent;
   AgentSender as;


   /****
    initId() initializes a number format id for each snmp request.
    It then creates three hashtables to contains the information of
```

```
 callback object, datareadyflag and output data file names for each
 snmp request.
****/

public String initId(snmpCallback callbackobj, String datafile)
{
        String requestId = new String(String.valueOf(client_id_num++));
        String datareadyflg = "notready";

        /* using requestId as key, push each callback object and output data
           data file name into the corresponding cell in the hash table.
        */
        idToCallBack.put(requestId, callbackobj);
        DataFiles.put(requestId, datafile);

        /* Intially, set all dataready flag to false, since no request data
           available now */
        DataReadyFlagTable.put(requestId, datareadyflg);

        return requestId;
}


/****
 Forwards the snmp requests from the client to the mobile agent, then listen
 to the snmp responses from the moblile agent.

 Create a String vector to contain all snmp command information.
   The meaning of flags in the vector is as below. Each element
   of the vector is of type String.

   -I requestId
   -r routerName(s)
   -o oids for snmp variables
   -i polling interval time
   -t polling time
   -c count (number of times to poll)

   Note: count field is optional if polling interval time is provided
   from the configuration file.
  ****/

public void collect(String snmp_command, String request_id, long clientReqTime)
{
        Vector CommVec = new Vector();
        Vector varVec = new Vector();
```

```
    String snmpinfo = null;
    int count;

// creates the vector to hold the snmp request information
    try {
        CommVec.addElement("-I");
        CommVec.addElement(request_id);

        /* Parse the snmp command */
        StringTokenizer commtoken = new StringTokenizer(snmp_command);

        // get router name
        CommVec.addElement("-r");
        CommVec.addElement(commtoken.nextToken());

        // get the interval time
        String interval = commtoken.nextToken();
        CommVec.addElement("-i");
        CommVec.addElement(interval);

        // get the polling time
        String time = commtoken.nextToken();
        CommVec.addElement("-t");
        CommVec.addElement(time);

        // get the snmp variables
        CommVec.addElement("-o");
        while (commtoken.hasMoreTokens()) {
              snmpinfo = commtoken.nextToken();
              CommVec.addElement(snmpinfo);
              varVec.addElement(snmpinfo);
        }

        // Create the hash table of number of times to poll for each
    // request.
        count = ((new Integer(time)).intValue()) /
              ((new Integer(interval)).intValue());

        PollCounts.put(request_id, (new Integer(count)));
    } catch (Exception e) {
        e.printStackTrace();
    }


    /* Forwards the snmp request to the mobile agent, then listens to the
   snmp response from the agent.
```

```
   */
      try {
         String[] filelist =
         {"SnmpAgent", "SnmpCfg", "SnmpMngr", "SnmpPoll"};
         AgentAddress[] aa;

      // finds out the polling station
         aa = Server.lookupService();

         agent = new SnmpAgent (CommVec);
         agent.setHostList(aa);

         as = new AgentSender(agent);
         as.setFileList(filelist, 4);

      // sends out the snmp request information to the polling station
      as.sendAgent();
         System.out.println("listen now");

      // The repository server listens to the snmp response from the
      // mobile server.
         as.listen(this);
      } catch (Exception e) {
         System.out.println("Unknow host");
         e.printStackTrace();
      }

}


/****
 Parses the message sent back by the mobile agent. This method will be
 called automatically by the agent while the agent sends back the message
 to the repository server. Then data are pushed back to the client by
 the repository server.
****/

public void processMessage()
{
      try {
         String msg = as.readMsg();
         System.out.println("message objtained: " + msg);
         switch(msg.charAt(0)) {
         case 'm':
               String datamsg = as.readMsg();
               System.out.println("data message: " + datamsg);
```

```
                PushdataToClient(datamsg);
                //as.sendMsg("hello back");
                break;
            case 'b':
                as.sendMsg("ok");
                agent = (SnmpAgent)as.readAgent();
                System.out.println("Agent Returned");
                break;
        case 'e':
                as.ser.close();
                //System.exit(0);
                break;
            default:
                System.out.println("Unknown Message");
            }
        } catch (IOException ioe) {
            System.out.println("msg reception -exception");
        } catch (SyncException se) {
            System.out.println("msg reception -syncex");
        }
}


/****
  data string format:
  Requestid pollingStartTime router snmpOid oidVaule, snmpOid oidValue,...

  PushdataToClient() call the CallBack interface of the client, pushes the
  snmp response to the client.
****/

public void PushdataToClient(String data)
{
        StringTokenizer datatoken = new StringTokenizer(data, ",");

        /* First token contains the information of id, polling start time,
           router name, snmpOid and corresponding value.
        */
        String varstr = (datatoken.nextToken()).trim();
        StringTokenizer vartoken = new StringTokenizer(varstr);

        String id = vartoken.nextToken();
        String pollstartTime = vartoken.nextToken();
        String routername = vartoken.nextToken();

        String snmpoid = vartoken.nextToken();
```

```
StringBuffer oidvalue = new StringBuffer(vartoken.nextToken());

while (vartoken.hasMoreTokens()) {
   oidvalue.append(" ").append(vartoken.nextToken());
}

// Append the result into the pushback message
StringBuffer databuff = new StringBuffer(routername);
databuff.append("\t").append(pollstartTime);
databuff.append("\n");
databuff.append(snmpoid).append("\t");
databuff.append(oidvalue.toString()).append("\n");

/* Other token just contains the information of snmpOid and the
  corresponding value
*/
while (datatoken.hasMoreTokens()) {
   varstr = (datatoken.nextToken()).trim();
   vartoken = new StringTokenizer(varstr);

   snmpoid = vartoken.nextToken();
   oidvalue = new StringBuffer(vartoken.nextToken());

   while (vartoken.hasMoreTokens()) {
         oidvalue.append(" ").append(vartoken.nextToken());
   }

   // Append the result into the push back message
   databuff.append(snmpoid).append("\t");
   databuff.append(oidvalue.toString()).append("\n");
}

snmpCallback client_callobj = (snmpCallback)idToCallBack.get(id);
Integer countobj = (Integer)PollCounts.get(id);

// number of times to poll
int count;

if (client_callobj != null && countobj != null) {
   count = countobj.intValue();

   // if there are polling results to be sent back
   if (count > 0) {
         // get the output data file name
         String datafileN = (String)DataFiles.get(id);
```

```
                // call the call back interface of the client
                client_callobj.CallBack(databuff.toString(), datafileN);

                count--;
                PollCounts.remove(id);
                PollCounts.put(id, new Integer(count));

                /* Check if the polling has done, if yes, remove the callback
                   object and output data file name from the hash tables.
                */
                if (count == 0) {
                    idToCallBack.remove(id);
                    DataFiles.remove(id);
                }
            }
        }
        else
            System.out.println ("no callobj matching the id or all results of the snmp
variables have been sending back.");

        // reset the dataReady flag in the DataReadyFlagTable
        String dataflg = (String)DataReadyFlagTable.remove(id);
        dataflg = "ready";
        DataReadyFlagTable.put(id, dataflg);
    }


    /****
     Gets the string flag of the request. If flag is "ready", it means that
     the snmp result has been sent back to the client. If flag is "notready",
     it means no snmp result has benn sent back.
    ****/

    public String getDataReadyFlag (String id)
    {
        String dataFlg = (String)DataReadyFlagTable.get(id);
        return dataFlg;
    }


    /****
     Get the flag that if all results of one particular snmp request have
     been sent back to the client.
    ****/

    public boolean getPollCounter(String id)
```

```
{
      Integer cntobj = (Integer)(PollCounts.get(id));

      if (cntobj != null)
         return (true);
      else
         return (false);
}


/****
  Client time out, server removes the corresponding callback object
  and data file name from the Hash table. Also, removes the dataReady
  flag from the DataReady flag table.
****/

public void removeCallBack(String id)
{
      idToCallBack.remove(id);
      DataFiles.remove(id);
      DataReadyFlagTable.remove(id);
      PollCounts.remove(id);
}


/****
  After push back the data, check if polling for this request has done.
  If all results of the request have been sent back, remove the counter
  from PollCounts hashtable and remove the flag from DataReadyFlagTable.
  If not, means there are snmp results not being sending back,
  then, change the status of flag to "notready" again.
****/

public void removeDataFlag(String id)
{
      // check polling counter
      Integer countobj = (Integer)PollCounts.get(id);
      int count;

      if (countobj != null) {
         count = countobj.intValue();

         if (count > 0) {
               String dataflg = (String)DataReadyFlagTable.remove(id);
               dataflg = "notready";
               DataReadyFlagTable.put(id, dataflg);
```

```
            }
            else {
                DataReadyFlagTable.remove(id);
                PollCounts.remove(id);
            }
        }
        else
            System.out.println("no correnspondingcountobj");
    }


}    // end of ProNet_SNMPServant class


/**************************************************************************
******
*
* class ProNet_serverCB creates an ORB object for the repository server, then
* registers the server object to the naming server. When the client connects
* with the naming server, the client can communicate with the repository
* server. The communication between the clients and the repository server is
* synchronized.
*
**************************************************************************
*****/

public class ProNet_serverCB {
    public static void main (String args[]) {
        try {
            ORB orb = ORB.init(args, null);

            // register the servant with the ORB
            ProNet_SNMPServant corbaServant = new ProNet_SNMPServant();
            orb.connect(corbaServant);

            // get the root naming context
            org.omg.CORBA.Object serverObjRef =
                orb.resolve_initial_references("NameService");
            NamingContext serverncRef = NamingContextHelper.narrow(serverObjRef);

            // bind the object reference in naming
            NameComponent servernc = new NameComponent("ProNet_SNMP", "");
            NameComponent serverpath[] = {servernc};
            serverncRef.rebind(serverpath, corbaServant);

            // wait for invocations from clients
```

```
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }
        } catch (Exception e) {
            System.err.println("error: " + e);
            e.printStackTrace(System.out);
        }
    }
}    // end of ProNet_serverCB class
```