
RAPP Tutorial

Using the NetPhantom RAPP API

Version 5.5

Document Revision 1

17 October 2011

Nexum Technologies SARL

NetPhantom®

Version 5.5

© Copyright Nexum Technologies SARL, 2011. All rights reserved.

Information in this document is subject to change without notice. Companies, names, and data used in examples are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Nexum Technologies.

Nexum Technologies may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you the license to these patents, trademarks, copyrights, or other intellectual property rights except as expressly provided in any written license agreement from Nexum Technologies.

Phantom® and NetPhantom® are registered trademarks of Nexum Technologies SARL. Java is a trademark of Sun Microsystems Incorporated. ActiveX, Microsoft, Windows are either registered trademarks or trademarks of Microsoft in the United States and/or other countries. IBM is a registered trademark of International Business Machines Corporation.

Nexum Technologies SARL

**World Trade Center, Nice – Sophia Antipolis
1300, route des Crêtes – BP 255
FR-06905 Sophia Antipolis Cedex
FRANCE**

Telephone: +33 954 15 30 20
Fax: +33 959 15 30 20
Web: <http://netphantom.com>
E-mail: info@nexum-tech.com

Support

Germany: +49 69 401 590 987
Sweden: +46 8 525 00 123
World-wide: +33 954 15 30 20
E-mail: support@netphantom.com

Contents

Preface – Intended Audience	3
1 What is RAPP?	5
1.1 How to Use RAPP.....	5
1.2 An Overview of the Workflow.....	5
A Continuous RAPP Session	6
An Interrupted RAPP Session	7
1.3 A Quick Overview of the RAPP API.....	8
2 The RAPP API Sample	11
2.1 The Emulated Host Application	11
The Sign On Screen	12
The Main Menu Screen.....	12
The Search Customer Screen	13
The Customers – Search Result Screen.....	13
The Customer Data Screen.....	14
2.2 The NetPhantom Application	14
The Sign On Panel	15
The Main Menu Panel	16
The Customer Search Panel	16
The Customers – Search Result Panel.....	16
The Customer Data Panel.....	18
2.3 The Java Application	19
Imports Needed in the Application	19
Set the Static Trace Mode	19
Get an Instance of the RemoteApplication.....	19
Connecting to the NetPhantom Server	20
How to Get Information about a Screen and Its Fields	20
Manipulating Global Variables	21
How to Call an Object, Disconnect and then Reconnect for the Reply.....	23
Accessing Listboxes.....	24
How to Read Data from a Panel.....	26
Manipulating Panel Controls.....	27
Entry Fields.....	27
Radio Buttons	27
Checkboxes	27
Listboxes.....	27
Control Attributes	29
Logging off from the Host	30
Calling a User Class.....	30
Event Logging.....	33
Summary	33
3 Setting up the NetPhantom Server	34
Configuring the NetPhantom Application.....	34
Setting up the Remote Application	34
Defining the Resource for the Remote Application	35

Preface – Intended Audience

*There are thousands of reasons why you cannot do what you want.
All you need is one reason why you can*

Willies R Whitney

This document, the *How to use the NetPhantom RAPP API*, is intended as a tutorial for Java application developers and NetPhantom developers. It will demonstrate how to write code for external applications to enable them to communicate with the NetPhantom Server using the Remote Application Protocol (RAPP).

1 What is RAPP?

*Reality is a mysterious changing thing,
because one's perception of it never remains the same.*

Joe Tan

This chapter will explain what the NetPhantom RAPP API is, and what its purpose is.

The RAPP API (Remote Application Protocol Application Programming Interface) is an API that can be used by other applications to interact with NetPhantom applications running on a NetPhantom Server.

Before the introduction of the RAPP API in the NetPhantom Server, it was impossible for other applications to use the NetPhantom Server to access host-based systems. This meant that all applications had to have their own module for accessing the host, even if they wanted to access the same host application. To write modules for host access was often a complicated and time-consuming process.

With the introduction of the NetPhantom RAPP API, it is now possible to use the NetPhantom Server as an integration hub. All the other server-side applications that want to access the host-based systems can now talk to the NetPhantom Server and use the NetPhantom Server for host access. This also means that when you add RAPP support to an application, you automatically get access to host systems on both AS/400 and Mainframe computers. Writing a module that can use RAPP is also a much easier, hence cheaper, process than writing a module that directly accesses the host.

1.1 How to Use RAPP

Explaining how to use RAPP is the purpose of this entire document and its accompanying sample. But in brief, you import a RAPP package that includes classes with methods that are used to manipulate the NetPhantom Application. This package, the RAPP API, makes it possible to navigate through the NetPhantom Application, to get/set data in the NetPhantom Application, and to directly access the host screens.

This means that the whole host application, and all its data, is available to every application that interfaces with the RAPP API. We call the application that uses the RAPP API to communicate with the NetPhantom Server a Client Application, even though it might be a server to other clients. From NetPhantom's point of view, it is a client.

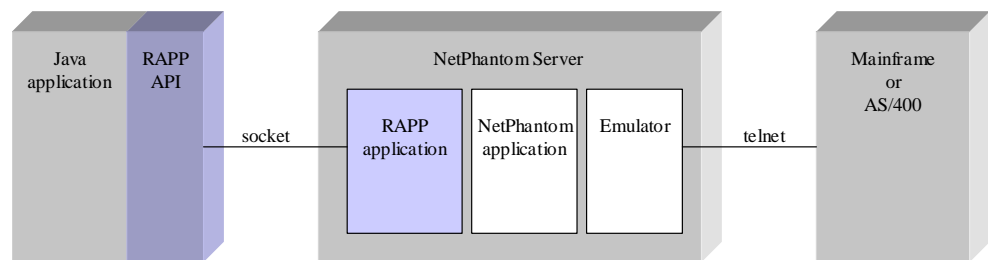


Fig. 1 How RAPP fits in the overall picture.

1.2 An Overview of the Workflow

Before getting into the details of using RAPP, we must take a short look at the workflow.

There are actually two different workflow cases. The first is when the Client Application that invokes the RAPP session is able to maintain the connection throughout the whole session. This can be called a Continuous RAPP Session. The second is when the Client

Application for some reason is unable to maintain the connection. This can be called an Interrupted RAPP Session.

A Continuous RAPP Session

The workflow for a Continuous RAPP Session consists of the following steps. The steps are described from the point of view of the Client Application.

1. Create a *RemoteApplication*.
2. Specify the hostname and port to connect to the NetPhantom Server.
3. Connect to the NetPhantom Server.
4. Create a *RAPPTransaction*.
5. Create a *RAPPCallHost*, *RAPPCallPanel*, *RAPPCallObject* or *RAPPCallClass*.
6. Do a request, with a *request_XXX* method in one of the above *RAPPCallXXX* classes.
7. Add the action to the *RAPPTransaction*.
8. Repeat steps 5-7 until all requests have been added to the *RAPPTransaction*.
9. Create a *RemoteTransaction* from the *RAPPTransaction*.
10. Do a *requestTransaction* from the *RemoteApplication*. A *requestID* is returned.
11. Get a reply from the *RemoteApplication* with the use of the *requestID*. The reply is returned as a new *RemoteTransaction*.
12. Call the *RemoteTransaction*'s *updateTransaction* method to update all *RAPPCallXXX* objects.
13. Disconnect from NetPhantom server.

The figure below presents an example of the workflow as a sequence diagram. In this example, the content of a listbox is requested. Steps 2, 3 and 13 are left out of the diagram.

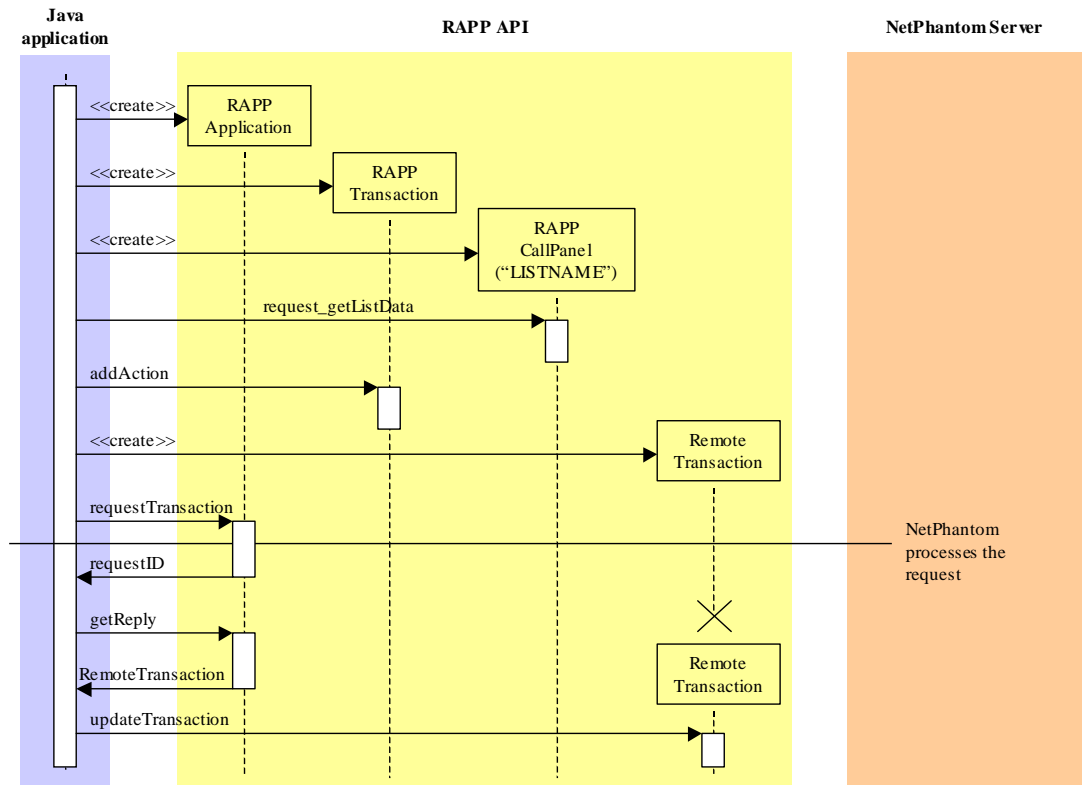


Fig. 2 A sequence diagram showing the workflow for a Client Application running in Continuous mode, requesting data from a listbox in a NetPhantom application.

An Interrupted RAPP Session

The workflow for an Interrupted RAPP Session consists of the following steps. The steps are described from the point of view of the Client Application. After step 14 the connection to the NetPhantom Server has been terminated.

1. Create a RemoteApplication.
2. Specify the hostname and port to connect to the NetPhantom Server.
3. Connect to the NetPhantom Server.
4. Create a *RAPPTransaction*.
5. Create a *RAPPCallHost*, *RAPPCallPanel*, *RAPPCallObject* or *RAPPCallClass*.
6. Do a request, with a request_XXX method in one of the above *RAPPCallXXX* classes.
7. Add the action to the *RAPPTransaction*.
8. Repeat steps 5-7 until all requests have been added to the *RAPPTransaction*.
9. Create a *RemoteTransaction* from the *RAPPTransaction*. A requestID is returned.
10. Do a requestTransaction from the *RemoteApplication*. A requestID is returned.
11. Get a reply from the *RemoteApplication* with the use of the requestID. The reply is returned as a new *RemoteTransaction*.

12. Call the *RemoteTransaction*'s `updateTransaction` method to update all *RAPPCallXXX* objects.
13. Get the `sessionID` (and host address and port number if backup servers are used).
14. Send the `sessionID` to the web-client
15. Close the connection to the server.
16. Send the `sessionID` from the web-client to the Java application.
17. Create a new *RemoteApplication* with the saved `sessionID`, host address and port number.
18. Connect to the NetPhantom Server.
19. Create a *RAPPTransaction*.
20. Continue with next request and reply.
21. And so on ...

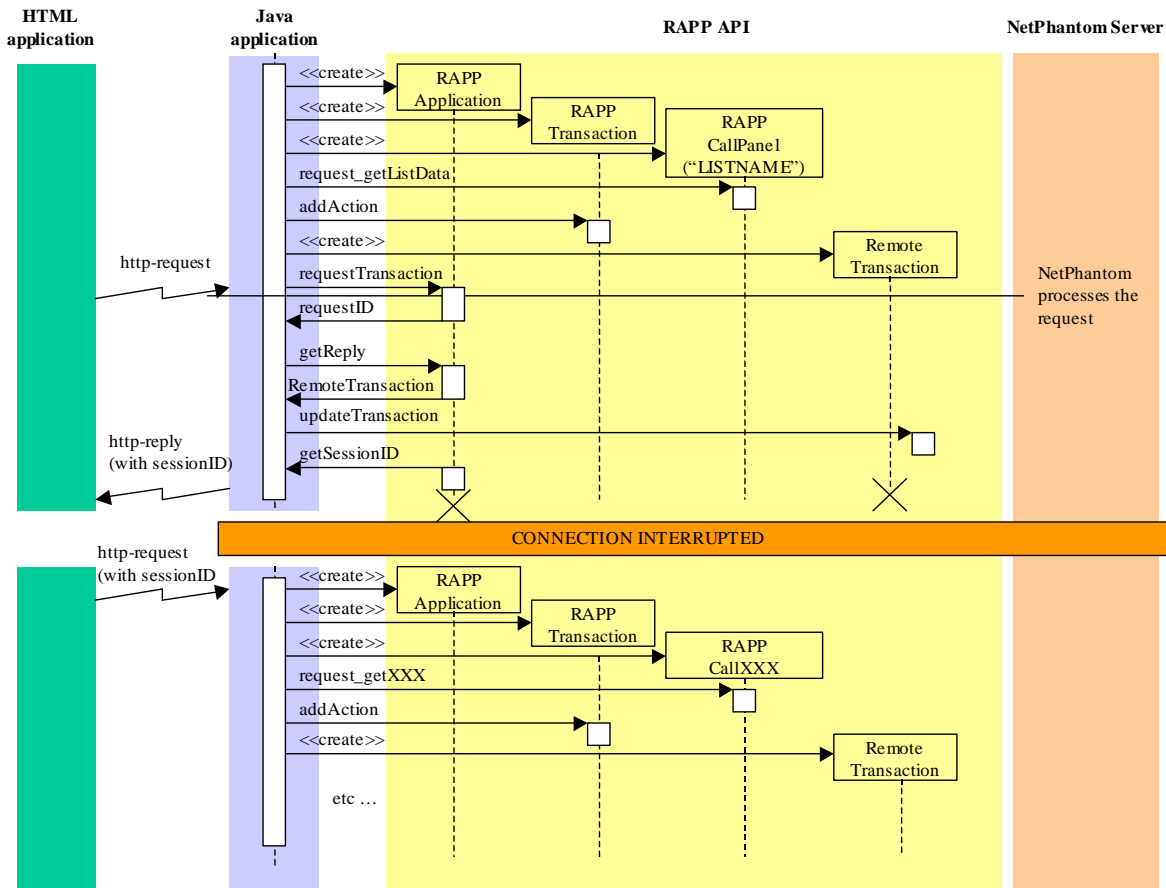


Fig. 3 A sequence diagram showing the workflow for a Client Application running in Interrupted mode, requesting data from a listbox in a NetPhantom application before disconnecting, and then reconnecting for the next request.

1.3 A Quick Overview of the RAPP API.

The whole RAPP API is available in the `se.entra.phantom.rapp` package.

The package contains two interfaces and 19 classes. They are as follows.

Interfaces

RAPPEnumerationItem
RAPPHostReply

Classes

RAPPAction
RAPPClass
RAPPClassHost
RAPPClassObject
RAPPClassPanel
RAPPElement
RAPPEnumeration
RAPPGlobVar
RAPPGlobVars
RAPPHostCell
RAPPHostCells
RAPPHostScreen
RAPPLine
RAPPLines
RAPPNameSpaceItem
RAPPTransaction
RemoteApplication
RemoteTransaction
SSLRemoteApplication

For a detailed description about these interfaces and classes, see the JavaDoc for the `se.entra.phantom.rapp` package.

2 The RAPP API Sample

A teacher who is attempting to teach without inspiring the pupil with a desire to learn is hammering on a cold iron.

Horace Mann

In this chapter we will take a look at the RAPP API Sample included in the NetPhantom package. This sample covers all the functionality in the RAPP API.

To run the NetPhantom Application used by the sample, connect your web browser to the NetPhantom Server and select the *RAPP API Client* item under *Browser Applications* in the *Menu*.



Fig.4 Selecting the RAPP API Sample from the NetPhantom Server homepage

You can also look at the Java source code for the sample. You will find it under *Browser Applications – RAPP API Client* in the *Menu*.

To run the sample RAPP application, open up a Command Prompt window. Under Windows, run the batch file `ExecClient.bat` or `ExecSSLClient.bat`. With other operating systems, run the commands similar to the ones in the batch files. The parameters to the batch files are `“hostname:port”`.

2.1 The Emulated Host Application

This sample uses a number of pre-recorded emulator screens (what are called EE-catches in NetPhantom) instead of a live host.

Since this tutorial assumes familiarity with NetPhantom applications and how to create them, we will only take a brief look at the host screens used as host application. There will be a short description of NetPhantom fields defined for these screens, because they will be accessed from the Java application.

The Sign On Screen

The first screen is the Sign On screen. The corresponding NetPhantom Screen is called SIGNON.

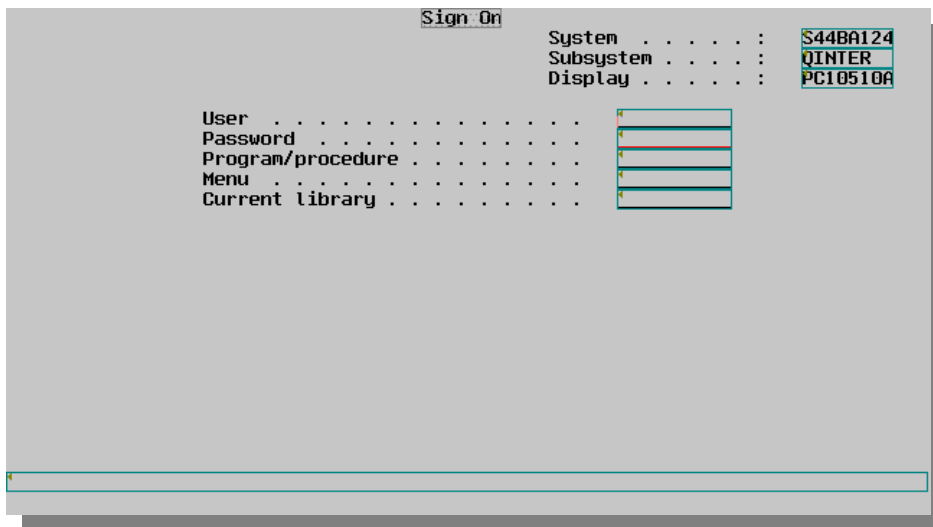


Fig. 5 The Sign On host screen

The following NetPhantom Host Fields have been defined for this screen.

SYS	The name of the System
SUBSYS	The name of the Subsystem
TERM	The Display name
USER	The input field for the username.
PASSWORD	The input field for the user's password.
PROGPROC	The input field for the program or procedure.
MENU	The input field for the menu name.
CURLIB	The input field for the current library.
S_MSG	The message field on line 24.

The Main Menu Screen

The second screen is the Main Menu screen. The corresponding NetPhantom Screen is called MAINMENU.



Fig. 6 The Main Menu host screen

For this screen only one NetPhantom Host Field has been defined.

CMD The entry field for the selection.

The Search Customer Screen

The third screen is the Search Customer screen. The corresponding NetPhantom Screen is called SEARCHCUSTOMER.

Fig. 7 The Search Customer host screen

The following NetPhantom Host Fields have been defined for this screen.

F1 The input field for the customer number.
F2 The input field for the customer name.

The Customers – Search Result Screen

The fourth screen is the Customers - Search Result screen. The corresponding NetPhantom Screen is called CUSTLIST.

CUSTNO	NAME	LAST PUR	AMOUNT
1003900200	SMITH, JOHN PAUL	30/12/94	23.85
1100300031	ADAMS, CHARLES K	05/05/94	1,000.00
1203900303	PETERSON, MARY	25/11/93	124.78
1204000440	SAINT-CLAIR, GEORGE	14/00/94	145.91
1303900200	LYTTON, SIR CHARLES	05/06/94	19,210.50
1500300031	DURBAN, PETER	18/12/93	450.21
1603900303	FRISWELL, ANNE W	05/02/92	23.34
1604000303	MCDONALD, THEODORE R	11/11/94	133.90
2039932043	TRAVERS, H DAVID	22/01/94	45.00
2139294943	ABRAMOVITCH, LLEWELLYN	03/08/94	23.90
2149000000	GOODMAN, IAN S	25/04/94	230.25
2500004000	GREEN, JOHN B	18/09/94	234.45
3400000001	JENKINS, TERRENCE W	20/12/94	98.50
4567020000	PETERS, MAXINE	17/12/94	1,390.05
4567992931	SMITH, LAURA B	01/03/94	805.30
4707388292	WHITNEY, SHELDON	02/05/94	697.65
5028380000	O'SHEA, LINDA	30/01/94	243.43
5905000532	SIMONS, JULIAN P	10/07/94	85.50
6000100000	HAMMER, GEORGE M	23/09/94	69.23

PF2=END PF7=BACKWARD PF8=FORWARD
25 CUSTOMERS LISTED

Fig. 8 The Customers – Search Result host screen

The following NetPhantom Host Fields have been defined for this screen.

SEL	The column for the selection field in the customer list.
CUSTNO	The column for the customer number.
NAME	The column for the customer name.
LASTPUR	The column for the last purchase dates.
AMOUNT	The column for the amounts.
MORE	The field that indicates if more list-pages are available.

The Customer Data Screen

The last screen is the Customer Data screen. The corresponding NetPhantom Screen is called CUSTDATA.

This screen will be used for testing several controls, so there are some special NetPhantom Host Fields on this screen. They will make more sense when we look at the NetPhantom panels and the Java code.

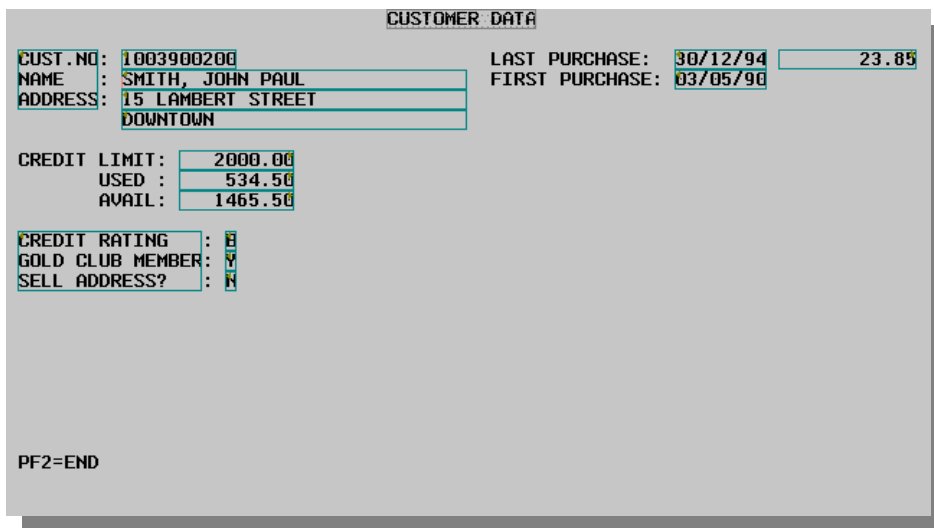


Fig. 9 The Customer Data host screen

The following NetPhantom Host Fields have been defined for this screen.

CUSTNO	The customer number.
NAME	The customer name.
ADDR1	The customer's address, first line.
ADDR2	The customer's address, second line.
CREDUSED	The amount of credit that has been used.
CREDREST	The amount of credit available.
CREDRATE	The customer's credit rating.
GOLDMBR	Indicates if the customer is Gold Club member.
SELLADDR	Indicates if the address can be sold.
LASTPUR	The date of the last purchase.
AMOUNT	The amount of the last purchase.
FIRSTPUR	The date of the first purchase.
COL1	A column for the labels for the Cust. No, Name and Address.
COL2	A column for the labels for the Credit Rating, Gold Club Member and Sell Address.

2.2 The NetPhantom Application

The NetPhantom application used in this sample consists of five panels. We will take a short look at them, and the objects connected to them. By building a NetPhantom

application that can be tested manually, it is possible to test it thoroughly and to make sure that it is bug free before it is accessed from another external application.

The Sign On Panel

This is the first panel in the application where you login to the system (actually no username or password is required in this sample).

The screenshot shows a terminal window with the following content:

```

System      SYS
Subsystem   SUBSYS
Display     TERM

User        USER
Password    PASSWORD
Program/procedure  PROGPROC
Menu        MENU
Current library  CURLIB

Normal logon
Logon to Customer List
  
```

Fig. 10 The Sign on Panel

You can continue from this panel in two ways. If you press the *Normal Logon* button, you will continue to the next panel, the *Main Menu* panel. The *Normal Logon* button just sends an Enter key to the host. If you click the *Logon to Customer List* button, an object called *SIGNON* will be activated, and will bring you directly to the *Customers - Search Result* panel. The code for the *SIGNON* object is listed below.

```

/* This object is called either by pressing "Logon to Customer List" */
/* button or by the Remote Application */

Parse Arg argId, argMsg, argStr

say "#SIGNON arguments: argId="argId", argMsg="argMsg", argStr="argStr

/* Enter logon info from RemoteApplication */
ui=GlobVarGet('USERID')
pw=GlobVarGet('PASSWORD')
say "#SIGNON userid='"ui"' password '"pw'"

/* Note: the password is not displayed on terminal screen normally */
rc=HostSetFld('', 'USER' ,ui)
rc=HostSetFld('', 'PASSWORD',pw)

/* Move to main screen */
rc=HostSend('@E')
rc=HostWait(5)

/* Move to customer initial screen */
rc=HostSetFld('', 'CMD', '2')
rc=HostSend('@E')
rc=HostWait(5)

/* Skip search customer */
rc=HostSend('@E')
rc=HostWait(5)

if HostGetScreen()=="CUSTLIST" then
    return "OK, on screen CUSTLIST"

return "Error, wrong screen!"
  
```

The Main Menu Panel

This is the panel from which you select the function you want to perform. In this sample application, only *Customers* and *Exit* work.

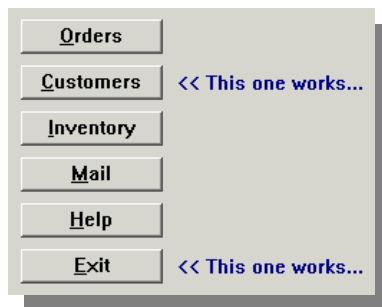


Fig. 11 The Main Menu Panel

Clicking the *Customers* push button brings you to the *Customers search* panel.

Clicking *Exit* push button will bring you back to the *Sign on* panel

The Customer Search Panel

From this panel, you start the search for customers. Since we are working with pre-recorded host screens, you don't have to enter customer number or name, you will always get the same search result.



Fig. 12 The Customer search Panel

Clicking the *OK* push button will bring you forward to the *Customers – Search Result* panel.

Clicking the *END* push button will bring you back to the *Main Menu* panel.

The Customers – Search Result Panel

This panel lists the result of the customer search in the previous panel. From here you can either choose to view the search result in the list, page up and down in the list (the pre-recorded search result consists of two pages) or select to view the data for the first customer in the list.

In a live application you would select the customer that you would want to view, but in this sample we only have one pre-recorded screen with customer data.

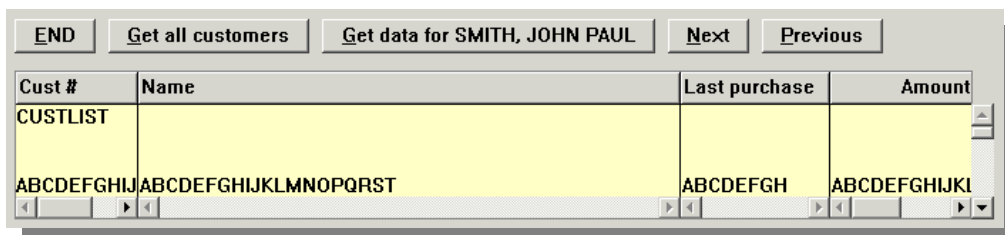


Fig. 13 The Customers – Search Result Panel

To view the search result in the listbox, press the *Get all customers* push button. This will activate a REXX-object called GETCUSTS, which inserts the result from host into the list box.

```

/* Gets all customers from host into the listbox */
/* This object can handle multiple screens of data */

Parse Arg argId, argMsg, argStr

/* Clear the list of any previous data */
rc=PanListClear('CUSTLIST')

/* Get all pages and keep track of how many screens are */
/* "paged" using PF8 */
pageCount=0
done=0
do while done=0
  /* Fill the list from the host */
  listEnd=0
  do line=1 to 19
    custno=HostGetFld('','CUSTNO',line)
    if custno='' then
      do
        listEnd=1
        leave
      end
    name =HostGetFld('','NAME' ,line)
    lastpur=HostGetFld('','LASTPUR',line)
    amount =HostGetFld('','AMOUNT' ,line)
    rc=PanListInsertData('CUSTLIST',0,custno || '09'X || name || '09'X ||
                        lastpur || '09'X || amount);
  end

  /* Check for end of list, make sure there's a MORE text below list */
  if listEnd=1 | HostGetFld('','MORE')<>'MORE' then
    done=1
  else
    do
      /* Get next page */
      rc=HostSend('@8')
      rc=HostWait(5)
      pageCount=pageCount+1
    end
  end

  /* Return OK */
  return 0

```

To view data for the first customer in the search result, press the *Get data for SMITH, JOHN PAUL* push button. This will activate a REXX-object named CUSTDAT, which performs the selection in the result list, and then continues to the *Customer data* panel.

```

/* Gets customer data */

Parse Arg argId, argMsg, argStr

/* Let's assume we want the first customer, */
/* otherwise the code becomes more complicated */
/* and clutters the sample code */

custIndex=0

/* In real world, one might have to PF8 to the correct page */
rc=HostSetFld('','SEL','s',custIndex+1)
rc=HostSend('@E')
rc=HostWait(5)

/* Return OK */
return 0

```

Clicking the *Next* push button gets the next list page, if one is available.

Clicking the *Previous* push button gets the previous list page, if one is available.

Clicking the *END* push button gets you back to the *Customer Search* panel.

The Customer Data Panel

This panel presents the customer data. There are also a lot of extra controls that are placed here so that we can see how they are accessed from the Java application later.

The screenshot shows a graphical user interface for customer data. It includes several input fields for customer information: CUST.NO, CUSTNO, NAME, ADDRESS (ADDR1, ADDR2), LAST PURCHASE (LASTPUR), AMOUNT, FIRST PURCHASE (FIRSTPUR), CREDIT LIMIT (CREDLIM), CREDIT RATING (C), GOLD CLUB MEMBER (G), and SELL ADDRESS? (S). There are also buttons for 'END' and 'Logoff'. A 'Test controls' section contains checkboxes for 'Checked' and 'Unchecked', radio buttons for 'Radio 1', 'Radio 2', and 'Radio 3', a 'COMBO' dropdown menu, and buttons for 'Visible -> VISIBL' and 'Hidden -> HIDDE'. At the bottom, there are two table-like structures labeled 'INLIST' and 'OUTLIST', each with columns 'Column 1' and 'Column 2', and a row 'COL1' with 'COL2'. An 'Output OUTPUT' label is also present.

Fig. 14 The Customer Data Panel

Clicking the *END* push button gets you back to the *Customers – Search Result* panel by activating a REXX-object named PF2.

Clicking the *Logoff* push button brings you back to the Sign on panel by activating a REXX-object named LOGOFF.

```

/* Logs off from the host by pressing PF2 until the MAINMENU */
/* and the 'X' (Exit) action */

Parse Arg argId, argMsg, argStr

/* Make sure the host is not busy due to CallControl(END) */
/* that sends PF2 from the CUSTDATA panel and moves the */
/* host to the CUSTLIST screen */
rc=HostWait(5)

/* Press PF2 until MAINMENU */
do while HostGetScreen() <> 'MAINMENU'
  rc=HostSend('@2')
  rc=HostWait(5)
end

/* Do the Exit action */
rc=HostSetFld('', 'CMD', 'X')
rc=HostSend('@E')
rc=HostWait(5)

return 0

```

2.3 The Java Application

It is finally time to look at the Java application that accesses the NetPhantom application via the RAPP API. This sample application is a small standalone Java application made up of a single class called `TestAPI`.

We will split up the code and describe it one piece at a time. All the JavaDoc comments have been removed to make it clearer. Only code that is significant to understanding the use of the RAPP API will be described here. To see the whole Java application, see the online documentation.

Imports Needed in the Application

Classes that have to be imported into the Java application can be divided into two types: those needed for the XML-document that the RAPP API uses internally, and those that make up the RAPP API package. Here each class has been specified separately, instead of using the '*' wildcard, in order to make it clear which classes are needed by this application.

The imports are as follows:

```
import java.io.IOException;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;

import se.entra.phantom.rapp.RAPPElement;
import se.entra.phantom.rapp.RAPPTransaction;
import se.entra.phantom.rapp.RAPPCallClass;
import se.entra.phantom.rapp.RAPPCallHost;
import se.entra.phantom.rapp.RAPPCallObject;
import se.entra.phantom.rapp.RAPPHostScreen;
import se.entra.phantom.rapp.RAPPHostCell;
import se.entra.phantom.rapp.RAPPHostCells;
import se.entra.phantom.rapp.RAPPGlobVar;
import se.entra.phantom.rapp.RAPPGlobVars;
import se.entra.phantom.rapp.RAPPCallPanel;
import se.entra.phantom.rapp.RAPPLine;
import se.entra.phantom.rapp.RAPPLines;
import se.entra.phantom.rapp.RemoteApplication;
import se.entra.phantom.rapp.RemoteTransaction;
```

Set the Static Trace Mode

During the development and test period, it can be helpful to see trace information in the console window. The following line turns on this feature.

```
RemoteApplication.setStaticDebugMode(true);
```

Get an Instance of the RemoteApplication

To be able to communicate with the NetPhantom Server, we will need to have an instance of a *RemoteApplication*. In the sample, this is done with the static method `createRemoteApplication`. This method needs the name of the remote application in the NetPhantom server as a parameter.

```
protected RemoteApplication createRemoteApplication(String app)
{
    return new RemoteApplication(app);
}
```

This static method will be called with the following code to create the *RemoteApplication* instance.

```
RemoteApplication ra=testAPI.createRemoteApplication(APP_NAME);
```

In the sample the application name has been hard coded as follows.

```
private static final String APP_NAME = "SAMPLES_RAPPAPI";
```

Connecting to the NetPhantom Server

In this sample there is only one NetPhantom server, so to connect to it, the *RemoteApplication* will only need to know the server name, and which port to use.

The server name is the name of the server machine, and the port number is the same port number that the standard NetPhantom clients use to access the server.

The following code will connect to the NetPhantom server:

```
ra.setServer(host,port);
ra.connect();
```

where the parameter *host* is a String containing the server name and *port* is an integer containing the port number. If backup servers are used, the server names and the port numbers must be passed as arrays.

How to Get Information about a Screen and Its Fields

After the sample application has connected to the NetPhantom server, it fetches information about the Sign on screen. In order to interact with the NetPhantom Server via the RAPP API, we must use something called a *RAPPTransaction*. The *RAPPTransaction* wraps requests that are to be transmitted to the server. We begin by creating a new *RAPPTransaction*.

```
RAPPTransaction trans=new RAPPTransaction();
```

All the requests that we want to pass to the NetPhantom server will be added to this transaction.

In order to fetch information about the current host screen, we need to create a request based upon *RAPPCallHost*.

```
RAPPCallHost ch1=new RAPPCallHost(true);
trans.addAction(ch1);
```

It is also possible to limit the amount of information to fetch from the host screen. Below follow two examples.

To create a request that will fetch information about all the NetPhantom Host fields with names starting with 'S' on the SIGNON screen, specify 'S*' as the second parameter.

```
RAPPCallHost ch2=new RAPPCallHost("SIGNON","S*");
trans.addAction(ch2);
```

To get information about a specific field on a host screen, call *RAPPCreateHost* with the screen name and the field name as parameters.

```
RAPPCallHost ch3=new RAPPCallHost("SIGNON","TERM");
trans.addAction(ch3);
```

After these requests have been added to the *RAPPTransaction*, a *RemoteTransaction* will be created from the *RAPPTransaction*. Then the *RemoteApplication* will process the transaction.

```
RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);
```

The *requestID* is a reference that will be used to get the answer to this request. We get the reply by calling the *getReply* method in the *RemoteApplication*. Because we want to wait for the reply before continuing, we set the second parameter to *true*.

```
rt=ra.getReply(requestID,true);
```

The reply is returned as a new *RemoteTransaction*. This transaction can now be used to update the *RAPPCallHost* instances used for the requests.

```
rt.updateTransaction(trans);
```

We can then get information about the host screen in the following way. In this sample code the information is printed out in the standard output device.

```
RAPPHostScreen screen=ch1.getScreen();
System.out.println("\nScreen information:");
+"\n error, lock    = "+ch1.hasError()+"", "+ch1.isLocked()
+"\n status        = "+ch1.getStatus()
+"\n screen name    = "+screen.getScreenName()
+"\n current field  = "+screen.getFieldName()
+"\n text at line 1 = "+screen.getScreenData().substring(0,80).
trim();
```

In a similar way we can print out information about all the host fields with names starting with 'S', that were fetched into the *RAPPCallHost ch2*.

```
RAPPHostCells hcs=ch2.getFields();
int count=hcs.getCount();
System.out.println("\nHost fields starting with 'S'");
+"\n count = "+count);
for ( int ii=0; ii<count; ++ii )
{
RAPPHostCell hc=hcs.getCell(ii);
System.out.println("\n field "+ii+":");
+"\n name          = "+hc.getName()
+"\n length         = "+hc.getLength()
+"\n hostLength     = "+hc.getHostLength()
+"\n column         = "+hc.getColumn()
+"\n hostColumn     = "+hc.getHostColumn()
+"\n hostFieldColumn = "+hc.getHostFieldColumn()
+"\n row           = "+hc.getRow()
+"\n hostRow       = "+hc.getHostRow()
+"\n hostFieldRow  = "+hc.getHostFieldRow()
+"\n protected    = "+hc.isProtected()
+"\n hidden       = "+hc.isHidden()
+"\n color        = 0x"+Integer.toHexString(hc.getColor())
+"\n value       = "+hc.getValue();
}
```

And finally we can print out the information about the host field 'TERM', that was fetched into *RAPPCallHost ch3*.

```
RAPPHostCell hc=ch3.getField();
System.out.println("\nTERM field:");
+"\n name          = "+hc.getName()
+"\n length         = "+hc.getLength()
+"\n hostLength     = "+hc.getHostLength()
+"\n column         = "+hc.getColumn()
+"\n hostColumn     = "+hc.getHostColumn()
+"\n hostFieldColumn = "+hc.getHostFieldColumn()
+"\n row           = "+hc.getRow()
+"\n hostRow       = "+hc.getHostRow()
+"\n hostFieldRow  = "+hc.getHostFieldRow()
+"\n protected    = "+hc.isProtected()
+"\n hidden       = "+hc.isHidden()
+"\n color        = 0x"+Integer.toHexString(hc.getColor())
+"\n value       = "+hc.getValue();
```

Manipulating Global Variables

In this section we will take a look at how it is possible to manipulate NetPhantom Global Variables. All manipulation of global variables in NetPhantom is done with the use of instances of the *RAPPGlobVar* class.

We will again start by creating a new *RAPPTransaction*.

```
RAPPTransaction trans=new RAPPTransaction();
```

To get information about all global variables, with a name containing the String ‘_HOST’, do the following.

```
RAPPGlobVar gv1=new RAPPGlobVar("*_HOST*",RAPPGlobVar.GLOB_VAR_GET);
trans.addAction(gv1);
```

The following code sets two global variables.

```
trans.addAction(new RAPPGlobVar("USERID", "TestUser"));
trans.addAction(new RAPPGlobVar("PASSWORD", "Secret" ));
```

In the sample we will fetch information about all the global variables again, so that we can check that the setting of the global variables for username and password worked.

```
RAPPGlobVar gv2=new RAPPGlobVar("*",RAPPGlobVar.GLOB_VAR_GET);
trans.addAction(gv2);
```

You can also delete global variables. In this sample code we will delete all global variables containing the String ‘_SERVER_’ somewhere in the variable name.

```
trans.addAction(new RAPPGlobVar("*_SERVER_*",
                                RAPPGlobVar.GLOB_VAR_DELETE));
```

Finally we get information about all the global variables again to see what’s left.

```
RAPPGlobVar gv3=new RAPPGlobVar("*",RAPPGlobVar.GLOB_VAR_GET);
trans.addAction(gv3);
```

To request these actions on the global variables, we create a *RemoteTransaction* from the *RAPPTransaction*, which is send to the *RemoteApplication*.

```
RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);
```

And again we will wait until all requests have been processed.

```
rt=ra.getReply(requestID,true);
```

When all the requests have been processed, we need to update the *RAPPGlobVar* instances we created for the requests.

```
rt.updateTransaction(trans);
```

And then we print the results. First the information about all global variables with names containing ‘_HOST’.

```
RAPPGlobVars vars=gv1.getVariables();
int cc=vars.getCount();
System.out.println("\nGLOBAL VARIABLES *_HOST*, count = "+cc);
for ( int ii=0; ii<cc; ++ii )
{
    RAPPGlobVar v=vars.getVariable(ii);
    System.out.println(" "+v.getName()+" = \""+v.getValue()+"\"");
}
```

Then we print the information about all the variables.

```
vars=gv2.getVariables();
cc=vars.getCount();
System.out.println("\nALL GLOBAL VARIABLES #1, count = "+cc);
for ( int ii=0; ii<cc; ++ii )
{
    RAPPGlobVar v=vars.getVariable(ii);
    System.out.println(" "+v.getName()+" = \""+v.getValue()+"\"");
}
```

And finally we print out the information about all the variables with names containing ‘_SERVER_’.

```

vars=gv3.getVariables();
cc=vars.getCount();
System.out.println("\nALL GLOBAL VARIABLES #2, count = "+cc);
for ( int ii=0; ii<cc; ++ii )
{
    RAPPGlobVar v=vars.getVariable(ii);
    System.out.println("  "+v.getName()+" = \""+v.getValue()+"\"");
}

```

How to Call an Object, Disconnect and then Reconnect for the Reply

In this section we will look at how you can call an object and then disconnect from the NetPhantom server. After that we will reconnect to fetch the reply for the call object. Of course, this disconnecting and reconnecting is not something you normally do when calling an object. It is just done here to show that it is possible to do it, and still be able to fetch the result after reconnecting.

To call a NetPhantom object, we create a new instance of a *RAPPCallObject*. The constructor takes four parameters, where the first parameter is the name of the NetPhantom object. The second parameter is the ID parameter (the first parameter) being passed to the object, the parameter often referred to as the “id – argId”. The third parameter is the action value, often referred to as the “command – argCmd”. The last parameter

```

RAPPTxaction trans=new RAPPTxaction();
RAPPCallObject co=new RAPPCallObject("SIGNON",
                                     "argIDNotUsedHere",
                                     RAPPCallObject.CMD,
                                     "argStrNotUsedHere");

trans.addAction(co);

```

Then we convert the *RAPPTxaction* into a *RemoteTransaction*, and request the transaction.

```

RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);

```

The NetPhantom Server will now execute the REXX application named SIGNON, which looks like this:

```

/* This object is called either by pressing "Logon to Customer List"*/
/* button or by the Remote Application */

Parse Arg argId, argMsg, argStr

say "#SIGNON arguments: argId="argId", argMsg="argMsg", argStr="argStr

/* Enter logon info from RemoteApplication */
ui=GlobVarGet('USERID')
pw=GlobVarGet('PASSWORD')
say "#SIGNON userid='"ui"' password '"pw'"

/* Note: the password is not displayed on terminal screen normally */
rc=HostSetFld('','USER',ui)
rc=HostSetFld('','PASSWORD',pw)

/* Move to main screen */
rc=HostSend('@E')
rc=HostWait(5)

/* Move to customer initial screen */
rc=HostSetFld('','CMD','2')
rc=HostSend('@E')
rc=HostWait(5)

/* Skip search customer */
rc=HostSend('@E')
rc=HostWait(5)

if HostGetScreen()="CUSTLIST" then
    return "OK, on screen CUSTLIST"

```

```
return "Error, wrong screen!"
```

We will now save the session id, the server address and the port number, before disconnecting. Server address and port number would only be needed if backup servers are used.

```
String sessID=ra.getSessionID();
String server=ra.getServerHostAddress();
int port =ra.getServerPort();
ra.close();
```

In this sample we will immediately reconnect using the same application name as with the first connection and the saved session id.

```
ra=testAPI.createRemoteApplication(APP_NAME, sessID);
ra.setServer(server, port);
ra.connect();
```

We will now get the reply for the request we sent before we disconnected.

```
rt=ra.getReply(requestID, true);
```

And then update the *RAPPCallObject* instance.

```
rt.updateTransaction(trans);
```

Since we called the SIGNON object, we should now be on the CUSTLIST host screen. We will make sure that this is so, by getting the current screen without screen data.

```
trans=new RAPPTransaction();
RAPPCallHost ch=new RAPPCallHost(false);
trans.addAction(ch);
```

Send the transaction, and get the reply.

```
rt=new RemoteTransaction(trans);
requestID=ra.requestTransaction(rt);
rt=ra.getReply(requestID, true);
rt.updateTransaction(trans);
```

And print the current screen name.

```
RAPPHostScreen scr=ch.getScreen();
System.out.println("\nCURRENT SCREEN = "+scr.getScreenName());
```

Accessing Listboxes

We will now fill the listbox on the CUSTLIST panel by calling the GETCUSTS object, and then retrieve the list of customers with the help of a *RAPPCallPanel* object.

We start by calling the object GETCUSTS that will fill the list with customers.

```
RAPPTransaction trans=new RAPPTransaction();
RAPPCallObject co=new RAPPCallObject("GETCUSTS", "",
                                     RAPPCallObject.CMD, "");
trans.addAction(co);
```

The REXX code that will be processed by the NetPhantom server is named GETCUSTS, and looks like this:

```
/* Gets all customers from host into the listbox */
/* This object can handle multiple screens of data */

Parse Arg argId, argMsg, argStr

/* Clear the list of any previous data */
rc=PanListClear('CUSTLIST')

/* Get all pages and keep track of how many screens are */
```

```

/* "paged" using PF8 */
pageCount=0
done=0
do while done=0
/* Fill the list from the host */
listEnd=0
do line=1 to 19
custno=HostGetFld('', 'CUSTNO', line)
if custno='' then
do
listEnd=1
leave
end
name =HostGetFld('', 'NAME' ,line)
lastpur=HostGetFld('', 'LASTPUR', line)
amount =HostGetFld('', 'AMOUNT' ,line)
rc=PanListInsertData('CUSTLIST',0,custno || '09'X || name || '09'X
|| lastpur || '09'X || amount);
end

/* Check for end of list, make sure there's a MORE text below list
*/
if listEnd=1 | HostGetFld('', 'MORE')<>'MORE' then
done=1
else
do
/* Get next page */
rc=HostSend('@8')
rc=HostWait(5)
pageCount=pageCount+1
end
end

/* Page back using F7 */
do pageCount
rc=HostSend('@7')
rc=HostWait(5)
end

/* Return zero = OK */
return 0

```

After the list has been filled, we can get the list data with the help of a *RAPPCallPanel* object.

```

RAPPCallPanel cp=new RAPPCallPanel("CUSTLIST");
cp.request_getListData();
trans.addAction(cp);

```

Send the request and get the reply.

```

RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);
rt=ra.getReply(requestID,true);
rt.updateTransaction(trans);

```

We can now print the list data.

```

RAPPLines lines=cp.get_getListData();
if ( lines==null )
{
System.out.println("\nCUSTOMER LIST: no data!");
}
else
{
int cc=lines.getCount();
System.out.println("\nCUSTOMER LIST: lineCount = "+cc);
for ( int ii=0; ii<cc; ++ii )
{
String [] items=lines.getLineItems(ii);
String s=" ";
for ( int jj=0; jj<items.length; ++jj )
{
if ( jj>0 )

```

```
        s+=" ";
        s+="\""+items[jj]+"\"";
    }
    System.out.println(s);
}
```

How to Read Data from a Panel

Here we will see how to read data from entry fields on a panel. We will start by calling the CUSTDATA object that selects the first customer in the list and displays the customer info.

```
RAPPTransaction trans=new RAPPTransaction();
RAPPCallObject co=new RAPPCallObject("CUSTDATA","",
                                     RAPPCallObject.CMD,"");
trans.addAction(co);
```

We can now retrieve the data from the controls. Of course we could retrieve this data directly from the host screen rather than from the panel, but we want to demonstrate how you can access the panel controls. In some applications the data in a panel is collected from multiple screens, thus it can be fetched in a single step from the panel, while getting the data from the host screens requires several steps.

```
RAPPCallPanel name    =new RAPPCallPanel("NAME"    );
RAPPCallPanel addr1   =new RAPPCallPanel("ADDR1"   );
RAPPCallPanel addr2   =new RAPPCallPanel("ADDR2"   );
RAPPCallPanel lastpur =new RAPPCallPanel("LASTPUR" );
RAPPCallPanel amount  =new RAPPCallPanel("AMOUNT"  );
RAPPCallPanel firstpur=new RAPPCallPanel("FIRSTPUR");
RAPPCallPanel credlim =new RAPPCallPanel("CREDLIM" );
RAPPCallPanel credused=new RAPPCallPanel("CREDUSED");
RAPPCallPanel credrest=new RAPPCallPanel("CREDREST");
RAPPCallPanel credrate=new RAPPCallPanel("CREDRATE");
RAPPCallPanel goldmbr =new RAPPCallPanel("GOLDMBR" );
RAPPCallPanel selladdr=new RAPPCallPanel("SELLADDR");

name    .request_getText();
addr1   .request_getText();
addr2   .request_getText();
lastpur .request_getText();
amount  .request_getText();
firstpur.request_getText();
credlim .request_getText();
credused.request_getText();
credrest.request_getText();
credrate.request_getText();
goldmbr .request_getText();
selladdr.request_getText();
```

Add the getText actions to the transaction.

```
trans.addAction(name    );
trans.addAction(addr1   );
trans.addAction(addr2   );
trans.addAction(lastpur );
trans.addAction(amount  );
trans.addAction(firstpur);
trans.addAction(credlim );
trans.addAction(credused);
trans.addAction(credrest);
trans.addAction(credrate);
trans.addAction(goldmbr );
trans.addAction(selladdr);
```

Send the request and get the reply.

```
RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);
rt=ra.getReply(requestID,true);
rt.updateTransaction(trans);
```

Print the variables.

```

System.out.println("CUSTOMER DATA: "
    + "\n NAME = \""+name .get_getText()+"\" "
    + "\n ADDR1 = \""+addr1 .get_getText()+"\" "
    + "\n ADDR2 = \""+addr2 .get_getText()+"\" "
    + "\n LASTPUR = \""+lastpur .get_getText()+"\" "
    + "\n AMOUNT = \""+amount .get_getText()+"\" "
    + "\n FIRSTPUR = \""+firstpur.get_getText()+"\" "
    + "\n CREDLIM = \""+credlim .get_getText()+"\" "
    + "\n CREDUSED = \""+credused.get_getText()+"\" "
    + "\n CREDREST = \""+credrest.get_getText()+"\" "
    + "\n CREDRATE = \""+credrate.get_getText()+"\" "
    + "\n GOLDMBR = \""+goldmbr .get_getText()+"\" "
    + "\n SELLADDR = \""+selladdr.get_getText()+"\"");

```

Manipulating Panel Controls

Here we will look at how you can manipulate different controls on a NetPhantom Panel using *RAPPCallPanel*. We will use all the controls placed on the Customer Data panel.

```
RAPPTransaction trans=new RAPPTransaction();
```

Entry Fields

To manipulate an entry field, you use `request_setText` and `request_getText` on a *RAPPCallPanel* object.

```

RAPPCallPanel setText=new RAPPCallPanel("OUTPUT");
setText.request_setText("Sets the text for a control");

RAPPCallPanel getText=new RAPPCallPanel("ADDR1");
getText.request_getText();

```

Radio Buttons

To manipulate radio buttons, you use `request_setEnabled` and `request_isEnabled` on a *RAPPCallPanel* object.

```

RAPPCallPanel setEnabled=new RAPPCallPanel("RADIO3");
setEnabled.request_setEnabled(false);

RAPPCallPanel isEnabled=new RAPPCallPanel("RADIO2");
isEnabled.request_isEnabled();

```

Checkboxes

To manipulate checkboxes, you use `request_setChecked` and `request_isChecked` on a *RAPPCallPanel* object.

```

RAPPCallPanel setChecked=new RAPPCallPanel("RADIO2");
setChecked.request_setChecked(1);

RAPPCallPanel isChecked=new RAPPCallPanel("RADIO2");
isChecked.request_isChecked();

```

Listboxes

To get the line count from a listbox, you do a `request_getLineCount` on a *RAPPCallPanel* object.

```

RAPPCallPanel getLineCount=new RAPPCallPanel("OUTLIST");
getLineCount.request_getLineCount();

```

To insert a new line in a listbox, you do `request_insertLine` with *RAPPLine* as a parameter on a *RAPPCallPanel* object. The *RAPPLine* object can be created from a String array, where each element is the data for a cell.

```

RAPPCallPanel insertLine=new RAPPCallPanel("OUTLIST");
String [] line=new String []
{
    "Line 0",
    "### Line inserted at beginning of list ###"
};

```

```
insertLine.request_insertLine(new RAPPLine(line),0);
```

To update an existing line in a listbox, you do a `request_setLine` with `RAPPLine` as a parameter on a `RAPPCallPanel` object. The `RAPPLine` object can be created from a `String` array, where each element is the data for a cell.

```
RAPPCallPanel setLine=new RAPPCallPanel("OUTLIST");
line=new String []
{
    "Used to be line 3, is now line 4",
    "The file \"outlist.txt\" is located...\"
};
setLine.request_setLine(new RAPPLine(line),3);
```

To get the data from a line in a listbox, you do a `request_getLine` with the line number as a parameter on a `RAPPCallPanel` object.

```
RAPPCallPanel getLine=new RAPPCallPanel("OUTLIST");
getLine.request_getLine(1);
```

To delete a line in a listbox, you do a `request_deleteLine` with the line number as a parameter on a `RAPPCallPanel` object.

```
RAPPCallPanel deleteLine=new RAPPCallPanel("OUTLIST");
deleteLine.request_deleteLine(4);
```

To set the contents of a single cell, you do a `request_setCell`, with the column number, the line number and the data as parameters, on a `RAPPCallPanel` object.

```
RAPPCallPanel setCell=new RAPPCallPanel("OUTLIST");
setCell.request_setCell(0,1,"Cell at current Line 1");
```

To get the contents of a single cell, you do a `request_getCell`, with the column number and the line number as parameters, on a `RAPPCallPanel` object.

```
RAPPCallPanel getCell=new RAPPCallPanel("OUTLIST");
getCell.request_getCell(1,0);
```

To set the selection in a listbox, you do a `request_setSelection`, with the line number and state for the selection as parameters, on a `RAPPCallPanel` object.

```
RAPPCallPanel setSelection=new RAPPCallPanel("INLIST");
setSelection.request_setSelection(2,true);
```

To get the next selection in a listbox, you do a `request_getNextSelection`, with the line number to start from as a parameter, on a `RAPPCallPanel` object.

```
RAPPCallPanel getNextSelection=new RAPPCallPanel("INLIST");
getNextSelection.request_getNextSelection(-1);
```

To delete the whole contents of a listbox, you do a `request_deleteAll` on a `RAPPCallPanel` object.

```
RAPPCallPanel deleteAll=new RAPPCallPanel("INLIST");
deleteAll.request_deleteAll();
```

To get all the data from a listbox, you do a `request_getListData` on a `RAPPCallPanel` object.

```
RAPPCallPanel getListData=new RAPPCallPanel("OUTLIST");
getListData.request_getListData();
```

To add several lines to a listbox, you do a `request_setListData` on `RAPPCallPanel` list object. The `request_setListData` takes a `RAPPLines` object as a parameter.

You add lines to a `RAPPLines` object by calling its `addLine` method that takes an array of `String` as parameter. Each `String` in the array is the content of a single cell on the line.

```

RAPPCallPanel setListData=new RAPPCallPanel("OUTLIST");
RAPPLines list=new RAPPLines();
list.addLine(new String [] { "row 1, col 1", "row 1, col 2" });
list.addLine(new String [] { "row 2, col 1", "row 2, col 2" });
list.addLine(new String [] { "row 3, col 1", "row 3, col 2" });
list.addLine(new String [] { "row 4, col 1", "row 4, col 2" });
list.addLine(new String [] { "row 5, col 1", "row 5, col 2" });
list.addLine(new String [] { "row 6, col 1", "row 6, col 2" });
list.addLine(new String [] { "row 7, col 1", "row 7, col 2" });
list.addLine(new String [] { "row 8, col 1", "row 8, col 2" });
list.addLine(new String [] { "row 9, col 1", "row 9, col 2" });
setListData.request_setListData(list);

RAPPCallPanel callControl=new RAPPCallPanel("END");
callControl.request_callControl();

```

Control Attributes

To manipulate the visibility of a control, you use the `request_setVisible` and the `request_isVisible` on a `RAPPCallPanel` object.

```

RAPPCallPanel setVisible=new RAPPCallPanel("HIDDEN");
setVisible.request_setVisible(false);

RAPPCallPanel isVisible=new RAPPCallPanel("VISIBLE");
isVisible.request_isVisible();

```

We are now ready to add all these action to the `RAPPTransaction`.

```

trans.addAction(setText);
trans.addAction(getText);
trans.addAction(setEnabled);
trans.addAction(isEnabled);
trans.addAction(setChecked);
trans.addAction(isChecked);
trans.addAction(getLineCount);
trans.addAction(insertLine);
trans.addAction(setLine);
trans.addAction(getLine);
trans.addAction(deleteLine);
trans.addAction(setCell);
trans.addAction(getCell);
trans.addAction(setSelection);
trans.addAction(getNextSelection);
trans.addAction(deleteAll);
trans.addAction(getListData);
trans.addAction(setListData);
trans.addAction(callControl);
trans.addAction(setVisible);
trans.addAction(isVisible);

```

And then we send the request, and get the reply.

```

RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);
rt=ra.getReply(requestID,true);
rt.updateTransaction(trans);

```

Finally we print the result.

```

System.out.println("CALL PANEL FUNCTIONS: "
+" \n  getText          = \" "+getText          .get_getText()+" \" "
+" \n  set_text         = "+set_text         .get_setText()
+" \n  setEnabled        = "+setEnabled        .get_setEnabled()
+" \n  isEnabled         = "+isEnabled         .get_isEnabled()
+" \n  setVisible        = "+setVisible        .get_setVisible()
+" \n  isVisible         = "+isVisible         .get_isVisible()
+" \n  setChecked        = "+setChecked        .get_setChecked()
+" \n  isChecked         = "+isChecked         .get_isChecked()
+" \n  getLineCount       = "+getLineCount       .get_getLineCount()
+" \n  insertLine         = "+insertLine         .get_insertLine()
+" \n  setLine           = "+setLine           .get_setLine()
+" \n  getLine           = \" "+getLine           .get_getLine()+" \" "
+" \n  deleteLine        = "+deleteLine        .get_deleteLine()

```

```
+ "\n setCell          = "+setCell          .get_setCell()
+ "\n getCell         = \"+getCell         .get_getCell()+\"\"
+ "\n setSelection    = "+setSelection    .get_setSelection()
+ "\n getNextSelection = "+getNextSelection.get_getNextSelection()
+ "\n deleteAll       = "+deleteAll       .get_deleteAll()
+ "\n setListData     = "+setListData     .get_setListData()
+ "\n callControl     = "+callControl     .get_callControl();
```

```
RAPPLines lines=getListData.get_getListData();
int cc=lines.getCount();
System.out.println(" list data, lineCount = "+cc+");
for ( int ii=0; ii<cc; ++ii )
    System.out.println(" line "+ii+ " = "+lines.getLine(ii));
```

Logging off from the Host

To logoff from the host, we use the REXX object called LOGOFF. We call this object in the same way that we have called REXX objects before, using a *RAPPCallObject* object.

```
RAPPTransaction trans=new RAPPTransaction();
RAPPCallObject co=new RAPPCallObject("LOGOFF","",
                                     RAPPCallObject.CMD,"");
trans.addAction(co);
```

Then we send the request and get reply.

```
RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);
rt=ra.getReply(requestID,true);
rt.updateTransaction(trans);
```

The REXX code for the LOGOFF object looks like this:

```
/* Logs off from the host by pressing PF2 until the MAINMENU screen*/
/* and then sends 'X' (Exit) action */

Parse Arg argId, argMsg, argStr

/* Make sure the host is not busy due to CallControl(END) */
/* that sends PF2 from the CUSTDATA panel and moves the */
/* host to the CUSTLIST screen */
rc=HostWait(5)

/* Press PF2 until MAINMENU */
do while HostGetScreen(<>'MAINMENU'
    rc=HostSend('@2')
    rc=HostWait(5)
end

/* Do the Exit action */
rc=HostSetFld('','CMD','X')
rc=HostSend('@E')
rc=HostWait(5)

return 0
```

And finally we print the result.

```
System.out.println("\nLOGOFF returned \"+co.getReturnString()+\"");
```

Calling a User Class

We will now look at how you can call a user class with user-defined data in an *Element*. The reply will be returned from the called class as an *Element*. It is then up to the caller to extract the information as required.

Element is a class in the `org.w3c.dom` package. For more details see the JavaDoc for that package.

In order for a user class to be callable from RAPP, it must implement the *CallClassInterface*.

We start with creating a transaction and our user-defined Element.

```
RAPPTransaction trans=new RAPPTransaction();
Element requestData=trans.createElement("myRequest");
```

The Element can then be used to create a RAPPElement. To this RAPPElement you then add attributes, which consist of name and data. As seen in the sample code below, the data can be of different types.

```
RAPPElement e=new RAPPElement(requestData,null);
e.setAttribute("myStringAttribute","myValue");
e.setAttribute("myIntegerAttribute",123);
e.setAttribute("myBooleanAttribute",true);
```

It is also possible to add a complete child-element with its own attributes, to the RAPPElement.

```
RAPPElement c=e.createAppendedChild("myChildElement");
c.setAttribute("myChildAttribute","childValue");
```

To send the request to the user class in the server, you use a RAPPClass, which takes the name of the user class as the first parameter and the Element as the second parameter.

```
RAPPClass cc=new RAPPClass("rappapi.server.CallClassTest",
    requestData);
```

The server side class, called CallClassTest, looks like this:

```
package rappapi.server;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import se.entra.phantom.rapp.RAPPElement;
import se.entra.phantom.rapp.RAPPTransaction;
import se.entra.phantom.rapp.RAPPClass;
import se.entra.phantom.rapp.RemoteApplication;
import se.entra.phantom.rapp.RemoteTransaction;
import se.entra.phantom.server.rapp.CallClassInterface;
import se.entra.phantom.server.rapp.DefaultRemoteApplication;
import se.entra.phantom.server.rapp.RequestThread;

/**
 * This class is called by the RAPP API in the NetPhantom Server
 * from the TestAPI application.
 *
 * <p>Each time this CallClass action is about to start, a new
 * instance of the class (that implements this interface) is
 * created and called with the method <code>runAction</code>.
 *
 * @see se.entra.phantom.rapp.RAPPClass
 * @see se.entra.phantom.server.rapp.DefaultRemoteApplication
 * @see se.entra.phantom.server.rapp.RequestThread
 */
public class CallClassTest implements CallClassInterface
{
    /**
     * Runs the CallClass action.
     *
     * <p>The return value is a success flag. A failure indicates
     * that the script must be halted after this action.
     *
     * <p>The action has to set its own error code in the class
     * and eventually also log the error/warning in the server log.
     *
     * @returns true for success, false for failure.
     *
     * @see se.entra.phantom.server.rapp.RequestThread#setReply
     * @see se.entra.phantom.server.rapp.RequestThread#setError
     */
}
```

```
public boolean runAction(DefaultRemoteApplication app,
                        RequestThread requestThread,
                        RAPPCallClass action)
{
    // Get the request data (it might be null) and print it out.
    Element request=action.getRequest();
    if ( request==null )
        System.out.println("#CallClassTest: No request data");
    else
        System.out.println("#CallClassTest: Request data = "+
                            RAPPElement.getElementString(request));

    // This code should normally do something meaningful, but
    // as this is just a sample, create a "static" reply now.
    // This reply is stored in an Element without a namespace.
    Document document=requestThread.getDocument();
    Element replyElement=document.createElement("myReplyData");
    RAPPElement reply=new RAPPElement(replyElement,null);
    reply.setAttribute("myReplyStringAttribute" ,"StringValue");
    reply.setAttribute("myReplyIntegerAttribute",456);
    reply.setAttribute("myReplyBooleanAttribute",true);

    // Create "child 1" element.
    RAPPElement child1=reply.createAppendedChild("myReplyChild1");
    child1.setAttribute("myReplyChild1AttrA","attr A, child #1");
    child1.setAttribute("myReplyChild1AttrB","attr B, child #1");

    // Create "child 2" element.
    RAPPElement child2=reply.createAppendedChild("myReplyChild2");
    child2.setAttribute("myReplyChild2AttrA","attr A, child #2");
    child2.setAttribute("myReplyChild2AttrB","attr B, child #2");

    // Add a nested child to "child 1" element.
    RAPPElement
    nestChild1=child1.createAppendedChild("myNestedReplyChild1");
    nestChild1.setAttribute("myNestedReplyChild1AttrA",
                            "attr A, nested child #2");
    nestChild1.setAttribute("myNestedReplyChild1AttrB",
                            "attr B, nested child #2");
    Element ncl=nestChild1.getElement();
    ncl.appendChild(document.createComment(" THIS IS THE NESTED CHILD
1 USER DATA TEXT DATA THAT FOLLOWS "));
    ncl.appendChild(document.createTextNode("This is an XML text
node\nthat can contain <any> data, but beware of whitespaces!"));

    // Set the reply for the action.
    action.setReplyData(replyElement);

    // Return OK to continue processing.
    return true;
}
}
```

The server handles errors such as calls to classes that do not exist, and calls to classes that do not implement the `CallClassInterface`. The sample demonstrates these two error cases by making such calls.

```
RAPPCallClass cc2=new RAPPClass("undefined.class.Nowhere");
trans.addAction(cc2);

RAPPClass cc3=
    new RAPPClass("se.entra.phantom.server.HostField");
trans.addAction(cc3);
```

Send the request and get the reply.

```
RemoteTransaction rt=new RemoteTransaction(trans);
int requestID=ra.requestTransaction(rt);
rt=ra.getReply(requestID,true);
rt.updateTransaction(trans);
```

Finally, we print all data from the reply (note that the reply might be null, depending on the server *CallClass* implementation). The string for the *Element* is retrieved from the *RAPPElement* class that also has static helper methods.

```
Element replyData=cc.getReply();
if ( replyData==null )
{
    System.out.println("\nCALL CLASS rappapi.server.CallClassTest: no
reply data"
                        +"\n errorCode   = "+cc.getErrorCode()
                        +"\n description = "+cc.getErrorDescription());
}
else
{
    String s=RAPPElement.getElementString(replyData);
    System.out.println("\nCALL CLASS rappapi.server.CallClassTest: reply
data:\n"
                        +" "+s);
}
```

And we print information about the two *CallClass* actions that fail.

```
System.out.println("\nCALL CLASS undefined.class.Nowhere:"
                  +"\n errorCode   = "+cc2.getErrorCode()
                  +"\n description = "+cc2.getErrorDescription());

System.out.println("\nCALL CLASS se.entra.phantom.server.HostField:"
                  +"\n errorCode   = "+cc3.getErrorCode()
                  +"\n description = "+cc3.getErrorDescription());
```

Event Logging

It is also possible to log events in the NetPhantom Server log with the help of the *RemoteApplication*. The following code demonstrates how this might be done.

```
ra.logServerEvent(RemoteApplication.EVENT_INFORMATIONAL,
                  "About to call rappapi.server.CallClassTest
action");
```

In a similar way it is possible to log debug information.

```
ra.logDebugOutput("Reply from rappapi.server.CallClassTest
action: "+s);
```

Summary

In this chapter we have only looked at the code relevant to demonstrate the functionality of the RAPP API. To view the entire code used in the sample, select *General – RAPP API sample* in the *Menu* on the NetPhantom Server index page (in a default installation). To run the NetPhantom Application that is used by the Remote Application Sample, select *Browser Application – RAPP API Client* on the *Menu* in the NetPhantom Server index page (in a default installation).

To run the sample, see the instructions at the beginning of this chapter.

3 Setting up the NetPhantom Server

*If at first you don't succeed, try, try again.
Then quit. There is no use being a damned fool about it*

W. C. Fields

The NetPhantom Server has to be configured to be able to run a Remote Application. This has already been done for the RAPP sample in a default installation, but we will go through all the settings to make it clear how to setup a Remote Application.

All required configuration is done in the Server Administration program.

Configuring the NetPhantom Application

The NetPhantom Application that you want to make available as a Remote Application, must be configured in the same way as any other NetPhantom Application.

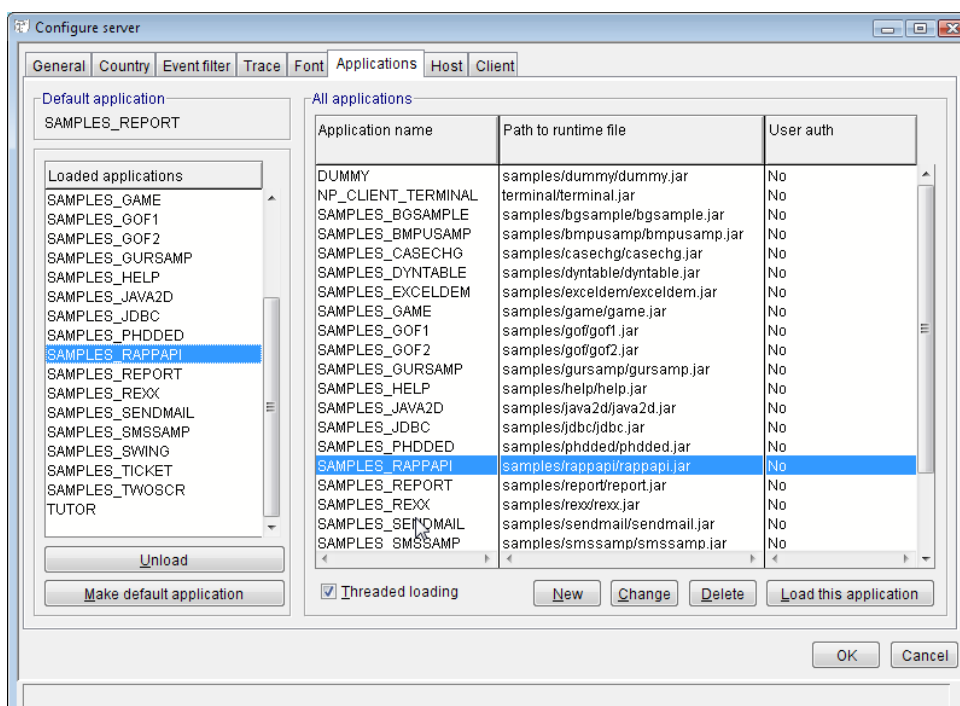


Fig. 15 Configuring the NetPhantom Application.

Setting up the Remote Application

To make a NetPhantom Application available as a Remote Application, you must define a Remote Application that points to the NetPhantom Application. You do this on the **Remote Apps** notebook page under **Configure Web server**.

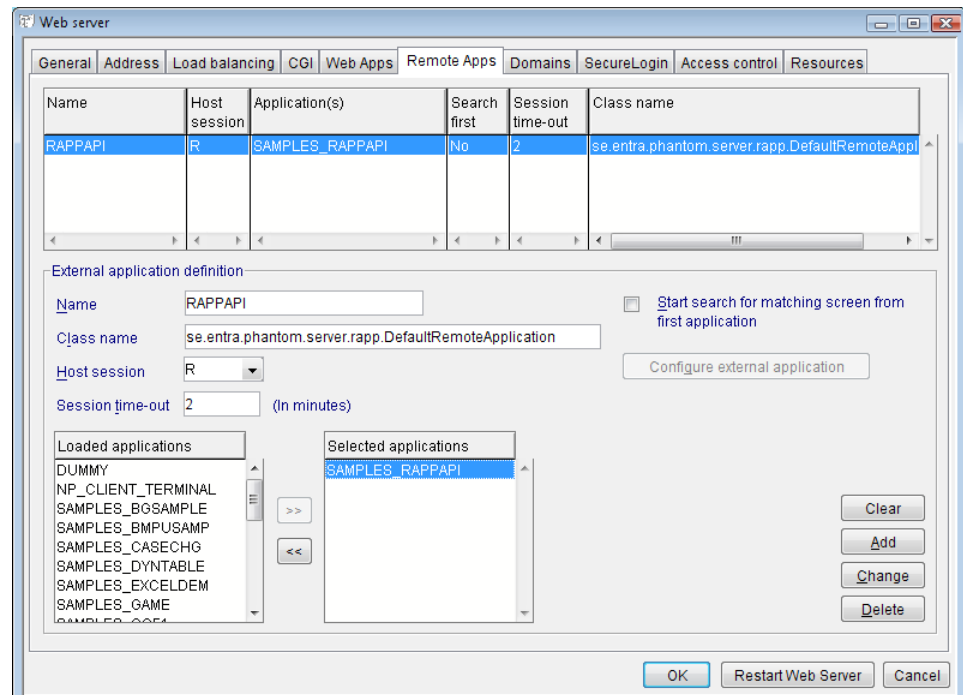


Fig. 16 Configuring the Remote Application.

The Remote Application must have a unique name. It can be the same as the name of the NetPhantom Application. In this sample, the name RAPPAPI has been chosen.

The class name is the name of the class that implements the server side RemoteApplication. In this case the default class *se.entra.phantom.server.rapp.DefaultRemoteApplication* will be used.

Choose an appropriate **Host session** for the Remote Application.

You can also specify a **Time-out** time in minutes for the Remote Application.

Defining the Resource for the Remote Application

In order to access this Remote Application, you must define a resource that points to it. You do this on the **Resources** notebook page under **Configure Web server**.

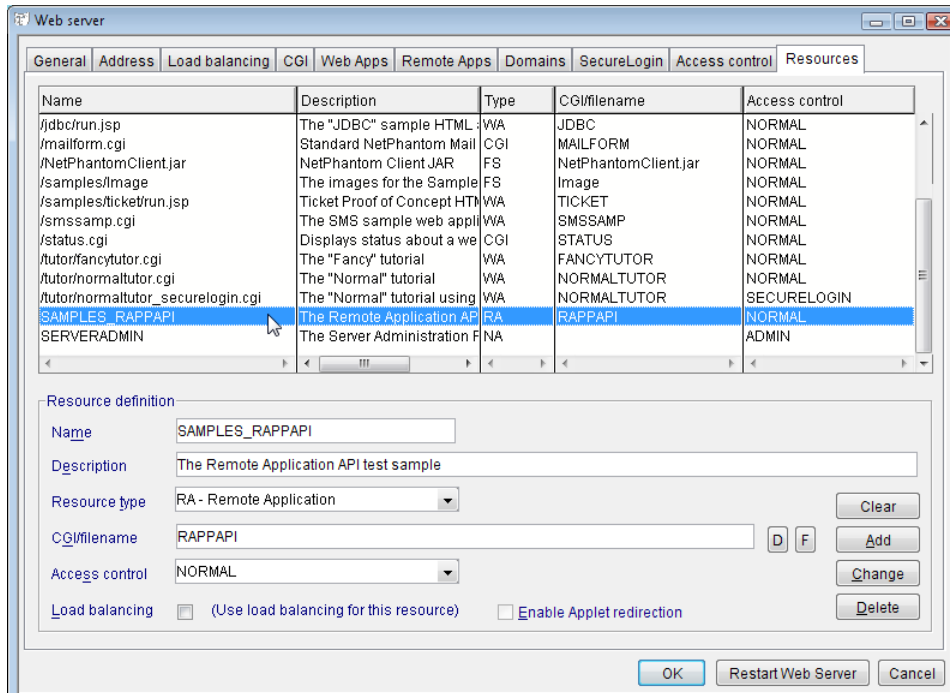


Fig. 17 Configuring the Resource for the Remote Application.

The **Resource** must have a unique **Name**. It can be the same as the name for the Remote Application. The resource name is the name that you use in the external application to specify which Remote Application you want to use. In this case the name is SAMPLES_RAPPAPI.

You can enter a description.

The **Resource type** must be set to RA - Remote Application.

The **CGI/Filename** is the name you defined for the Remote Application.

If you want, you can use **Access control** for this resource. It is not used for this sample, so it is left to NORMAL.

Check the **Load balancing** checkbox if load balancing should be used for this resource.