

Nextra Server Developer's Guide

Version 5
2nd Edition



Contents

Chapter 1 Introduction	3
Using this book	3
Format Conventions	4
Chapter 2 Technical Overview	6
RPC (Remote Procedure Call)	6
The Three-Tiered Architecture	8
Chapter 3 Server Tutorial.....	13
Before You Start the Tutorial	13
Your First Distributed Application.....	13
Write the Server Code	14
Create a Definition or IDL File	18
Generate Client Stub and Server Skeleton	20
Prepare Your Server Code for Compilation.....	21
Prepare Your Client Code for Compilation.....	23
Compile the Executable Server and Client.....	27
Edit the Auxiliary File.....	27
Start the Distributed Application	28
Try RPCDebug	30
Chapter 4 Understanding the Tutorial.....	33
Writing server code.....	33
Writing IDL file	33
Generating Skeletons	34
Creating auxiliary files.....	34
Starting applications	34
Summary.....	35
Chapter 5 The Development Process.....	36
The Nextera Development Paradigm	36
Development Process Outline	38
Chapter 6 Building Functionality Servers.....	44
Writing a IDL file	44
Writing Server Code	50
Generating a server skeleton	60
Compiling a Server	65
Creating an Environment file	69
Testing a Nextera Server	70
Testing RPCs	73

Modifying servers	75
Conclusion: implement the server	75
Chapter 7 DB access server	76
The DB access server	76
Before You Start	77
Environment variables for dedicated server	79
Runtime overview	79
Development Process	81
Chapter 8 Building DB access server	83
Building a DB access server	83
Using Special Features.....	93
Chapter 9 Building Java application server	101
Characteristics of Java application server	101
Architecture	101
Requirements.....	102
Development process	102
Reminder over implementation.....	105

Chapter 1 Introduction

This chapter covers how to use this boo, who should use this book, and some of the concepts that provide a foundation for using Nextra.

Using this book

Welcome to this book that gets you started in designing and building three-tiered based distributed applications. This book combines detailed discussions of important topics with step-by-step instructions you can use to build distributed applications. Take a minute to read these two pages – make sure you have the background that we assume you have, and make sure you are reading the right book.

Who should use it

This book designed for all developers and administrators who work with three-tiered based distributed applications. Whether you are responsible for building servers, scripting GUI clients, or configuring an application, this book provides a fundamental understanding of how to design and build open distributed applications using Nextra. All other volumes of the documentation set assume that you have read this book.

What you should already know

In genera, you should have a basics understanding of distributed computing. You should have read *Read Me First* before delving into this book.

When to use it

Through hands-on tutorials and plain-language discussions, this book provides you with the core skill-set you need to build and run open distributed applications. You should use this book as a springboard to begin building your knowledge base.

To help you understand complex subject matter in a short time, we have kept strictly to the basics in this book. Many of the software's more impressive features and options are documented not here, but in *Configuration Guide & Reference*. Be sure to read these volumes before you begin working on a production application.

Format Conventions

Text conventions

Understanding the conventions used in this manual will help you to learn how to use the utilities and to navigate the manual's structure.

Format	Explanation	Example
terminal	Designates operating system or third-party utilities, file names, or constant values for variables.	Kermit, telnet cust.def
<i>sub-text</i>	Designates text that represents many possible literal values; substitute your particular value here.	<i>server_c.pl</i> <i>-e environment_file</i>
bold	In body text, bold designates Nextra utilities. In examples, bold highlights parts of the code.	rpcperl AppMinder
[brackets]	Designate optional text unless a vertical bar appears inside the brackets; in this case, one of the choices is required.	[-d <i>def_file</i>] [NONE ERROR WARN DEBU G]

Paragraphs set off in the following manner are code examples:

```
#include <stdio.h>

main() {
    int i;
    printf("The number is %d",i);
}
```

Symbols

The following symbols are used throughout the documentation to help you navigate the text.



Warning Message

Indicates that you should pay special attention to the accompanying message. The message contains crucial information, without which you will not be able to continue properly.



Hint Message

Indicates that the accompanying text, while it is not crucial information, does supply you with helpful instructions, depending on your situations.



Optional Message

Indicates that the accompanying text is optional. The message may outline additional functionality or an alternate method, or detail a process step that many aid you in understanding a concept.



Debug Tip

Indicates that the accompanying text contains instructions on debugging the current step of your project. Debugging tips may be skipped if you use other successful debugging methods or if you choose not to debug (at your own risk).

Chapter 2 Technical Overview

This chapter provides you with a basic, high-level understanding of how three-tiered based distributed applications operate.

This chapter is designed for developers and administrators who want to learn how Nextra works.

RPC (Remote Procedure Call)

In an open distributed environment, programs running on different machines can communicate with each other through Remote Procedure Calls (RPCs). In an RPC, one program (the client) asks another program (the server) to run a procedure. The server accepts parameters from the client, executes the procedure, and returns parameters and/or a return value, just as through it were a local procedure within the client code.

With Nextra, building distributed applications is easy, because you do not have to worry about writing the network communications routines. Instead, you write a brief Interface Definition Language (IDL) file that defines the data types of each remote function and its parameters. The communications routines stub for client and skeleton for server are then generated automatically from this file. The stub and skeleton, in conjunction with the Nextra runtime library, handle all aspects of packaging data for transport (data “marshalling”) and carrying out the transport, making the RPC completely transparent. Since all the underlying complexities are addressed by the stub and skeleton, you can focus on coding the application functionality. You write remote calls exactly as though they were local calls.

Naming service

A naming service provides a method a method of locating server resources on a network. In the same way that you can call “Information” to get the phone number of a person you wish to reach, a client application can query the naming service to locate a server offering the interface that it needs.

In Nextra, naming service is provided by the Broker, a special kind of server that keeps track of all the other servers running in each application.

Whenever an application starts up, it informs the naming server of its host machine and the port number at which it is operating. The naming service stores this information and gives it out to clients that ask for it.

To extend the telephone analogy: most of the times you call people directly, but if you do not know their number, you might make use of the operator. In the same way, clients usually send RPCs directly to servers. But if a client does not know how to reach a server, it turns to the naming service for help.

Unlike most servers, the Broker does not use a stub, since it offers a well known interface built into the Nextra TCP runtime library.

The sequence of events

Figure 2.1 shows what happens each time an RPC is executed. When the client code calls a remote function, the client stub executes a function of the same name, which in turn calls a sequence of functions defined in the Nextra TCP runtime library.

First, the client stub searches an internal table for the network address of a server which can handle the function. If the table does not contain an appropriate server's address, then the client stub queries the naming service.

The naming service sends the server's address to the client stub, which, in turn, updates its internal table of server locations. The client then connects with the server, and sends to the server the name and arguments of the called function. The client stub waits until the server skeleton sends back a response.

When the server skeleton receives the function name and arguments, it calls the specified server function (now local) with the arguments it received across the wire, and then sends the results back to the client stub.

The client stub closes the connection with the server, and returns the results of the remote procedure call to the client code.

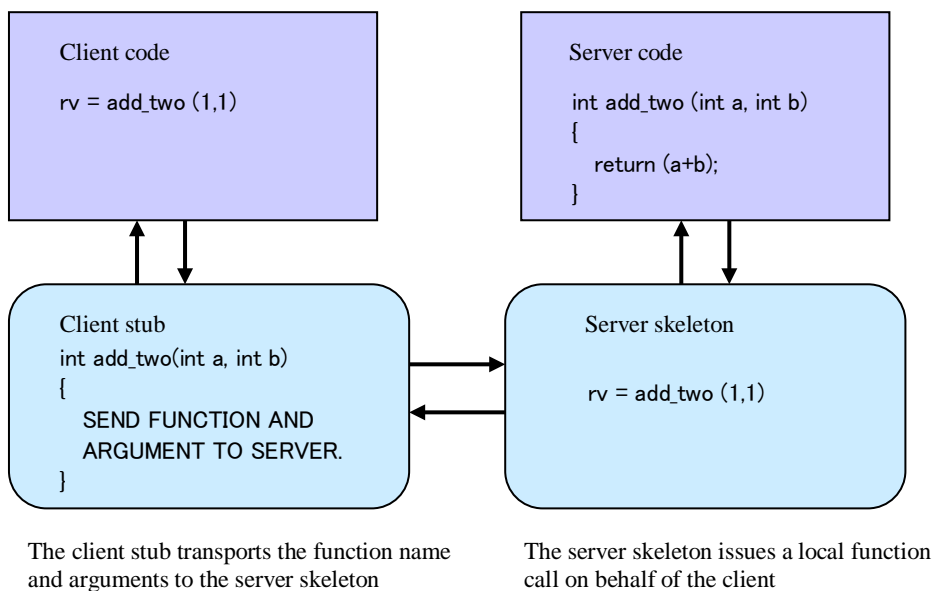


Figure 2.1: A remote function call

Why RPC?

RPC technology allows you to use server code as a shared resource – many different clients can access the same server. The clients and the server need not be written in the same language, nor must they run on the same OS or hardware platform.

The Three-Tiered Architecture

As an application programmer using Nextra, part of your job is to write code that supports an open, object-oriented paradigm. You not only become a better developer, but you also find yourself creating solutions you did not even know existed.

Developing in three tiers is a methodology that helps you to create open, modular, scalable, high performance, easy-to-modify, enterprise-wide applications. You can use a three-tiered approach in any fully-distributed environment.

What is it?

The three tiered architecture divides your application into three distinct segments: presentation tier, functionality tier and data tier. The user interface is the program through which the computer operator interacts with the application. In a three-tiered environment, the functions of the user interface are usually limited to presenting output and receiving input from the user, as well as data editing and verification. The application logic consists of the business methods through which your company carries out its functions, such as inventory, ordering, finance, etc. In a three-tiered environment, this tier contains practically all the data processing that the application performs. The data sources tier consists of RDBMSs, flat files, mainframes, or any other resource where corporate data is stored.

The three tiered architecture, as shown in Figure 2.2, depends on well-defined interfaces between your data sources and your application logic and one hands, and between your application logic and your user interfaces on the other. Because the interfaces are well-defined, both at the application level and the module level, changes are very easy to make, whether you are changing to a different graphical GUI, modifying application services, or moving to a new DB.

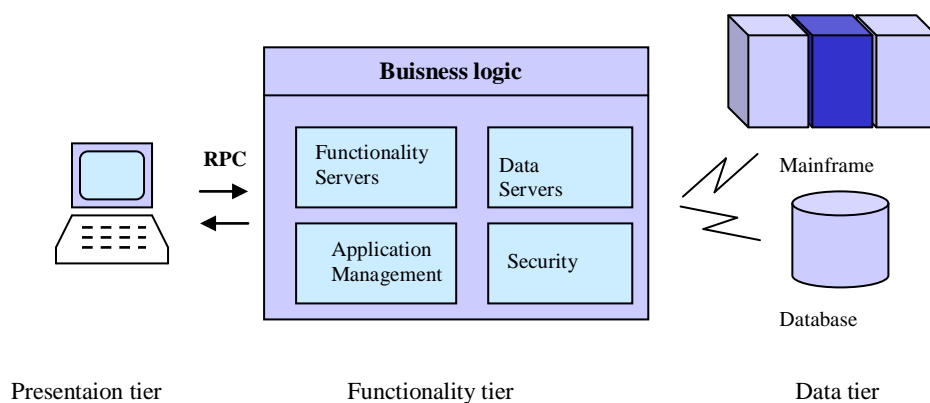


Figure 2.2: Three-tiered architecture

Keep in mind that the middle tier is a logical distinction which can take many forms. For example, an organization might have a set of database servers that provide raw data access for every application in the company. A different set of servers for each application then processes the information returned in a variety of ways, depending on the needs of the end users for that application. The application logic is not just a single server that grabs data and passes it to the user interface; it can be a set of services each capable of using other services' functions. A new application

can use any existing services that meet its needs. The three-tiered methodology encourages this kind of flexibility and reusability.

As figure 2.3 shows, the logic tier itself can be physically distributed as geography requires. Note that the diagram shows only one possible data path – nothing prohibits servers from other applications from making use of the Raw Data Access shown in the diagram.

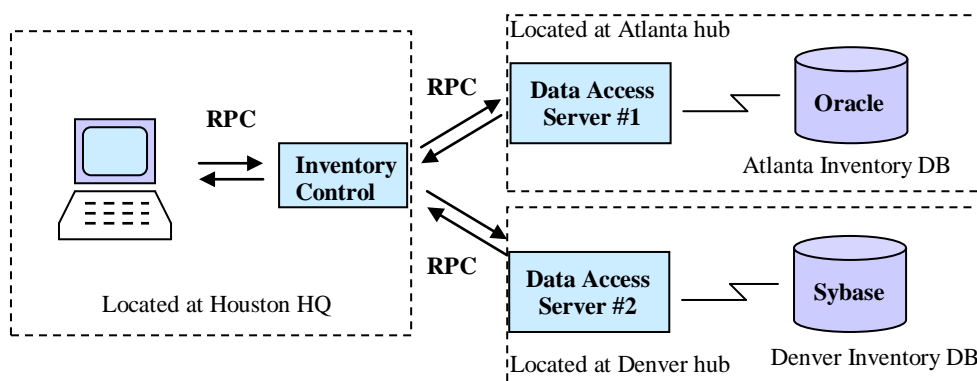


Figure 2.3: Example data path

Why three-tier?

The fundamental principles behind the three-tiered architecture are simple. An application should take advantage of the strengths – and avoid the weaknesses – of all the hardware and software resources that an organization owns. For example, while an inexpensive PC running a GUI such as VB makes a superb user interface, adding too much application logic behind the screens can cause the machine to suffer extreme performance degradation. In contrast, UNIX machines are often too expensive to use merely as user interfaces, but they provides tremendous processing power.

So why not use the Windows PC as a pure UI, and use the processing power of the UNIX machine to crunch the application data? And since your company probably has a large investment in existing mainframes or other databases, why not include software servers that can integrate these legacy data sources into your distributed environment?

Presentation tier

Get as much input as you can from the end –users when designing the user interface. They are the people who have to use it, so count on them

to let you know what they want. The earlier you include them in the design process, the fewer large changes you have to make later. Give them several chances to try it out. Expect lots of changes. Do not prototype in a vacuum.

When coding the front-end, include as little functionality as possible. Let the servers do the dirty work. Everything you code inside a GUI you have to re-code if a different GUI comes on line. However, your servers stay the same no matter how many different GUIs your department or organization uses.

Functionality tier

We designed the Nextra software to make your job easier and the quality of your work higher. You are not locked into proprietary products anymore. Learn how to use the middle tier – it is your playground. It is where your applications should do 90% of their work.

DB access server

DB access server is used to access RDBMSs and translate the stored data into an abstract format which can be easily manipulated by an application. This type of server is generated from a file of ANSI-SQL database queries that have been tagged with function names. You do not need to write a single line of C or SQL – or nay other programming language – to build a DB access server.

While you are not prevented from calling functions in a DB access server directly from a GUI client, you will realize the most significant benefits of the three-tiered architecture if you adhere to the methodology and build a tier of application servers between the GUI presentation clients and the raw data access server. To support such architecture, your DB access server must provide a comprehensive suite of methods for accessing your company's stored data.

Transaction server

With this kind of server, you can access multi-platform, multi-vendor databases within a single transaction. To build a transaction server, you must write one or more files of tagged ANSI-SQL statements, generate server code form the SQL files, write the code that groups the database functions into transactions, and write the IDL file.

The transaction server provides commit and abort capability across all corporate data stored in supported RDBMSs. This type of server should be used in situations where two or more database manipulations must be executed atomically.

Functionality server

Also known as business logic, functionality server provides the data processing methods (business rules) required by each application. To build a functionality server, you must write the data processing logic and the IDL file.

Functionality server should be used to turn the formatted data returned by the DB access server into information ready to be displayed on the client GUI.

When coding in an open distributed environment, modularity is the most powerful weapon in your arsenal. Servers are not sculptures – they are bricks, beams, windows and trimming, all of which come in standard sizes so they can be used again and again. Write servers to be as flexible as possible. When specifying a new application, see how many existing servers you can use.

You should think of your server tier as a set of libraries of methods for accessing data. Applications that use the same DB should be able to use the same data access servers, regardless of what the applications do. Partition application-specific processing into a different set of servers.

Data tier

Nextra affects this tier the least of the three. Remote systems (mainframes or other legacy platforms) and RDBMSs are included in the data tier – they are both sources of information.

When creating DBs, design them with traditional DB design tools and methodologies. Consider RDBMSs as a powerful method for accessing in the DB – let your servers do most of the work. Special care should be given in designing this tier as it is the most common location for slowed response time.

Chapter 3 Server Tutorial

This tutorial walks you through our example file, showing you how to build a small distributed application using Nextra. You can build a server that runs on TCP using the Nextra transport library. The object of this section is to provide you with an understanding of how distributed applications are constructed, and how easy building a distributed application can be using Nextra code-generation tools.

Before You Start the Tutorial

There are a few pieces of software you need before you embark on the tutorial:

- Required Nextra software packages: Developer Package
- Required Third Party Products: Compilers for each language

Your First Distributed Application

Before You Start

UNIX

In the UNIX environment, be sure to set the ODEDIR environment variable to the correct directory. For Nextra, ODEDIR must point to the tcp subdirectory of the Nextra installation directory. Then include \$ODEDIR at the front of PATH:

```
[sh or ksh]
```

```
ODEDIR=/usr/nextra/tcp
```

```
PATH=$ODEDIR/bin:$ODEDIR/../cmn/bin:$PATH
```

```
export ODEDIR PATH
```

Windows

In the Windows environment, check that the runtime library is installed in the PATH. Click [Control Panel] → [System], then add a new environment variable as: ODEDIR=*install_dir*\tcp. Include the

"%ODEDIR%\bin;%ODEDIR%\..\cmn\bin" to PATH environment variable.

Location of files

All the files you need are included on your UNIX or Windows distribution media in the directory \$ODEDIR/./samples (C:\ODEDIR\..\samples in Windows). Files in these directories are used and referred to by the rest of this tutorial.

Process overview

1. Write the program code. (In an actual development situation, this step is usually switched with the next one.)
2. Create an RPC definition file.
3. Generate client code (stub) and server code (skeleton) using RPCMake.
4. Compile the server and/or client.
5. Create or edit the environment file.
6. Run the client and the server.

Write the Server Code

The quickest way to get started writing this distributed application is to write it as a local one. Once the local program works, you can split it in two to make server and client pieces. When building production applications, you will probably not wish to work in this manner.

In this tutorial, the server consists of a short program containing two procedures: one to add two integers, and another to convert a string of lowercase characters to uppercase. The examples below explain two local programs, one in C and one in Perl, that prompt the user for input and then execute the addition and lower-to-upper functions.

C server

Start by writing a normal C program that prompts for integers to add and strings to convert to uppercase. Include procedures that execute the addition and the conversion. The example local C program is called "cprog.c"

```
cprog.c
```

```

#include <stdio.h>
#include <ctype.h>

/* ----- */
/* this is the code for the add function */
/* ----- */
int add(int x, int y)
{
    return(x+y);
}
/* ----- */
/* this is the code for the lower2upper function */
/* ----- */
int lower2upper(char *before, char **after)
{
    int i=0;
    char *a;
    *after = (char *)malloc(strlen(before) + 1);
    if (*after==NULL) {
        printf("Could not allocate memory!");
        exit(1);
    }
    a = *after;
    for (i=0; i<strlen(before);i++){
        a[i] = toupper(before[i]);
    }
    a[i] = '\0';
    return i;
}
main(int argc, char **argv)
{
    int one,two,result;
    char instring[100], *outstring;
    char inbuf[100];

    printf("\nPlease enter two numbers to add
(e.g. 3,6): ");
    gets(inbuf);
    sscanf(inbuf,"%d,%d", &one, &two);
    result = add(one,two);
    printf("\nThe result is %d\n", result);
    printf("\nPlease enter the string to
capitalize:\n");
    gets(instring);
    lower2upper(instring,&outstring);
    printf("\nThe string is <%s>\n", outstring);
    free(outstring);
}

```

```
}
```

You can compile the local program and run it. Be sure to use the ANSI option, if your compiler doesn't recognize ANSI code by default.

Table 3.1: Compiler commands

Operating System	C compiler command
HP-UX	cc -Aa
IBM AIX	cc
Windows	CL

The following is a template for the compiler command syntax in a UNIX environment. Note that Windows compilers have different flags and argument syntax from the UNIX compilers – and from each other.

```
command cprog.c -o testprog
```

After the program compilers run the executable by typing its name (testprog, in the UNIX example above.)

If the code you have written is bug-free, then the local application should prompt for input and print the results you expect.

Perl server

Start by writing a local Perl program that executes the specified functions. The example program is called perlprog.procs, and looks like this (in the UNIX environment-for Windows, the path names are different):

```
perlprog.procs

eval "exec `which_rpcperl` -S $0 $*"
  if 0;
$INC[$#INC+1]="$ENV{ODEDIR}/lib";# These lines tell
$INC[$#INC+1]="/usr/local/lib/perl"; # rpcperl where
to
$INC[$#INC+1]="/usr/lib/perl"; # find libraries

#####
# This code tests the procedure add
#####
```

```

print "Please enter the two numbers you wish to add
(e.g. 3,4) >";
$ans=;
chop($ans);
($op1,$op2)=split(",",$ans);
$retval = &add($op1,$op2);
print "\n\nThe result is <$retval>\n\n";

#####
# This code tests the remote procedure lower2upper
#####
print "Please enter the lower case sentence you wish
to convert to upper case:\n";
$ans=;
chop($ans);
&lower2upper($ans,*new);
print "\n\nThe new word is <$new>\n\n";

#
# add:
# returns the sum of two numbers.
#
sub add {
    local($op1,$op2)=@_;
    return($op1+$op2);
}

#
# lower2upper:
# converts the first parameter to upper case,
# returning the result in the second parameter.
#
sub lower2upper {
    local($in_string,*out_string)=@_;
    $out_string=$in_string;
    $out_string=~tr/a-z/A-Z/;
}

```

Now, if the program is not executable (in the UNIX environment), make it so by typing:

```
> chmod +x perlprog.procs
```

Once you have made the program executable, you can run it by typing:

```
> perlprog.procs
```

Enter two numbers to add, and then a string to convert to uppercase, as prompted. Verify that the results are what you expect.

Next step

You have just performed the first step of the development process - writing code free of local errors.

Next, you will take the first step involved in distributing the application. That step is defining the interface between server and client.

In the local code, this interface is built-in. You can think of the main program routine as the client, and the other routines as the server.

Create a Definition or IDL File

Why?

Since the client of a distributed application assumes the role that (main) routine plays in your local program, and the server assumes the role of the sub-routines, you need a method of ensuring that both pieces understand how to send function parameters back and forth. Nextra builds this knowledge directly into the client and server, using function definitions that have been written in a file.

This file is called an IDL file for Nextra/TCP and is used to describe the inputs and outputs of each procedure that the server will be capable of executing. These parameters provide enough information for Nextra tools to generate all the necessary marshalling and communications routines that will allow your client to request services from your server and exchange data.

How?

In the local program you saw in the first step (`cprog.c` or `perlprog.procs`), you can find all the information you need to create the definition file for the server. By inspecting the inputs and outputs of the procedures, you can determine what information the interface definition file should contain. Each procedure has a return value, and input and/or output parameters.

Definition files are independent of programming language or platform, so the same file can define both your Perl server and your C server.

Example

Here is a listing of the definition file `basics.def` That corresponds to the local program code from the previous step:

```
basics.def

# interface definition file for: basics
[uuid(uuid) version(1.0)]
interface basics {
    int add (
        [in] int x,
        [in] int y);
    int lower2upper (
        [in] char s1[],
        [out] char s2[]);
}
```

Quick explanation

The first line is a comment.

The second line identifies the “interface” (the IDL word for a server), by using a universal unique ID (UUID). Next servers use the server name to identify servers, not the UUID, so if you are building a server you may use any UUID in the proper format.

The third line braces describes all functions. The `add` function returns a value of `int` data type; the function takes two input parameters of the same type, and has no output parameters. The `lower2upper` function has one input and a one output parameter, and an integer return value. Note that you don't have to give the function parameters the same names that they have in your server code. You do, however, have to use the same function names that appear in your code, because the functions are identified by name.

Next step

The interface definition file will be used in the next step: generating the communication routines. The **RPCMake** utility reads the interface definition file to create all the low level communications code that this client and server pair needs to take to each other.

Generate Client Stub and Server Skeleton

Now you're ready to generate network communications routines - called *stub/skeleton* - based on the information you have specified in the interface definition file. The **RPCMake** utility provides this service. The stub/skeleton will take care of all the low-level communications that you'd rather not worry about.



Automatic stub/skeleton generation.

The Makefile included with the C example code automatically generate stub/skeleton when you issue the make command. If you are working in C, you do not need to generate stub/skeleton now, but you may. If you do generate stub/skeleton now, the command in the Makefile will simply overwrite the stub/skeleton when you compile. This is perfectly acceptable.

On the **RPCMake** command line, use the `-d` option to flag your definition file name, the `-c` option to flag each desired client stub language, and the `-s` option to flag the desired server skeleton language. See the specific examples below.

C sever

To generate server and client stubs in the C language under Nexra, enter

```
> rpcmake -d basics.def -c c -s c -c perl
```

After you hit <Return>, **RPCMake** generates a server skeleton named `basics_s.c`, a client stub named `basics_c.c`, a header file named `basics.h`.

A Perl client stub, `basics_c.pl` that you will use with the **RPCDebug** testing utility later.

Perl server

To generate server and client stubs in Perl, enter:

```
> rpcmake -d basics.def -p basics.procs -c perl -s perl
```

For Perl servers, you have to add the `-p` option flagging the name of the server code file because the Perl server skeleton includes a literal reference to your server code.

After you hit <Return> **RPCMake** generates a server skeleton named `basics_s.pl`, and a client stub named `basics_c.pl`.

RPCMake GUI

This interface utility allows you to click on buttons to generate stub/skeleton in various languages.

See *Reference* for complete information on **RPCMake**.

Prepare Your Server ode for Compilation

To split off server code from local code, make a copy of the local code and rename it, so that you can make changes without altering the original program.

C server

To turn your local C program into a network server, you need only make a few small changes to your local program code. Our example server code is called `basics.c`. Compare `basics.c` to `cprog.c` and you'll see that changes are minimal.

What to do

You need only:

- Replace each memory allocation function with the corresponding `dce_` allocation function.
- Delete `main()` routine.

You have now transformed the local program code into C code that can be compiled to become a server.

Example

The completed server source code, `basics.c`, looks like this:

basics.c

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "dceinc.h"

/* ----- */
/* this is the code for the add function */
/* ----- */
int    add(int x, int y)
{
    return(x+y);
}

/* ----- */
/* this is the code for the lower2upper function */
/* ----- */
int lower2upper(char *before, char **after)
{
    size_t i=0;
    char *a;

    *after = (char *)dce_malloc(strlen(before) + 1);
    a = *after;
    for (i=0; i<strlen(before); i++) {
        a[i] = toupper(before[i]);
    }
    a[i] = '\0';
    return i;
}

```

Perl server

To turn your local Perl program into an Nextra server, you need only delete the main routine of your program, leaving the server routines.

What to do

Delete the main routine. Add a single line at the end like this:

```
1;
```

This is required so that the server skeleton can successfully access the server code routines. Technically, the last line of the file must evaluate to a non-zero value for the server skeleton's `do` command to work.

Since Perl is an interpreted language, there is no need to compile.

Example

Compare `perlprogs.procs` to `basics.procs`, your complete examplePerl server. The complete Perl server code looks like this:

```
basics.procs
```

```
# add:
# returns the sum of two numbers.
#
sub add {
    local($op1,$op2)=@_;
    return($op1+$op2);
}

# lower2upper:
# converts the first parameter to upper case,
# returning the result in the second parameter.
#
sub lower2upper {
    local($in_string,*out_string)=@_;
    $out_string=$in_string;
    $out_string=~tr/a-z/A-Z/;
}

1; # this is needed for the "do" command in the server skeleton to work
```

Prepare Your Client Code for Compilation

To split off client code from local code, make another copy of the local code and rename it, so that you can make changes without altering the original program.

C client

The client requires only a standard start-up routine to be added to your source code. If you want, you can also include the header file generated by **RPCMake**. If you do so, you will be warned by the compiler if the

parameters of your client function calls do not match the parameters of your server code.

What to do

To complete your client source code, you must:

- Delete the server routines from your local program.
- Replace calls to `free()` with a call to `dce_release()`. In both cases, these are only necessary if you are receiving dynamic arrays from the server.
- Write the standard start-up routine.
- Add RPC error handling routines.

Example

Compare the completed client source code, `cclient.c` with the local C program. The client code looks like this, with the standard start-up routine in **bold text**:

```
cclient.c

#include <stdio.h>
#include "dceinc.h"
#include "basics.h"

main(int argc, char **argv)
{
    int one,two,result;
    char envfile[100];
    char instring[100], *outstring;
    char inbuf[100];

    printf("Please enter the name of the env file: ");
    gets(envfile);
    printf("File: %s", envfile);
    if (!dce_setenv(envfile,NULL,NULL)) {
        printf("Error: %s\n", dce_errstr());
        exit(1);
    }
    printf("\nPlease enter two numbers
                                                to add (e.g. 3,6): ");
    gets(inbuf);
    sscanf(inbuf,"%d,%d", &one, &two);
    result = add(one,two);
```

```

    if (dce_errnum()!=0) {
        printf("Error: %s\n", dce_errstr());
        exit(1);
    }
    printf("\nThe result is %d\n", result);
    printf("\nPlease enter the string
                                                to capitalize:\n");

    gets(instring);
    lower2upper(instring,&outstring);
    if (dce_errnum()!=0) {
        printf("Error: %s\n", dce_errstr());
        exit(1);
    }
    printf("\nThe string is <%s>\n", outstring);
    dce_release();
    return (0);
}

```

You are now completely finished writing code. Your server and client are ready to be compiled.

Perl client

The client requires a standard start-up routine to be added to your source code. The start-up routine initializes the client's data structures with runtime information such as the location of the RPC, the naming server that routes the client's remote procedure call to the proper server.

What to do

To complete your client source code, you must:

- Delete the server routines from your local program.
- Write the standard start-up routine.

Example

The start-up routine appears in **bold text** below. Your example Perl client, `perlclient`, looks like this (in the UNIX environment, that is—for Windows, the path names are different):

```

> perlclient

    eval "exec $ODEDIR/bin/rpcperl -S $0 $*"
        if 0;

```

```

unshift(INC, "$ENV{ODEDIR}/lib");

do "dce_func.pl" || die "<OEC ERROR> could not load
dce_func.pl\n";
do "basics_c.pl";#Client code must 'do' client stub

#####
# This code gets the env.
# file from the command line or asks for it.
#####
if (defined @ARGV) {
    $env_file="@ARGV";
}
else {
    print "Please enter the environment file: ";
    chop($env_file = <STDIN>);
}

if (!&dce_setenv($env_file, '', '')) {
    print "Could not set env using <$env_file>\n";
    exit(1);
}

#####
# This code tests the remote procedure add
#####
print "Please enter the two numbers you wish to add
(e.g. 3,4) >";
$ans=<STDIN>;
chop($ans);
($op1,$op2)=split(",",$ans);
$retval = &add($op1,$op2);
print "\n\n\nThe result is <$retval>\n\n";

#####
# This code tests the remote procedure lower2upper
#####
print "Please enter the lower case sentence you
        wish to convert to upper case:\n";
$ans=<STDIN>;
chop($ans);
&lower2upper($ans,*new);
print "\n\n\nThe new word is <$new>\n\n";

```

You can skip the next step for Perl clients or server, because they do not require compilation.

Compile the Executable Server and Client

Skip this step if you are working in Perl.

The commands to compile the C executables *in the UNIX environment* are in the makefile named `Makefile`, in the same directory as your example files.

What to do

To compile the C server, type `make server`.

To compile the C client, type `make client`.

To compile both parts, type `make all`.

The output files, named `basics` and `cclient`, are the executable program that marks up the distributed application.

On Windows, use the sample `makefile` to compile your applications.

Edit the Auxiliary File

Nextra client and server use a file called environment file in order to locate the Broker's location where it runs. For the Broker, the environment file is used to decide which host and port it uses in order to receive RPC request by the client and server, respectively. The Broker requires having the file (`broker.env` in this example) at start-up.

What to do

Look at the example environment file. Change the IP address specified by `DCE_BROKER` to point to the machine you want to run your Broker on (use your current machine, for now). If you don't know your machine's network name, you can find out by entering the command `hostname`. Use the name returned by this command to replace `193.1.1.60` in the sample environment file.

If you encounter errors later when trying to run the Broker, try a different port number. Any unused port between 2048~65535 should work. On a UNIX machine, you can find out if a port is in use by entering the following command.

```
> netstat -an | grep port_num
```

If a line including the port you specified prints to the screen, then the port is in use. If you receive no response, then the port is free.

Save your changes to the environment file, and then mark two more copies of it, called `server.env` and `client.env`. In each copy, specify a different file in the `DCE_LOG` variable. The log file specified in `broker.env` will be used by your broker, while the other log files will be used by the server and client, as specified.

Example

The environment file included with your example files looks like this:

```
DCE_BROKER=193.1.1.60,7800
DCE_DEBUGLEVEL=DEBUG,DEBUG
DCE_LOG=broker.log
```

These settings indicate that a high level of debugging information will be written to a file in the current directory called `broker.log`, and that the Broker will be listening at port number 7800 on machine 193.1.1.60.

For more information on environment files, see “Environment Files” in *Reference*.

Start the Distributed Application

For now, to verify that everything is working currently, make sure that your client and server executables—and their environment files—are in the same directory. From that directory, start up the Broker:

```
> broker -e broker.env &           (for UNIX)
> start /b broker -e broker.env (for Windows)
```

This command starts the Broker as a background process. The Broker reads the specified environment file and then starts up at the port specified in the file.

C server

If you have constructed a C server, start it using this command:

```
> basics -e server.env &           (for UNIX)
> start /b basics -e server.env (for Windows)
```

The server locates the Broker by reading the environment file, and then registers with the Broker as an available server.

C client

If you have constructed a C user interface, start it by typing this command:

```
> cclient
```

The command starts up the client, which prompts you for the environment file and then contacts the Broker for the names and locations of servers capable of fulfilling its requests. It then prompts you for input. Follow the instructions to see your distributed application in action.

Feel free to use your C client to access your Perl server, or use your Perl client to access your C server. This kind of interoperability is what our open distributed environment is all about.

Perl server

For a Perl server built on UNIX, use the following command to start it up:

```
> basics_s.pl -e server.env
```

The server locates the broker by reading the environment file, and then registers with the Broker as an available server.

For a Perl server built on Windows, use this command to start the Perl server.

```
> start rpcperl basics_s.pl -e server.env
```

Perl client

If you built a Perl client, start it using this command:

```
> perlclient
```

This command starts up the client, which prompts you for the environment file and then contacts the Broker for the names and locations of servers capable of fulfilling its requests. It then prompts you for input.

Feel free to use your C client to access your Perl server, or use your Perl client to access your C server. This kind of interoperability is what our open distributed environment is all about.

Next step

Go ahead and enter a few different numbers to add and lowercase strings to convert, and inspect the results. Even though all parts of the program are running on the same machine, you are making remote procedure calls and running a distributed application.

Try RPCDebug

Instead of starting a user interface, start up the **RPCDebug** utility (universal client) to test your new server:

```
> rpcdebug
```

On Windows, you can double click on the **RPC Developer** icon in the Program Manager.

This utility lets you call each function that the server offers, with dynamic input values. You can use it to test whether a server's functions are working properly.

1. When the window comes up, click on the Environment File button to load the environment file.
2. Click on the Definition File button to load the definition (IDL) file corresponding to your server(s). Select `basics.def` from the choices in the next panel.

3. Select a function to test from the box in the upper left corner of the main window. Select `add`, or `lower2upper` by clicking on the function name.
4. Enter input values for the selected function.
5. Click the “Execute” button.
6. It may take a few seconds the first time, but in a moment the “Outputs” window fills with the value returned by the function.

Now change your inputs or try the other function. When you click “Execute”, the response comes back almost instantaneously. This is because the client stub (**RPCDebug** uses the Perl client stub you generated) stores the list of available servers internally, so that it only queries the Broker during the first RPC, or when it cannot find an appropriate server. Figure 3.1 is a picture of the **RPCDebug** window:

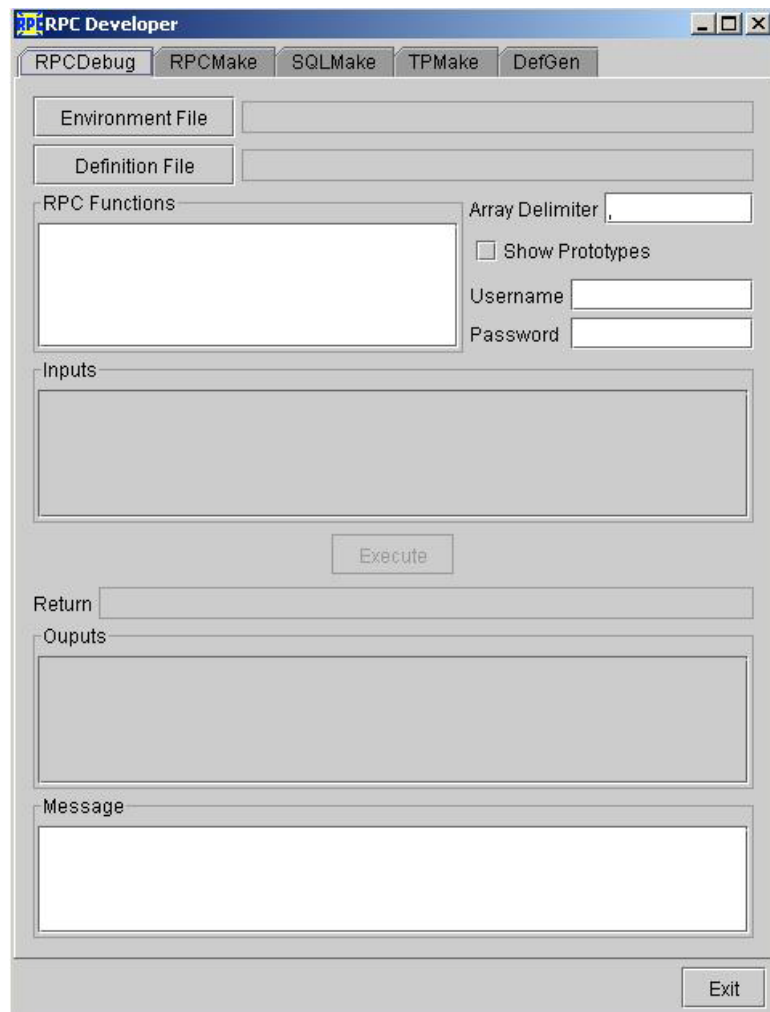


Figure 3.1: RPCDebug main window

You're finished

So now take a break, and pat yourself on the back for completing your first distributed application!

Chapert4 Understanding the Tutorial

This chapter provides you with a few more details about what you did in the last chapter. Last chapter showed you how; this chapter tells you why.

Writing server code

Server code contains only functions and procedures, because the generated server skeleton includes a main program routine that calls your server functions. This main routine accepts the arguments from the client and chooses the proper sever procedure to execute.

Once you know how to code servers, you will probably want to write server code directly, rather than splitting a local program in half, as we did in the tutorial. If you choose to write your server code directly, you need to combine the “Writing server code” step with the step “Prepare” your server code for compilation.

The one significant change that you made in the server code—substituting the `dce_` memory allocation function—is covered in detail in the section “Memory allocation for function arguments in C”.

Also, in an actual development situation, your team will probably write the definition or IDL file first, allowing client and server development to proceed in parallel.

Writing IDL file

IDL files may be the most important part of your application, because both clients and servers rely on these files for proper communication.

Once you have a correct IDL file, you can generate communications routines in any supported language or script.

Because IDL files essentially define the application programming interface (API) for each server, they allow different members of a development team to work in parallel. Using the functions define in the IDL file, your team can build the user interface and the server logic at the same time without worrying about consistency issues. This is why

your development team will usually write the IDL file *before* starting the server or client code.

Generating Skeletons

Each server uses a single server skeleton. A server skeleton needs to handle only the incoming RPCs that the server code can execute.

Because a client is not limited to accessing one server, a client can use as many client stubs as necessary. You need to generate a client stub for each server that the client is expected to access.

If you want a server to act as a client to other servers, you must generate a *client* stub from another IDL file, and include the client stub in the server code, either by compilation (for C) or by a `do` statement (for Perl).

The language of any stub must be the same as the client or server code with which it is included or compiled.

Creating auxiliary files

Broker-server-client start-up, each object scans the setting in its auxiliary file to customize its configuration. Objects use the auxiliary file setting rather than command line options.

You can specify many object attributes in the auxiliary file. *Reference* contains a complete list.

Starting applications

Naming services, application servers, and application clients must be started in a particular order.

When starting a application, the Broker must be started first, since servers do not start up unless they can register with the Broker.

Once the naming service is up and running, the servers can be started.

You must start clients last, since they're no good unless there are servers ready to be accessed.

Brokers and servers should be stated in the background, or non-interactively.

Summary

By now, you should have a basic understanding of how Nextra runtime works, and how to build a small application. You should also feel comfortable with some of the core utilities used to develop distributed applications.

The remainder of *Server Developer's Guide* is dedicated to explaining in detail how to build client and server application elements. For detailed information on the features and capabilities of Nextra tools, see *Reference or Configuration Guide*.

Chapter 5 The Development Process

This chapter discusses issues that arise from the application specification and design process recommended by Inspire International Inc. It also outlines the technical steps for implementing the client/server combinations that are at the heart of Nextra distributed applications. These sections from a high level generalization of the process you undertook in the Tutorial earlier in this manual.

A more detailed discussion of the process outlined in the latter sections of this chapter is presented in the remaining chapters.

The Nextra Development Paradigm

Education courses presented by Inspire International Inc have identified user interface design as a viable method of defining the specifications for pilot applications and smaller production applications (which use only one or two different user interfaces). High volumes, multi-user applications, however, require something more. Such systems must be optimized for speed; they must minimize redundant data; they must guarantee referential integrity; and, like any good application, they must be easy to use. For this, they require rigorous data flow analysis and data model definition.

For complex systems, the user interface design remains a very important step in the development process. When done properly, with continuous input and 'reality checks' from the end-user, this front-end design helps to nail down what the *user* think the data should look like, and what the *user* thinks his or her role in the data flow is. Naturally, the user interface should be optimized to fit these parameters.

The actual data model, however, and the actual data flow, must be modeled with system efficiency foremost in mind. Working toward application development. The power and flexibility of today's development software allow an application to have the best of both worlds: an extremely friendly user interface as well as a data model optimized for speed. The 'trade-off' no longer exists.

With the twin goals of friendly user interface and data model optimization in mind, the initial design of distributed applications involves three components:

- Involvement of users during development
- Definition of the data model
- Structured analysis of the data flow

While a detailed explanation of these processes is outside the scope of this book, the following guidelines and hints should help you determine how to structure your application.

Involvement of users during development

Developers must maintain contact with the end users throughout the development process. Not only will this save testing time when the pilot system is rolled-out, but it may also provide developers with some new perspectives, and possibly spark more elegant solutions. In addition, users should be allowed to specify changes during this stage, has been put into production. At all stages, the development team members should remember that they are building system to make the users' work easier and more efficient.

Definition of the data model

Users should draft a comprehensive list of the data to which they need to access, such as names, addresses, birthdays, past purchases, account information, etc. In addition to the kinds of information they want, users should also indicate how much of each kind: whether their idea of an address is 20 characters or 100; whether they envision a comment field of 40 characters or 4,000.

Once all users have agreed on the complete data set, it is the design team's responsibility to define a Data Model which includes specifications of how data entities relate to one another, and the degree of the relationship (i.e., one to one, one to many; mandatory to optional, mandatory to mandatory, etc.). The Data Model should be designed to balance the following considerations:

- Minimize coding-deliver data to users via elegant solutions
- Minimize data redundancy - save storage space and limit possible disarrangement of data
- Maximize performance-use indexes and data body relationships that help keep database transactions fast
- Maximize integrity-design methods of database access that prohibit data corruption

While the first three considerations have changed little with the advent of a three-tiered environment, new issues arise concerning how to include referential integrity in a distributed application. The following guidelines should help your design team decide how to guarantee data integrity.

Structured analysis of the data flow

Users and developers must analyze how data will be transformed by users, and how it will be processed between users and databases.

During this process, the design team should decide in which tier to locate functionality. The goal here should be to specify the minimum data processing that needs to occur within the user interface, and to specify the larger amount of processing that should be coded into servers in the functionality tier. Further, the functionality should be divided into servers based on some method of this approach will be a system that is as modular as possible. Adding or subtracting user interfaces, servers, and additional databases should require the fewest possible changes to other modules, and the smallest amount of code in the new module itself.

For example, if a Visual Basic user interface contains code to pass a field through a complex calculation, multiply by a second field, and put the result in a third field, all this code must be translated and rewritten if a Power builder version of the same interface is brought on line. In contrast, if this functionality were localized to a server, the new PowerBuilder interface would need only to make a remote procedure call. The server code would not require any changes.

Development Process Outline

This section provides a high level template for the steps necessary to build a server and a client and to debug the interaction between team.

For specific details, see the chapters that follow, “Building Next-generation Servers” and “*Client Developer's Guide*.”

Setting the environment variables

Please refer to “Environment variable set-up after installation” in *Read Me First*.

Building the distributed application

After designing the user interface, after planning the architecture for the distributed application, and after verifying that the appropriate environment variables have been set, perform the following steps for *each* client-server relationship. (The specific methods will depend upon the programming languages in which you implement the server and client, and upon the platforms on which the server and client will run.)


Building the server

1. Write the definition file or IDL file.

the remote function calls handled by the server: specify the name, return value type, and parameter types for each RPC that the server will execute.

2. Write the server code.

functions that the server will provide via RPCs.

	<h3>Debug</h3>
<p>Test this server code as a local program-by temporarily appending a set of local function calls to the program and running it-to check for syntactical errors, and to make sure function calls return the correct results.</p>	

3. Generate a server skeleton.

Pass the definition file and the file containing the server code as inputs to the **RPCMake** utility, to automatically generate a server skeleton.

In necessary, move the skeleton to the platform on which it will be tested. If the server is implemented in a compiled language (C or COBOL), compile the executable server.

4. Create or edit the auxiliary file.

Test platform must reflect the location where the Broker(s) will be running.

The server can now be executed.

**Debug**

Test the new server, before implementing it in the runtime distributed application.

To test:

- Build and run a test client that makes remote function calls to the new server, or
- Use the Object Interface Tester, **RPCDebug** utilities to interactively make these calls.

If the server is implemented in a compiled language, move the server execute table and environment file to the production platform.

If server is implemented in an interpreted language, move the server skeleton, the file(s) containing the server code and the environment file to the production platform.

Building the client

1. Write the client code.

Write code that makes remote procedure calls. (Refer to the server's RPC file to review the proper syntax for making a particular remote function call)

**Debug**

Test this client code as a local program-by commenting out remote function calls and running the program-to check for syntactical errors.

Where the client is a graphical user interface, the code has probably been mostly completed during the GUI design and prototyping stage.

2. Generate the client stub(s).

For each server that the client will access, pass the server's IDL file as the input to the **RPCMake** utility, to automatically generate a client stub. Move the stub(s) to the client test platform, if necessary.

3. Set or edit the auxiliary file.

The client program must link in or access the Nextra runtime library, as well as link in or access the client stub(s)

If the client is implemented in a compiled language, compile the executable client.

4. Create or edit the auxiliary file.

For Nextra clients, the environment file on the client test platform must reflect the location where the Broker(s) will be running.

At this point, the client is empowered to invoke any function performed by the server(s) whose skeleton(s) it contains.

If the client is implemented in a compile language, move the client executable and environment file to the production platform.

If the client is implemented in an interpreted language, move the client stub(s), the file(s) containing the client code and the environment file to the production platform.

Debugging distributed applications

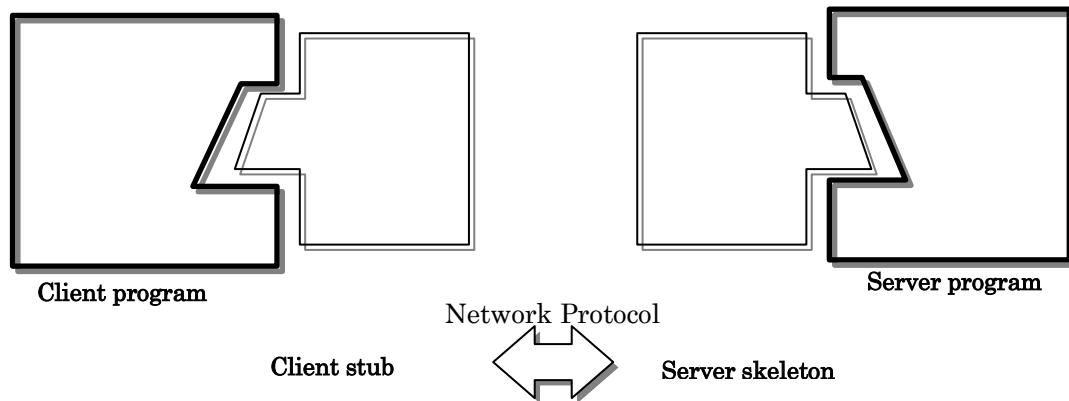
In contrast to local applications, distributed applications have two dedicational sources of error. Not only must the code that the developer has written be syntactically correct and procedurally effective:

- The communication mechanism must have been generated properly, and
- The client and server platforms must be able to communicate.

In the previous sections on building a server and building a client, debugging steps were inserted in a sequence that allows a developer to test a distributed application in a modular manner. What follows is another look at the debugging process, seen as a unified process:

When debugging an application with parts distributed over a network, the most important concept to keep in mind is isolation of errors. On the largest scale, each new part must be completely free of errors before it is added to a runtime system. At the developer's level, each step of the development process must be debugged to prevent errors from propagating into generated code, and to guarantee that the next step has a solid foundation on which to build. In addition, proper debugging

techniques (i.e., isolation of errors) will save you many hours of stress by limiting an error to one or two possible sources.



1. Test the server as a local program.
2. Test the server from a local client.
3. Test the client code as a local program.
4. Test the client-server interface.

Figure 5.1: Debugging methodology

As long as adherence to this debugging method is strict, the developer will be able to isolate errors at each debugging step, greatly reducing overall development time.

1. Test the server as a local program.

By testing the server code as a local program, before generating a server skeleton, you can isolate difficulties to syntactical and procedural errors in the server code itself, without worrying that errors might be arising from network communication problems.

2. Test the server from its own platform.

By testing the executable server from its own platform, after performing step 1 and before making that server accessible to clients on other platforms, you can limit difficulties to errors in the IDL file or server code, without worrying that error might be arising from network communication.

3. Test the client code as a local program.

By testing the client code as a local program, before generating a client stub, you can isolate difficulties to syntactical and procedural errors in the client code itself, without worrying that errors might be arising from network communication problems.

4. Test the client-server interface.

By testing the client-server interface after performing step 1, 2, and 3, you can isolate difficulties to a failure of network communication or differences between the client's function call and the server's function definition.

Within each of the above steps, we recommend a general debugging approach that will help you to locate and fix errors quickly. The steps to keep in mind are as follows:

- Verify the syntax and functionality of hand-written code before generating other code.
- Where possible, cross-check definition files and IDL files against the inputs, outputs, and data types of the actual procedures.
- After generating stubs (and compiling where necessary), use the **RPCDebug** utility to verify that all of a few server's procedures work properly.
- Write clients for testing that automate function calls and execute 100% path and statement coverage of server code. (Note that this step must be tailored specifically to your application.)

Chapter 6 Building Functionality Servers

This chapter covers the general process for building and debugging Nextra servers, with a focus on functionality servers. For more detailed information about building data access servers, see the appropriate chapters in this book. For detailed information on building transaction processing or remote connectivity servers, see the appropriate manual.

Because this chapter refers to concepts introduced earlier, we recommend that you read all previous chapters or try to build a server.

Please refer to "[Chapter 9 Building Java application server](#)" for Java application server development after reading this chapter.

Writing a IDL file

The IDL file is a text file in which you define the functions that a server is capable of executing.

The IDL file has a two-fold significance: First, by feeding the IDL file into the **RPCMake** utility, you can generate stub/skeleton automatically, avoiding tedious hand-coding. Second, by defining server within the IDL file, you can formalize the specification of the server interface so that other programmers can code to that specification without concern for how the server itself is implemented. After you have created the interface IDL file, both the client and the server teams can code to it without depending on one another. *Programmers are capable of parallel development within a distributed environment.*

Interfaces, and therefore interface IDL files, are language-independent. This means if you have a server written in COBOL, and another server in C that executes identical functionality, you can use the same IDL file for both servers.

Assembling server information

To write the IDL file for a server, you need to gather the following information:

1. The name and universal unique identification number (UUID) of the server.

The Nextra runtime library identifies interfaces by name, so an interface name must be supplied.

The interface name must consist of a combination of alpha-numeric characters or the underscore”_”.

Each interface name must be unique among all interfaces on your network.

The UUID field in the IDL file must be included, but the contents are ignored.

2. The version number of the server.

You must associate a version number with each server. The version must be included, but the value is ignored.

3. The names and return-value types of all functions handled by the server.

The function names you specify in the interface IDL file must match the actual function names in your server code. Function names must be unique among all functions that a given client might call.

4. The names and data types of all input and output parameters for each function.

Input or output status is determined relative to server. An input parameter is passed from the client into the server only. An output parameter is passed from the server to the client.

When choosing names for parameters, bear in mind that only the *sequence* of parameters has to match that of the corresponding parameters in the server code. The names of parameters declared in the IDL file do not have to match those used for the corresponding parameters in the server code. Variables are referenced by position, not by name.

Please see “Data Type Information” of *Reference* for a complete list.

File syntax

Once you assemble this information, then you may write the IDL file using the following format:

```
# RPC Interface Definition File for: interface_name
[uuid(uuid) version(ver)]

interface <<$>>interface_name {
    <<declarations>>
    data_type procedure_name (
        <<[in]|[out]>> data_type variable_name<<[]>> <<,
        ...
        <<[in]|[out]>> data_type variable_name<<[]>> >>);
    data_type procedure_name (
        <<[in]|[out]>> data_type variable_name<<[]>> <<,
        ...
        <<[in]|[out]>> data_type variable_name<<[]>> >>;
    }
}
```

Symbol	Meaning
#	When first non-space character of a line, # identifies the rest of the line as a comment.
<i>UUID</i>	A UUID. This value must be present but is ignored.
<i>ver</i>	a version number for the server. This value must be present but is ignored.
<i>interface_name</i>	Identifies the server by a name unique among all servers on a network. If the name is preceded by a dollar sign "\$", then the server name becomes variable, allowing you to specify the server's name upon start-up. Please refer to "Variable named server" in Configuration Guide for more information.
<i>data_type</i>	One of: char, double, float, int, short, long, void.
<i>procedure_name</i>	Identifies the remote function by a name unique among all functions available in an application. The client program uses <i>procedure_name</i> to invoke the remote function and <i>procedure_name</i> is defined in the server program. You can not use the <i>procedure_name</i> starting with dce_, DCE_, ODE_, ode_.

[in] [out]	Identifies each variable as either Input or Output.
<i>variable_name</i> <<[]>>	Identifies a parameter by name unique among all variable names within a given function. Parameters are not required to match the parameter names defined in the server code.

Table 6.1: IDL Syntax

Examples of IDL files

These IDL files are associated with the C, COBOL or Perl server code listed below.

IDL file for C or Perl

```
server.def

#IDL file for:cserver (or perlserver or cblserv)
[uuid(123) version(1.0)]

interface cserver {
    long c_add (
        [in] long x,
        [in] long y,
        [out] long z);
}

```

If you want clients to contact either server interchangeably, you could call any of the servers by the same name (`cserver` or `perlserver`), regardless of the fact that they could be written in different languages. If you wanted to distinguish between the two, then you would need two separate IDL files, each with a different `interface` name. (You could also create a variable-named server. See *Configuration Guide* and *Reference* for more information.)

Naming conventions

The name of an IDL file is appended by the suffix `.def`. It is often helpful to name the prefix after your interface. For example, `cserver.def` (or `perlserver.def`) is the name of the IDL file above.

Data type declarations

As an illustration of data type declaration, the IDL file associated with the server code that implements the `send_file` function is listed below.

```
file_server.def

#IDL for: file_server
[uuid(123) version(1.0)]

interface file_server {
    short send_file (
        [in] char filename[],
        [out] long arraysize,
        [out] char outarray[arraysize]);
}
```

Including COBOL size information

The COBOL size file and the IDL file must both be used as input to the **RPCMake** utility in order to generate COBOL stubs that are compatible with NULL-terminated arrays.


The COBOL size file must contain the same function names as appear in the interface IDL file. For each NULL-terminated array, add an entry in the COBOL size file below the function label according to the syntax below:

```
#cobol variable size [number]
```

<i>variable</i>	Any null-terminated array in the interface definition file (e.g. a column name is a select statement). If an alias is used for a column name within an SQL statement, the variable name must correspond to the alias, not the column name.
<i>size</i> (Example: COLUMN)	is the byte size of each element of the array (e.g. the size of the largest element in the column, or the maximum size allowed by the column definition). Common for both 1 & 2 dimensional. Note that you must terminate with NULL using

	dce_null_terminate() API in the user code when using string data.
<i>Number</i> (<i>Line:ROW</i>)	is the number of elements in the array (e.g. the maximum number of rows expected to be returned from the database). Default is 200, or as set by <code>rcmax</code> (see below)

Table 6.2: COBOL size file syntax

	<p>Fixed size string array</p> <hr/> <p>This is the case specifying the size in the IDL file, not using COBOL size file. Unlike Constrained string array, you are not required to terminate with NULL at the end character. However, you can use size – 1 bytes for the data.</p>
---	--

By defining the following statement under the function label, you can assign *number* for the variable.

```
#cobol rcmax number
```

Example COBOL size file

```
get_tt:
#cobol rcmax 400
#cobol tool_type_name 40 rcmax
get_tl:
#cobol tool_name 40
```

- The variable "tool_type_name" is a 2-dimensional constrained string array defined as 400 rows and 40 columns in the function "get_tt".
- The variable "tool_name" is a 2-dimensional constrained string array defined as 200 rows and 40 columns in the function "get_tl".

Moving on

For more information on supported data types, see "IDL Data Types" of *Reference*. For more information on IDL files, see the "File Specifications" chapter in *Reference*.

You are now ready to proceed to the next step: generating a server skeleton.

Writing Server Code

Server code consists of units of functionality packaged within procedure or function definitions. Server code may contain any number of procedure definitions, as long as each procedure has been defined in the server's IDL file. Each server function must use the return value type and the parameter types defined for it in the IDL file.

Include files

C include issues

Include the header file that is generated when you create the server skeleton (see "Generating a Server skeleton" for more information), and the standard RPC header `dceinc.h`, found in the directory.

Perl include issues

In Perl, be sure to add a `do` statement on the first line that includes the file `$ODEDIR/lib/dce_func.pl`. Also, the last line should consist of the numeral one followed by a semi-colon, to guarantee successful inclusion of your server code with your server skeleton.

Example server code

C example

The following is an elementary example of server code written in ANSI C:

```
cserver.c

#include "cserver_s.h"
#include < dceinc.h >
long c_add(long x, long y, long *z)
{
    *z=x+y;
}
```

This short example, called `cserver.c`, meets all the requirements of server code. You could compile it with its stub (generated in a later step) and run it as a server.

COBOL example

The following is an elementary example of server code written in COBOL:

```
cblserv.cbl

IDENTIFICATION DIVISION.
PROGRAM-ID. CADD.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(9) BINARY SYNC.
01 Y PIC S9(9) BINARY SYNC.
01 Z PIC S9(9) BINARY SYNC.

PROCEDURE DIVISION USING X, Y, Z.
DO-C-ADD.
MOVE 0 TO Z.
ADD X, Y TO Z.
GOBACK.
END PROGRAM CADD.
```

Call this file `cblserv.cbl`

Perl example

The following is an elementary example of server code written in Perl:

```
perlserver.procs or perlserv.pro

do "$ENV{'ODEDIR'}/lib/dce_func.pl"
sub perl_add{
    local($x,$y,*z)=@_;
    $z=$x+$y;
}
1;
```

Call this file `perlserver.procs`.

Note that the following line must be the last line in any file or Perl server routines:


```
1;
```

This is required because the Perl server skeleton includes the server routines with a `do` statement. The value 1 indicates successful return from including the server routines.

Do this step at the end of the file, not at the end of each routine.

Technically, your last server function must return a non-zero value. Using the “1” routine is just one way to ensure that this is the case.

General coding issues

	<p>Reserved words</p> <p>OEC_, oec_, dce_, DCE_ are reserved for use by Nextra. Use of names starting with these characters may result in error or unpredictable behavior.</p>
--	---

Passing C pointer data types – arrays of strings

When passing arrays of strings, it is important to make sure that the array is NULL-terminated.

For example, if you want to send an array of three strings (*instring1*, *instring2*, and *instring3*) to a server, and want the server code to process each string to *outstring1*, *outstring2* and *outstring3* respectively, allocate memory for the array of out strings as:


- *pointer1*-> *outstring1*
- *pointer2*-> *outstring2*
- *pointer3*-> *outstring3*
- *pointer4*-> normally undefined.

If *pointer4* is left undefined, it may assume any value, including NULL, causing the server to crash with a SIGSEGV error. Therefore,

- *pointer4* must be set to *NULL*.

COBOL issues

Your server will consist of PROGRAM modules, which will be dispatched by the RPCMAIN module.

	<p><u>Return value</u></p> <p>If, in the IDL file, the declaration of a COBOL server function has a return value, you must add an extra integer parameter at the end of the parameter list for that function. COBOL does not support C-like return values. In addition, the value to be returned must be explicitly assigned in the extra variable, so remember to declare the variable in the Linkage section of your code. The value of this variable is marshaled back to the client as the return value, and it appears to the calling program as if the COBOL routine supports C-like return values. <code>DCE_RESULT</code> is the variable which holds the return value.</p>
---	--

When calling a Nextra function, pass pointer parameters by reference, and regular parameters by value.

NULL termination

In COBOL code, the constrained string arrays which define the column size in the COBOL size file must be null-terminated.

There are two ways to null-terminate variables whose values are not yet known. The first way is to call `dce_null_terminate()` API for each variable. This method does not work if there is a space inside the string. Call these functions as follows:

```
CALL "dce_null_terminate" USING login.
CALL "dce_null_terminate" USING passwd.
```

The second method may be used for both variables and constants:

```
STRING "server.env" DELIMITED BY SIZE, LOW-VALUE
                    DELIMITED BY SIZE INTO ENVFILE.
CALL "dce_setenv" USING ENVFILE, DCE-NULL, DCE-NULL
                    giving RV.
IF RV = 0
```

```

    DISPLAY "SET ENV WITH ", ENVFILE, " FAILED."
    STOP RUN
END-IF.

```

Example of COBOL arrays translating to C arrays

When you declare array variables of type char, in the IDL file, remember that a terminator character will take up the last space.

For instance, declare as follows in the IDL file:

```

[in] char item_in[],
[out] char item_out[],

```

and you want to have 10 bytes in data, declare an array of size 10 at the LINKAGE SECTION in the COBOL server logic as:

```

01 item_in PIC X(10).
01 item_out PIC X(10).

```

declare as follows in the COBOL size file:

```

#cobol item_in 11
#cobol item_out 11

```



Perl issues

Perl always uses a variable argument list, so that if you supply 20 arguments for an RPC that only has 10 parameters, the last ten parameter values are ignored. If you then try to read the extra parameters from the perl client, they could contain anything.

For more detailed instructions on writing Perl servers, see “rpcperl” in *Reference*.

Memory allocation for function arguments in C

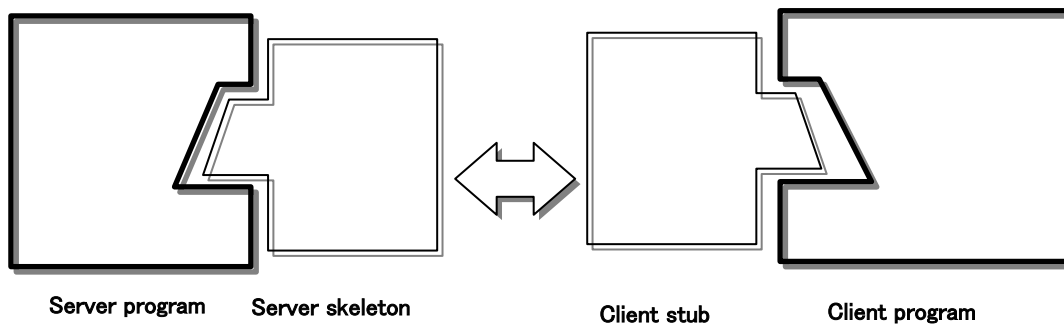
The C language and its supersets require you to encode memory management into programs involving data structures defined as pointers. You must write code to allocate memory (using the `malloc`

family of functions) for data referenced by these pointers, and then to free up memory after usage.

In a local C program, you can allocate and free memory at will to prevent memory leaks. In contrast, the client/server model requires that you work with two distinct modules accessing each memory, and server code and stub accessing the server machine memory.

Using standard `malloc` functions can cause problems in an open distributed environment. For example, if your server allocates space for dynamic output arguments to be returned to the client, you shouldn't free that space until after the arguments have been sent across the network. But the server code returns execution to the server skeleton before anything goes out on the network, so there is no way for your server code to free the memory. Only the stub can do it.

Similarly, if the client stub allocates memory for incoming dynamic arguments, it can't free that memory space, because your client code uses those arguments after the stub turns over execution. Once your client code has finished using those arguments, only then can the space be de-allocated.



1. The server code allocates specified memory.
2. The server skeleton sends information over the network and frees the memory allocated by the server.
3. The client stub receives information from the server skeleton and allocates memory.
4. The client code reads the received data and frees the memory allocated by the client stub.

Figure 6.1: Memory allocation using Nextra

Nextra solves the memory allocation problem by providing a family of modified allocation functions. These functions (`dce_malloc()`,

`dce_realloc()`, `dce_calloc()` are used with the same parameters and return values as their standard C cousins. You should use these functions to allocate server machine memory for dynamic output arguments (NULL-terminated and constrained arrays). After sending outputs over the network, your server skeleton automatically calls another Nextra function, `dce_release()`, which frees all memory allocated by Nextra memory management functions in your server code.

On the client side, your client stub uses the `dce_malloc()` family to allocate space for incoming dynamic arguments. You must add calls to `dce_release()` in your client code to free that memory when you are done using it.

Nextra automatically handles all allocation for arguments being passed into the server. Thus, only output argument types need to appear in the table above. Nextra memory functions are *not* necessary when sending information from the client to the server. This asymmetry arises from the server's nature as a shared resource.

These IDL data types require <code>dce_malloc()</code> in the server code (for parameter <code>x</code> only) and <code>dce_release()</code> in the client code	
<code>[out] char x []</code>	<code>[out] long n</code> <code>[out] any_type x [n]</code>
<code>[out] char x [][]</code>	<code>[out] long n</code> <code>[out] char x [n][n]</code>

Table 6.3: Dynamic data types

Example of C memory allocation

To illustrate the usage of the `dce_malloc()` function, consider the following server code example, which allocates memory for a constrained array, in order to send a file to the client that requests it (error-handling has been removed from this code in order to highlight the strategy for sending a binary file to the client):

```
#include <stdio.h>
#include <string.h>
#include "file_server_s.h"

short send_file(char *filename, long *arraysize, char
**out_array)
{
    long filedescr, returnval, maxlen, packet=1000,
```

```

                                offset=0;
char *buffer;
FILE *fileptr;

maxlen=packet;
buffer=(char *)dce_malloc(maxlen);

fileptr=fopen(filename,"r");
if (fileptr == NULL) {
    printf("ERROR: unable to
                                open file <%s>\n",filename);
}
filedescr=fopen(fileptr);
while ((returnval=read(filedescr,&(buffer[offset]),packet))>0)
{
    offset+=returnval;
    maxlen+=packet;
    if (returnval<packet)
        break;
    buffer=(char *)dce_realloc(buffer, maxlen);
}
*out_array=buffer;
*arraysize=offset;
fclose(fileptr);
return 1;
}

```

This code allocates memory for an output argument defined as a constrained array. Note that, where a `malloc()` function is normally expected, a `dce_malloc()` function is called. The server skeleton for this server calls `dce_release` after it sends the file across the network, thereby freeing the memory allocated to `buffer` in the code above.

Windows static arrays

Windows applications can optionally use a module definition file when they are compiled. When using static arrays in either clients or servers on Windows, you must set `HEAPSIZE` in your definition file to be large enough to accommodate the arrays. For example:

```

char array1[ARRAY_SIZE1];
char array2[ARRAY_SIZE2];

```

If arrays are defined as above, then the `HEAPSIZE` statement in the application's module definition file must have a value greater than the size of the arrays:

```
HEAPSIZE ARRAY_SIZE1 + ARRAY_SIZE2 + (other
variable space)
```

Note that you total the three numbers to get a total value, rather than putting the actual variable names into the file. Thus, for `ARRAY_SIZE1=6000` and `ARRAY_SIZE2=7000`, you might enter this line into the module definition file:

```
HEAPSIZE 16384
```

Naming conventions

The name of a file containing server code should be appended by a suffix reflecting the specific language in which the code is written.

Language	Suffix
C	.c
COBOL	.cbl
Perl	.procs

Table 6.4: Server code naming conventions

Use only the following characters in the server name:

- Upper case letters (A-Z)
- Lower case letters (a-z)
- integers (0-9)
- underscore (_)

Debugging the server code

By testing the code at this early stage, you can localize difficulties to syntactical or procedural errors in the server code itself, avoiding possible headaches later on.

Test the server code by temporarily appending a set of local function calls.

1. Comment out the last line.

For Perl server code, comment out the last line (The line that reads `1 ;`).

For C server code, skip this step.

2. Add a main routine that calls your defined functions.

This routine should call the procedures locally, allowing you to spot errors before the system becomes more complex through the addition of network transport.

3. Compile the server.

Skip this step for Perl servers.

4. Execute the program.

Check for syntactical errors. Verify that the results returning from the function calls are what you expect.

C example

For example, the following lines could be appended to the example C code.

```
main()
{
    long a,rv;
    rv=c_add(1,2, &a);
    printf("The return value of add is %d \n", a);
}
```

This program would then be compiled using a C compiler, and executed to verify that the right results are returned.

Perl example

The following lines could be appended to the example Perl code.

```
&perl_add(1,2,*a);
```

```
print("The return value of add is $a \n");
```

This program would then be executed (by typing `rpcperl perlserver.procs`) to verify that the proper results are returned.

Moving on

Once you have finished writing the server code and testing it as a local program, proceed to the next step: generating a server skeleton.

Generating a server skeleton

After writing the IDL file, you may use the **RPCMake** utility to generate the server skeleton and client stub for that interface. You need to specify the following information:

- the name of the IDL file;
- the language in which the server skeleton should be written.

Each server uses only one server skeleton. A server's stub must be generated in the same language used to write the server.

To generate a server skeleton, you have the option of calling the **RPCMake** from the command line, or invoking a graphical interface.

Calling the RPCMake Generator from the command line

To call **RPCMake** and generate a server skeleton, use the syntax:


```
rpcmake -d file.def -s language1 [-p file.procs] [-size file.size] [-k]
```

Where *file.def* is the name of your interface definition file, and *language1* is an abbreviation for the language in which you want the server skeleton to be generated. Use `c` for C or `mfcobol` for COBOL or `perl` for Perl.

If you are generating a Perl server skeleton, you need to include the `-p` option, which flags the file containing your Perl server code.

If you are generating a COBOL server skeleton, and your code deals with NULL-terminated arrays, you must include the `-size` option, which specifies the COBOL size file.

You have seen the simplest form of **RPCMake**. It is possible to make one call to **RPCMake** and generate client and server skeletons in multiple languages. To generate a client stub, you need only use `-c` instead of `-s` with the language abbreviation. You can specify multiple comma-separated languages with each `-c` or `-s` option. See **RPCMake** of *Reference* for a full description of its capabilities and features.

	<p><u>If something goes wrong</u></p>
	<p>If RPCMake reports an error, check two things: (1) Make sure that the <code>ODEDIR</code> environment variable is set correctly for the Nextra toolkit you are using, and that the <code>PATH</code> environment variable includes the <code>\$ODEDIR/bin</code> directory; (2) When specifying the IDL file, either verify that the IDL file is located in the directory in which RPCMake is invoked, or specify the full path name for the IDL file.</p>

C example

To illustrate the usage of **RPCMake**, the following command would generate a C server skeleton from the IDL file `cserver.def`:

```
> rpcmake -d cserver.def -s c
```

COBOL example

This command would generate a COBOL server skeleton for the server code held in `cblserv.cbl`, using the definitions in `cblserv.def`:

```
> rpcmake -d cblserv.def -s mfcobol -size cblserv.size
```

Perl example

This command would generate a Perl server skeleton for the server code held in `perlserver.procs`, using the definitions in `perlserver.def`:

```
> rpcmake -d perlserver.def -p perlserver.procs -s perl
```

Calling the RPCMake with a GUI

To invoke the **RPCMake** with a friendly graphical user interface (GUI), type the command:

```
> rpcmake
```

Once **RPCMake** is invoked, the following window appears:

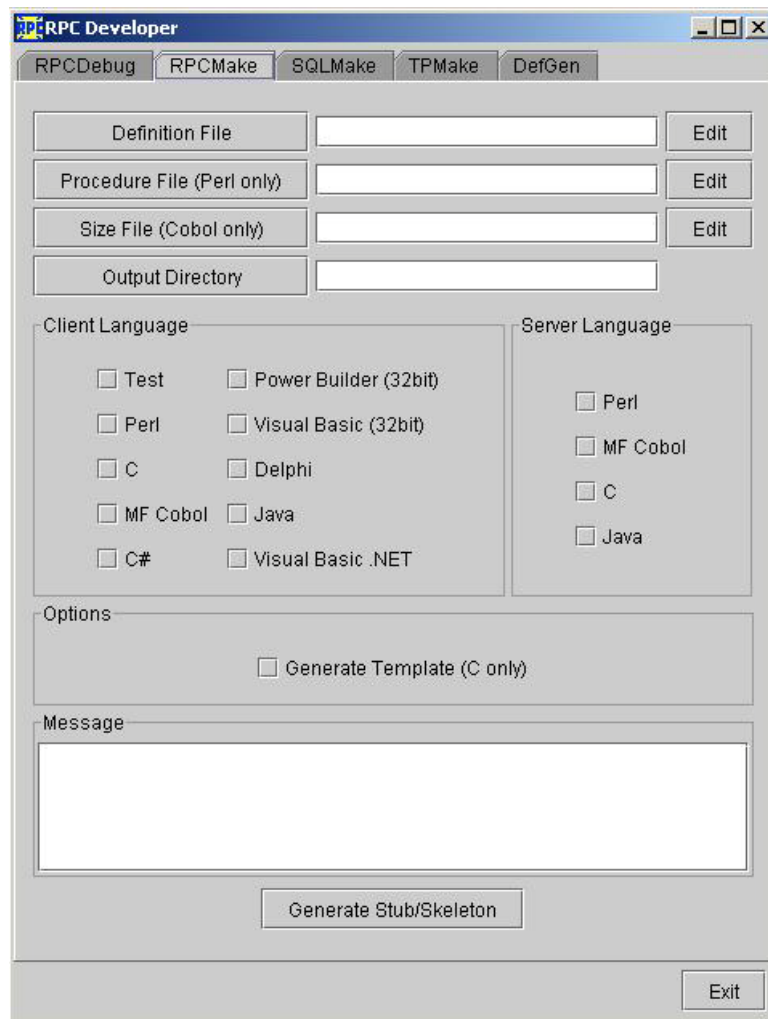


Figure 6.2: RPCMake GUI

1. Enter the name of the IDL file.

Either type the name of the IDL file in the appropriate text box, or click on the “Definition File” button. If you choose the second method, then another window appears, as below:

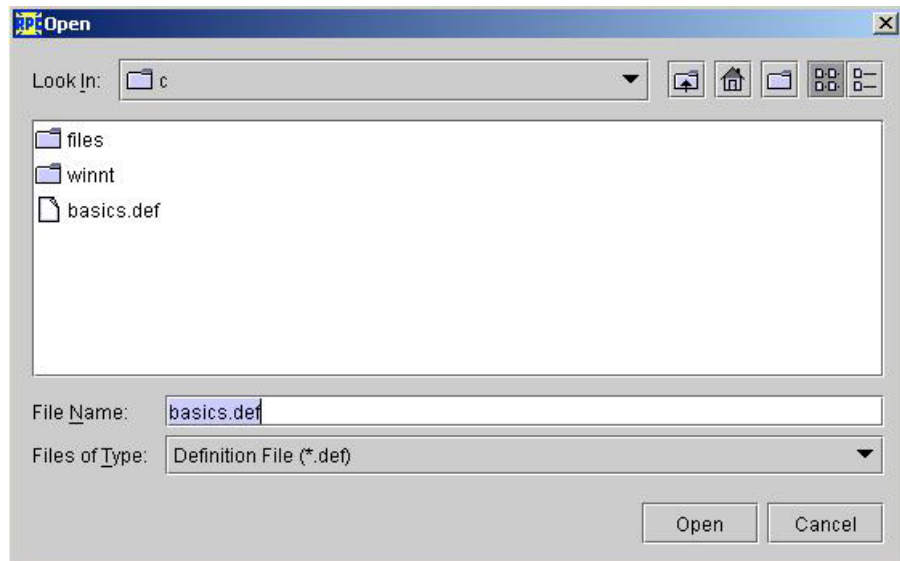


Figure 6.3: Find file window

You can then select a IDL file by clicking on it in the “Files” list box and clicking “OK”.

2. Back in the primary RPCMake window, enter the name of the procedure file if using Perl.

Leave the “Procedure File” field blank unless you plan to generate a Perl server skeleton. If you are using Perl, enter the name of the file containing your Perl server code.

3. If using COBOL, enter the name of the COBOL size file in the “Size File” field.

Leave the “Size File” field blank, unless you plan to generate a COBOL server skeleton. If you are using COBOL, enter the name of the file containing COBOL NULL-terminated array size information.

4. Select server and/or client stub languages.

Click one of the language option buttons under “Servers:” to select the server skeleton language. The language used for your server skeleton and your server code must be the same. Click on the language option buttons under “Clients:” to select client stub language(s).

5. Edit files if you need to.

Clicking the “Edit” button invokes a text editor for modifying files on the fly.

6. Generate stub/skeleton(s).

Click in the “Generate Stub/Skeleton” button to generate all selected stub/skeletons.

7. Exit from RPCMake.

Click in the “Exit” button.

Files generated

After you finish your session with **RPCMake**, your current directory contains a new file for each `-s` option specified on the command line, or for each language checkbox selected in the GUI.

Also, if you generated any stubs in C, your directory contains one or two header files. The file `server.h` should be included I your client code, while the file `server_s.h` should be included in your server code. See the example.

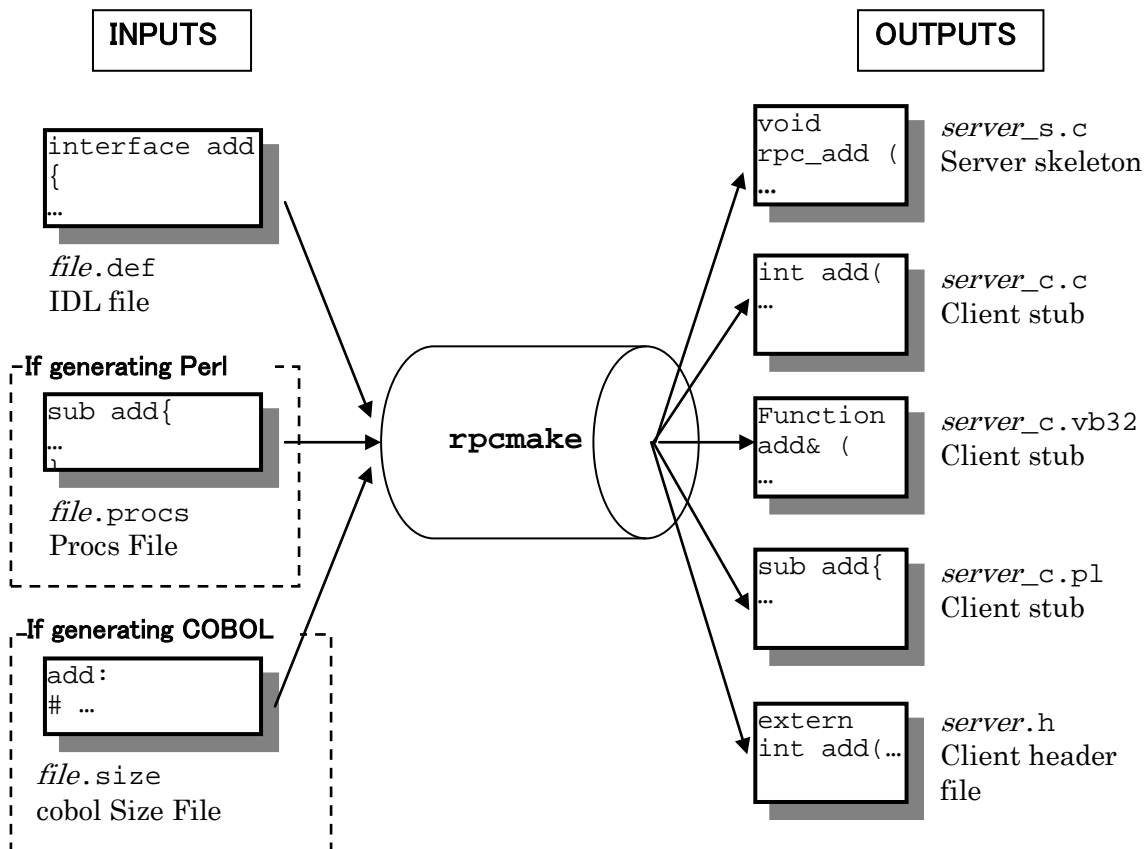


Figure 6.4: Files associated with Nextra RPCMake

Example

The example **RPCMake** command (“C example”) generates the following files:

```
cserver_c.c
cserver_s.c
cserver.h
```

Naming conventions

The prefix of a stub name is derived from the interface name specified in the IDL file. The suffix of a stub name reflects whether the stub is a client stub or server skeleton, as well as the specific language in which the code is implemented.

Language	Server skeleton suffix	Client stub suffix
C	_s.c	_c.c
COBOL	_s.cbl	_c.cbl
Perl	_s.pl	_c.pl
Java	_s.java	_c.java

Table 6.5: Stub/Skeleton naming conventions

Moving on

Once you have created a server skeleton, you are ready to proceed to the next step: compiling your server.

Compiling a Server

If your server is in Perl, skip this section.

Compiling in COBOL

For servers written in COBOL, the next step is to compile the executable server.

```
> cob -xe "RPCMAIN" cobol_files libraries -o executable_name
```

The `x` option tells the compiler to generate an executable, as opposed to object files. The `e "RPCMAIN"` option specifies that the entry function is named "RPCMAIN". This information is important, because the name of that function is hard-coded to be "RPCMAIN" in the server skeleton.

cobol_files should be the list of COBOL source files, e.g. `cb1serv.cbl`.

libraries should be the list of libraries needed; for Nextra COBOL programs, this would be at least `librpc.ext`, `librpccobol.ext`.

executable_name is the name of the executable to be created.



Compiler options

Refer to `Makefile` or `makefile.nt` for Windows coming along with Nextra samples.

Compiling in C or C++

For servers written in C or C++, the next step is to compile the executable server. Compilation of servers written in the ANSI C family of languages follows these steps:

- The object files from the preceding compilations must be linked together with the runtime library. For UNIX, `librpc.ext` is located in the installation directory `$ODEDIR/lib`.
- For all steps in the compilation process, the developer must pass platform-specific flags to the compiler.

UNIX makefile example

As an example of C server compilation, the `makefile` listed below compiles a server with its stubs in a UNIX environment when you type `make server`:

```
SERV = server_name

CC = `getplatform cc`
LD = `getplatform ld`

SOBJ = $(SERV).o $(SERV)_s.o
COBJ = $(CLNT).o $(SERV)_c.o
```

```

EXTRALIBS = `getplatform netlib`
LIBS = `getplatform libdir` `getplatform lib`
INCS = -I$(OEDIR)/include `getplatform inc`

.c.o:
    $(CC) -c $< $(INCS)

all: server

server: $(SOBJ)
    $(LD) -o $(SERV) $(SOBJ) $(LIBS)

clean:
    rm -f *.o
    rm -f $(SERV) $(CLNT) *_[sc].*

```


For *server_name*, you need to substitute the name of your source code file, with the `.c` extension removed. For our example, `cserver.c`, you would use the following:

```
SERV = cserver
```

For *compiler_command*, you need to substitute a platform-specific compiler command. The table below shows what commands to use.

OS	Set CC equal to this value
HP-UX	<code>cc -Aa -D_POSIX_SOURCE</code>
IBM AIX	<code>cc</code>
Solaris	<code>acc -D__unix -Dunix -D_REENTRANT -D_POSIX_SOURCE -D_solaris_ -DODEDCE</code>
Windows	<code>CL</code>

Table 6.6: compiler commands

	<p>Advanced makefiles 101</p> <p>If you want, you can use your makefile to do other tasks in addition to compiling. Add the following lines to force your Makefile to call RPCMake to generate a server skeleton, so that you don't have to do it by hand:</p> <pre>\$(SERV)_s.c: \$(SERV).def</pre>
---	--

```
rpcmake.real -d $(SERV).def -s c -y
```

Or, if you add the lines below, you can remove unnecessary files by entering `make clean` or `make cleanup`:

```
clean:
```

```
rm -f *.o
```

```
cleanup: clean
```

```
rm -f $(SERV) $(CLNT) *_[sc].*
```

Windows makefile example

The following is an example makefile that can be invoked using `nmake`. The example compiles both a client and server.

```
#THE FOLLOWING NEED TO BE SET:
# ODEDIR= Directory where Entra tools are installed
# SERV= Server Executable name
# CLNT= Client Executable name

SERV = basics
CLNT = cclient

CC = cl /nologo /I $(ODEDIR)/include
LD = cl

SOBJ = $(SERV).obj $(SERV)_s.obj
COBJ = $(CLNT).obj $(SERV)_c.obj
LIBS = -link /SUBSYSTEM:console /NOLOGO
$(ODEDIR)/lib/librpc.lib
INCS = -I$(ODEDIR)/include

.c.o:
    $(CC) -c $< $(INCS)

all: server client

server: $(SOBJ)
    $(LD) -o $(SERV) $(SOBJ) $(LIBS)

client: $(COBJ)
```

```

$(LD) -o $(CLNT) $(COBJ) $(LIBS)

$(SERV)_s.c $(SERV)_c.c $(SERV).h: $(SERV).def
    rpcmake -d $(SERV).def -c c -s c -y

clean:
del *.sbr *.exe *.obj *_[sc].* $(SERV).h *_c.c *_s.c

```

Moving on

When you have successfully compiled your server, you are ready to proceed to the next step: creating an environment file.

Creating an Environment file

The environment file contains an attribute specifying the location(s) of at least one Broker. Optionally, it may contain other attribute assignments associated with debugging, error logging and configuration. Upon start-up, a server scans its environment file for the location of the Broker with which the server should register.

While only one environment file is required per platform, we recommend that you create an environment file for each server, so as to afford maximum flexibility in configuring runtime applications for multiple-user networks.

Basic syntax

The basic syntax for writing an environment file is specified below:

```

DCE_BROKER=broker_host, port_num
[DCE_LOG=log_file]
[DCE_LOG_MAXSIZE=size]
[DCE_DEBUGLEVEL= normlevel, errlevel]

```

The bracketed text is optional, but highly recommended. The table below explains the italic text:

Field	Explanation
<i>broker_host</i>	The IP address or machine name of the machine on which the Broker is running.
<i>port_num</i>	The port number at which the Broker is listening. use a

	number around 10000 if you are not sure what to use. Be sure that the Broker's environment file specifies the same host and port.
<i>log_file</i>	The file where debugging output from the server should go. Use server.log if you are not sure.
<i>size</i>	The maximum size, in bytes, of. <i>log_file</i> . Default is 1000000, or 1MB.
<i>normlevel</i>	The debugging level during normal operation. Use WARNING if you are not sure.
<i>errlevel</i>	The debugging level that should be used before and during an error condition. Use DEBUG if you are not sure.

Table 6.7: Basic environment file attributes

Use only the following characters in the environment file name:

- upper case letters (A-Z)
- lower case letters (a-z)
- integers (0-9)
- underscore (_)

Environment files have a large number of optional attributes and other uses. For complete information on environment files, see "File Specifications" in *Reference*.

Moving on

Once you have an error-free environment file, you can run your server. You are ready to proceed to the next step: testing your server in a runtime situation.

Testing a Nextra Server

Before you can run a server in a production application, you need to test it extensively. The **RPCDebug** utility provides you with a GUI to test a server by executing the server's RPCs one at a time, with input arguments you specify.

At this point, you should be testing the server to verify that

- the IDL was written correctly (i.e., function declarations in the IDL file match the function declarations in the server code);
- an up-to-date server skeleton was generated; and
- the server executable was compiled properly.

Also, at this point, you should test the server from the same platform on which it is running, in order to localize difficulties to the causes just mentioned. Networking concerns should be addressed at a later stage in the debugging process, after you have verified that your server is operating properly.

The following steps illustrate how to use the **RPCDebug**.

Setting up

Follow these steps to set up the testing environment for your server:

1. Generate a Perl client stub with RPCDebug regardless of your server language.

```
> rpcmake -d file.def -c perl
```

The RPCDebug uses the Perl client stub to communicate with your server.

2. Start the Broker.

If a Broker is not already running, copy your server's environment file and change the setting of DCE_LOG attribute to a different file, such as broker.log. Then start a Broker using the new environment file.

```
> broker -e broker_env_file & [UNIX]
```

```
> start /b broker -e broker_env_file [Windows]
```

If you think you are encountering problems with the Broker, inspect the log file (as specified by DCE_LOG attribute in the environment file). It lists any Broker errors that have occurred.

3. Start the server.

```
> server_name -e server_env_file & [UNIX]
```

```
> start /b server_name -e server_env_file [Windows]
```

OR, start your Perl server as a background process (make sure your server skeleton and server code are in the current directory).

Enter:

```
> server_stub_name -e server_env_file &    [UNIX]

> start /b rpcperl server_stub_name -e server_env_file
[Windows]
```

If you think you are encountering problems with the server, inspect the log file (as specified by DCE_LOG attribute in the environment file). It lists any Broker errors that have occurred.

4. Start RPCDebug.

Copy your server environment file and change the desired log file pointer to a different file, such as debug.log. This copy of your server environment file is your client environment file. This file and the Perl client stub that you created must both reside in the current directory.

```
> rpcdebug
```

Alternately, use the text-based interface. Start it by entering:

```
> rpctest -e debug_env_file client_stub_name
```

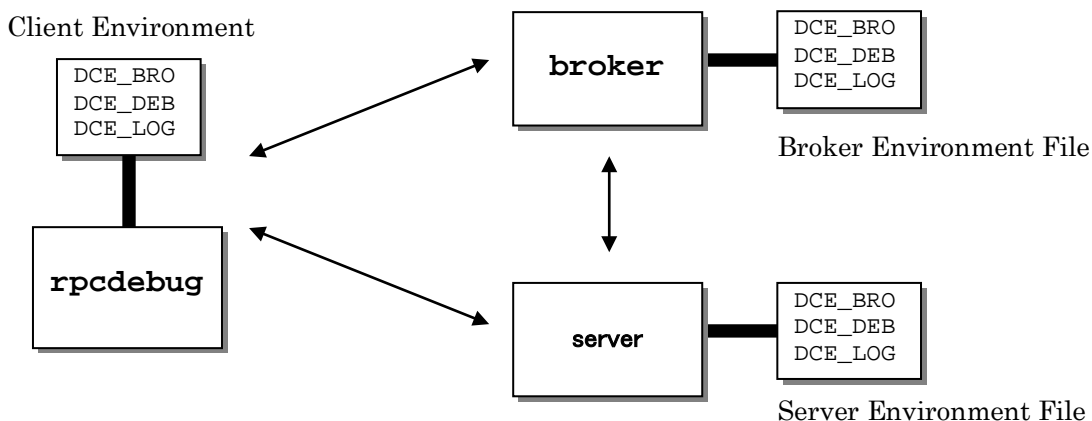


Figure 6.5: Configuration for testing a Nextra server

Testing RPCs

Using RPCDebug GUI

The window below should appear after you issue the command to bring up **RPCDebug**:

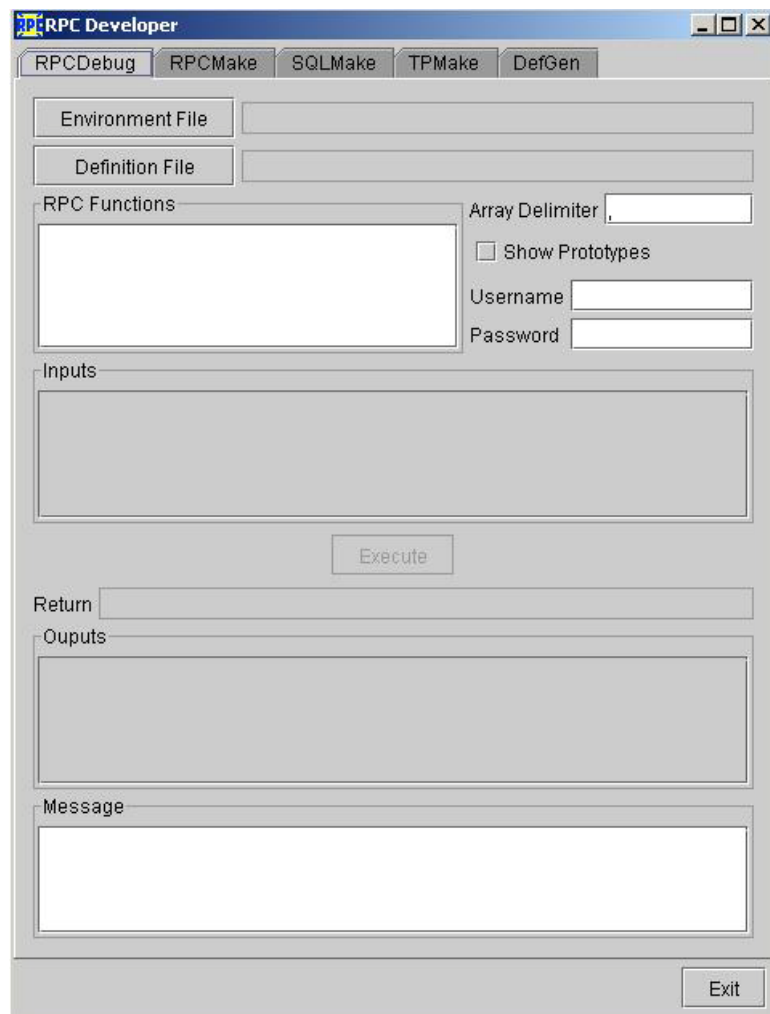


Figure 6.6: RPCDebug GUI

1. Click the “Environment File” button and then the “Definition File” button to bring up a file search window.
2. Select the IDL file (.def) on which you are basing your test client.
3. Then click the function name in the “RPC Functions” window to select a test function.

4. In the input box, enter values to send into the function.

5. Execute the function call by clicking the “Execute” button.

The output variables and their values appear in the “Outputs” box. Inspect the m to verify that they are the values you expect.

6. Test another function.

Click on a different function in the “RPC Functions” box and then enter new input values. Click the “Execute” button again.

7. Exit the utility.

Using RPCtest command line interface

After you start the command line RPCtest utility, the terminal should response with the following prompt:

```
rpctest>
```

At the `rpctest>` prompt, you can execute RPCs and view the result according to standard perl syntax.

Execute function calls using the following syntax:

```
rpctest> &func_name(in_arg1, in_arg2, ..., in_argN,
  *out_arg1, *out_arg2, ..., *out_argN)
```

If you are unsure about the parameters for a function, inspect the function definition in the IDL file. Each *in_arg* corresponds to an [in] parameter, which each *out_arg* corresponds to an [out] parameter. This correspondence is by position only - *in_arg1* is the first input parameter listed in the IDL file, while *out_arg1* is the first output parameter listed. The parameter names in your function call do not have to match the names in your IDL file or your server code – only position is significant.

To print out the results of the last function call, type `show`. To exit the RPCtest shell, type `exit`.

Refer to “rpctest” in *Reference* for further details on executing function calls.

Modifying servers

In a function returns unexpected values, you need to inspect your code. Quite from the testing utility, fix the code and try again.

To save time, keep in mind that as long as the IDL file for a server does not change, the server skeleton does not have to be re-generated.

Therefore, if you modify server code, *do not re-generate the server skeleton unless you change the function name, the return value type, the number of parameters, their data types, or their order*. If you change any of these items, you must also change the IDL file.

In Perl, simply bring down the old server and start the new one. The server skeleton automatically includes the new server code (assuming that you did not change the name of the server code file.).。

Conclusion: implement the server

After the server has been fully tested, move the server program and its associated environment file to the platform where the server should run.

Chapter 7 DB access server

This chapter introduces the capabilities of the DB access server and summarizes the development process for creating DB access server. Step-by—step details for developing servers are laid out in the following chapter.

You should know how to code DB access routines using ANSI-SQL.

The DB access server

The DB access server facilitates the construction of servers that access and manipulate RDBMSs. You need only code SQL statements in a Nextra-compliant format; the DB access tools produce or provide all remaining DB server functionality.

The DB access server can be used in conjunction with the following RDBMSs: Oracle, MS SQL Server, DB/2 and HiRDB.

Components

The DB access server consists of:

- SQLMake
- dbcommon.h
- a RDBMS –specific server start-up utility

SQLMake generates an IDL automatically, based up an input file consisting of SQL statements written in a Nextra-compliant format., and then does client.

The head file dbcommon.h contains definitions of all the error codes tha can be returned by a DB server.

All functionality related to RDBMS-interfacing has been centralized to minimize the amount of code tha you have to write. The RDBMS-specify start-up utility containing this functionality is used to execute all RDBMS servers developed with **SQLMake**. Depending upon your RDBMSs , your start-up utility is one of the following: **ora_startxx** (**xx** for Oracle version), **sql_start** (SQL Server), **db2_start** (DB2) and **hirdb_start** (HIRDB). This book refers to it as **DB_start** .

Use

Figure 7.1 shows the usage model for the DB access server. To implement three-tiered connectivity with RDBMSs using the DB access server, you write standard SQL statements (modified slightly to accept variable input arguments), and feed this code into **SQLMake**, generates an IDL and call **RPCMake** to generate the client stub(s). You can then launch DB access server by starting up an instance of the RDBMS-specific start-up utility, and passing it the name of the file containing the modified SQL statements.

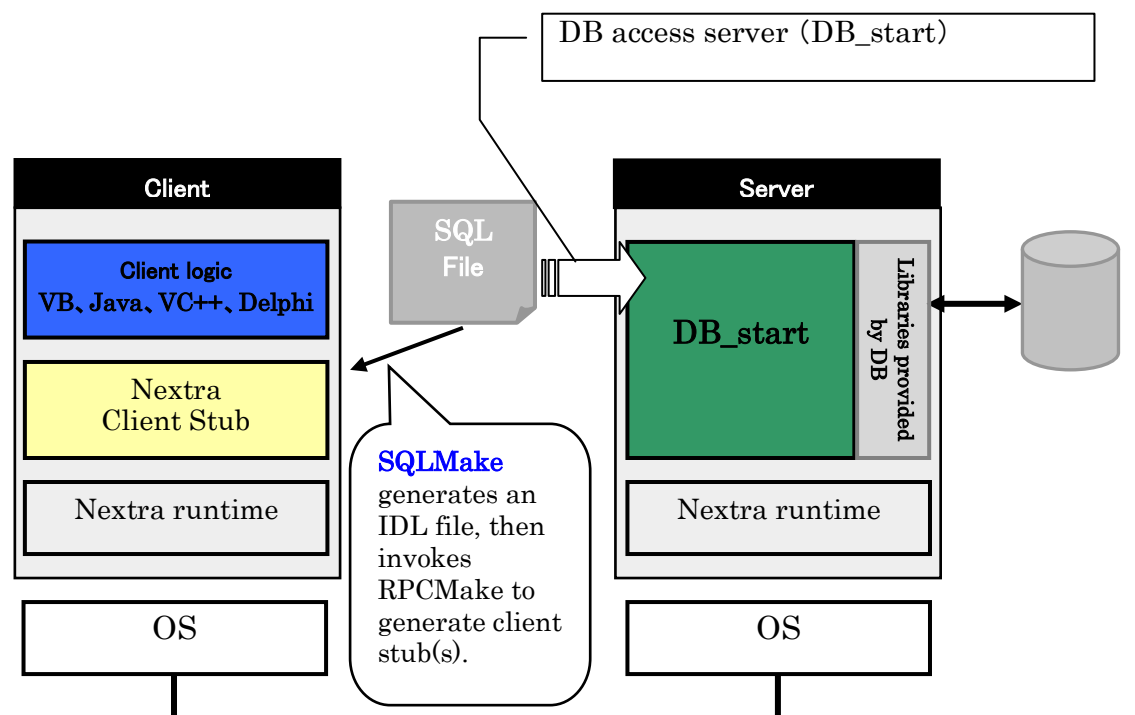


Figure 7.1: Use of the DB access server

Before You Start

Platform requirements

Nextra development package or runtime package must be installed on every machine where DB access server will be running.

In addition, if the DB access server will be running on a different machine than the RDBMS it accesses, RDBMS-specific networking

software must be installed on both platforms, to allow the DB access server to connect with the DB.

Setting the environment variables

Before using any Nextra tools, you need to set several environment variables on every platform on which a DB access server will be developed or implemented. See “Setting the environment variables” for details on these variables.

RDBMS variables

For all RDBMSs, ensure that the PATH variable includes the directory containing the RDBMS executables (usually *home_directory/bin*).

To set the number of decimal places in data returned from a floating point column, use the PRECISION environment variable. Set the value to the number of places you want to appear to the right of the decimal point. If you do not set PRECISION, or if you set it to zero (0) or a negative value, the default is six (6) decimal places.

You may specify exponential notation for all values by appending an ‘e’ to the PRECISION value. You may specify exponential notation for only very large and very small values by appending a ‘g’ to the PREVISION value.

PRECISION=5	PRECISION=4	PRECISION=4e	PRECISION=4g
7648.65431	7648.6543	7.6486e3	7648.6543
4.12457	4.1246	4.1246e0	4.1246
675391.00000	675391.0000	6.7539e5	6.7539e5
.02100	.0210	2.1000e-2	.0210

Table 7.1: Example PRECISION variable values

RDBMS-specific variables

For SQL Server and Oracle:

- Set DB_LOGIN to the name fo the DB user.
- Set DB_PWD to the DB user's password.

Note that you will need to run `sql_start` and `ora_start` in different sessions if you wish to use different user-names and passwords.

For Oracle only, you may specify other combinations of `DB_PWD` and `DB_LOGIN`. For example, you can use `DB_LOGIN` to specify both the user name and password, using a slash (/) as a separator:

```
DB_LOGIN=user_name/password
DB_PWD=
```

or, if you have Oracle*NET configured and you would like to access remote db:

```
DB_LOGIN=user_name/password@database_alias
DB_PWD=
```

In addition, Table 7.2 shows which RDBMS-specific variables must be set:

RDBMS	RDBMS-specific variables
SQL Server	DSQUERY= <i>name_of_remote_SQL_server</i> includes <i>SQL_Server_directoy</i> \DLL to the PATH.
DB2	DB2INSTANCE= <i>user_name_of_RDBMS_owner</i>
Oracle	ORACLE_SID= <i>database</i> ORACLE_HOME= <i>dir</i>

Table 7.2: RDBMS-specify values

Environment variables for dedicated server

For Oracle, the environment variable `DB_NOCOMMIT` must be set in order to use a dedicated server.

If `DB_NOCIMMIT` is set to 1, the server will not perform implicit commits on non-select SQL statements. If it is set to 0, the default, the `DB_start` server will perform implicit commits.

For more information on dedicated server, see `DB_start` documentation in *Reference*.

Runtime overview

A database session begins when the RDBMS-specific start-up utility is executed with at least the required command line options:

```
> ora_start -q query_file -d db_name -e env_file [-s interface]
```

Flag	Description
-q <i>query_file</i>	The name of the query file used as input to SQLMake to generate the IDL file.
-d <i>db_name</i>	The name of the DB the server will access.
-e <i>env_file</i>	The name of the environment file.
-s <i>interface</i>	The name of the server/interface name. If you do not use this option, then you must specify it in either SQL statement file or to DCE_SERVERNAME attribute in the environment file.

Table 7.3: Required *DB_start* options

See *Reference* for a full description of all possible options.

When the start command is executed,

- First, the utility reads the environment file. If DCE_SERVERPORT attribute in the environment file, then the utility allows the OS to assign it an available port.
- Next, the utility parses the query file.
- Then, if it is not a dedicated server, the start-up utility initializes a login to the RDBMS.
- Finally, the utility registers with the Broker as a server, and waits for client requests.

Each time a client makes a function call to that server, the server returns a specific value that indicates the status of the attempt. The value will be a positive or negative integer, where:

Value	Meaning
≥ 0	Successfully executed. The return value is the number of rows successfully fetched, inserted or updated.
$-1000000 < \text{errnum} < -1$	RDBMS internal error code. For the error code, refer to the RDBMS-specific manual.

<= -1000000	None RDBMS-specific error code. Use <code>dce_errnum()</code> or <code>dce_errstr()</code> Nextra API in the client logic to check the error.
-------------	---

The login session to the DB ends when the DB access server process is killed. When this happens, the server closes its connection and exits.

SQL supported

The DB access server supports all ANSI SQL statements, including the invocation of stored procedures (where applicable), with one exception. ROLLBACK is not currently supported by the utility, because the DB access server performs an implicit COMMIT after each function call.

Data types supported

The supported SQL '92 Entry Level data types are: character, integer, small integer, float, real and double. In addition to these data types, the Advanced level Binary large Object (BLOB) type is supported for all RDBMSs. Binary is used to indicate BLOB data. The supported data types are mapped onto their corresponding IDL data types by the client stub and onto their RDBMS-specific data types by the *DB_start*. Decimal and numeric data types are not supported, because they have no corresponding IDL types. All unsupported data types are treated as strings.

Development Process

This section summarizes the steps necessary to implement a DB access server in the Nextra environment.

Building the server – process overview

Perform the following steps for each DB access server.

1. Write the ANSI SQL statements that will be converted into remote procedures.

	Debug
--	--------------



Test these statements for proper SQL syntax by piping the text into the RDBMS and redirecting the results to a file. Read the output file to see if each statement had the intended effect.

2. Translate the SQL statements into a Nextra-compliant format.
3. If generating COBOL stubs, add COOBL size information.
4. Generate a IDL file and client stubs, using SQLMake utility.

No server skeleton should be generated since the RDBMS-specific start-up utility already incorporates all server stub functionality.

5. Create or edit the environment file.

At this point, the DB access server can be launched.



Debug

Test the new server before implementing it in a runtime distributed application.

After the DB access server has been tested, move the environment file to the platform where the DB access server will run.

Chapter 8 Building DB access server

This chapter covers the specific process for building and debugging DB access server. For an outline of the process, see “Development Process”.

Because this chapter refers to concepts introduced earlier, we recommend that you read the previous chapters of this manual before you read this chapter or try to build a DB access server. You should know how to code DB access routines using NSI-SQL.

Building a DB access server

Begin by assembling a list of the ANSI SQL statements whose functionality shall be implemented in the server. Although Nextra tools make the process of incorporating new SQL statements into a server very efficient, this list should be exhaustive.

Writing SQL statements

Use an editor to enter SQL statements, according to RDBMS-specific syntax, in a file. A sample Oracle file might look like this:

```
select
    base_cost,
    indust_strg
from
    tools
where
    tool_name= 'WonderTool' ;

insert
    into tools
values
    ( 'WonderTool' ,500,8) ;

delete from
    tools
where
    tool_name = 'WonderTool' ;
```

The statements need not be in any particular order. Only one statement per semicolon delimiter is allowed. Several RDBMSs use different

delimiters; check our documentation for details about your particular RDBMS.

Testing the statements locally


When all the SQL statements have been entered into the file, test the syntax by piping the file into the DB manager through the RDBMS-specific interactive access utility:

```
> cat test_file | DB_access_method > resultfile
```

where:

<i>test_file</i>	is the name of the file containing your SQL statements
<i>DB_access_method</i>	is a string that invokes the RDBMS-specific interactive access program (i.e. sqlplus for Oracle). Pass in the arguments – username, password and/or DB name - as required
<i>resultfile</i>	is the name of a file that will receive the output of the SQL statements after the DB manger executes them.

Table 8.1: Statement file testing syntax

	<p><u>Do not skip this step!</u></p> <p>If your SQL statements are not valid for input using your interactive SQL utility, they will not work as server code, either. You must test your SQL statements against the database before continuing.</p>
---	--

NOTE: The exact syntax for your DB and OS may vary. The idea is to pipe the statements into the DB and then redirect the output to a file, printer, or screen. You may execute this plan in whatever manner is preferred.

Another option is to make the statement file executable (chmod +x *filename* in the UNIX environment), and add a command at the beginning to invoke the interactive SQL interpreter. The following examples how:

[Oracle]

```
pathname/sqlplus user/passwd <
statements
!
```

[MS SQL Server]

Create a file containing the following liens:

```
use db_name
go
statement
go

...
```

Then run the interactive SQL utility as follows:

```
pathname%i sql - U user -P password -S server <<filename>
!
```

Using this method, you will want to send the output directly to a file, so you would run your statement file this way:

```
test_file > resultfile
```

In either case, when the test run is finished, inspect the output file for error messages, and verify that the data returned is what you expected. If you get any errors or spurious data, edit the statements and run the test again. When you are satisfied by the contents of the output file, you may proceed to the next step in the development process.

Modifying SQL statements

After you test the syntax for each SQL statement, the next step is to convert each statement into Nextra-compliant format. The following steps are required:

- 1. Prefix each statement with a function label (and optional function attributes).**

Function labels are required in order to enable server code to call each query by name. Function labels use the format

name:

where *name* is a function name unique among all the function names available in an application.

2. Substitute descriptive argument variables for literal strings.

Substitute variable by inserting a variable-name, flagged by a preceding dollar sign "\$", in place of literal values. If you are not using data type attributes, quotation marks are required when substituting variables from string literals. For character fields, simply substitute the variable name inside the quotes. For numeric fields, quotes are not needed.

3. Insert an attribute string indicating each variable's data type.

All SQL '92 Entry Level data types are supported, with the exception of numeric and decimal types. These two types have no IDEL equivalents, so they are mapped onto string by the SQLMake. If no data type is specified for a variable, it is assumed to be a string (char). The valid data types and their corresponding case-insensitive attribute strings are:

Datatype	Attribute String
character	char, character
integer	int, integer
small integer	smallint, small int, small integer
real	real
float	float
double	double
binary	bin, binary, BLOB

To indicate the data type, use one of these formats:

Input	Output
<i>input</i> [<i>datatype</i>]	[<i>datatype</i>] <i>output</i>

where *input* and *output* are the variable names, and *datatype* is one of the supported data types in the list above.

For more details about attribute clauses and their format, see “Adding attribute clauses to variables” in *Reference*.

4. (Optional) Add and adhoc function, if you wish to enter SQL statements dynamically during runtime.

Add a function tag beginning with the characters “adhoc”, containing a blank statement. This allows you to type in an SQL statement during runtime and submit it to the DB. Remember that the function name must still be unique, so if your application contains two DB access servers, each containing an adhoc function, you must name them differently (e.g., “adhoc1” and “adhoc2”).

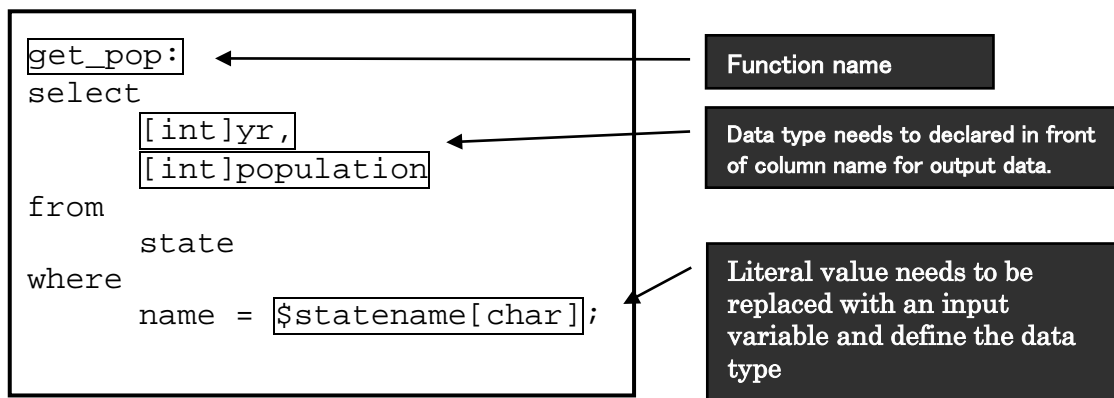


Figure 8.1: An example of function label, variable and data types

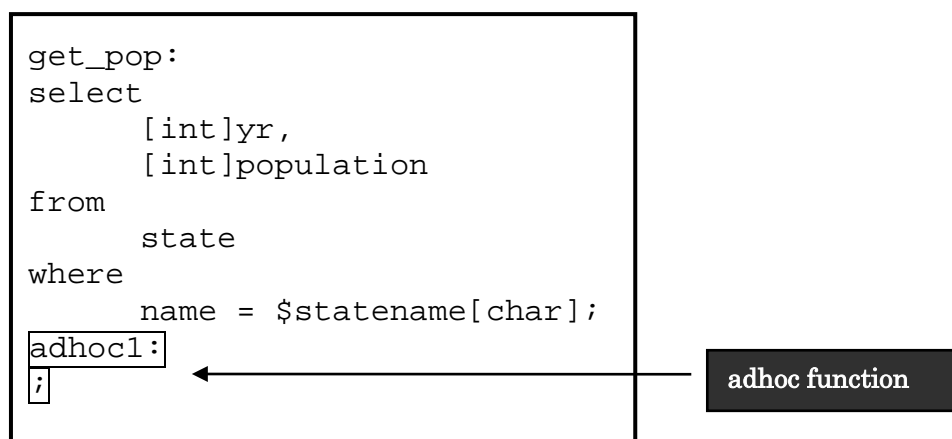



Figure 8.2: The adhoc line in a statement file

After all SQL statements have been converted into Nexera-compliant format, your statement file is syntactically complete. If several different servers will be operating, server files should be named to distinguish

them by their contents. A good convention is *database_name.sql* or *server_name.sql* for SQL statement files.

Possible function attributes are described in detail in the “File Specifications” chapter in *Reference*.

	<p>Warning</p> <p>Each statement file may access no more than one DB. This means, for example, if you need to access two different DBs under a single RDBMS, you need to write two statement files.</p>
---	--

Inserting COBOL size information

If your DB access server will be accessed by a COBOL client, the size attributes for all null-terminated array variables must be given in the SQL statement file.

Every variable used in a function must have a corresponding line in the statement file to specify its size. This line should have the format:

```
#COBOL [ output type] variable  size1 [size2]  
#COBOL variable [input type]  size1 [size2]
```

where *variable* is the name of the variable, and either *output type* or *input type* is included, depending on the kind of variable. These indicate the variable's datatype. *size1* and *size2* are integers. Variables of type binary (BLOBs) cannot be used with COBOL, because they cannot be used with static allocation. Because simple data types (int, float, etc.) have a fixed size, you need to set only *size1*, which sets the maximum number of values the array can store. For strings, both values need to be set. *size1* specifies how many characters can be in each string, and *size2* specifies how many strings the variable can hold. The default value for *size2* is 200. In general, *size1* should be the same as the size of the data column whose values are stored in the variable, and *size2* should be the maximum number of rows you expect to be returned.

For example, here is the `get_pop` query from Figure 8.1, with appropriate COBOL additions.

```
get_pop:
```

```

#cobol yr 10
#cobol population 10
#cobol statename 11
select
  [int]yr,
  [int]population
from
  state
where
  name = $statename[char];

```

yr and population are arrays of up to 10 integers, and statename is an array of up to 10 strings, each of which can be up to 11 characters in length. 1 byte extra is required for NULL-fermentation.

Generating stubs and IDL files

Variable naming

While parsing a select statement in a SQL statement file to generate an IDL file, the **SQLMake** tries to use the most intuitive function argument name for each select-list item in the statement. It will concatenate all the valid string tokens in the expression, insert an underscore in between them, and use that as a name. If this conflicts with a previous select-list expression, the utility will append the string `_xxx` to it, where `xxx` is a numerical string corresponding to the item's place in the select statement. If an alias is specified for the element, the utility will attempt to use that first.

DB access server does not require server skeleton.

Figure 8.3 shows the layout of the **SQLMake** GUI.

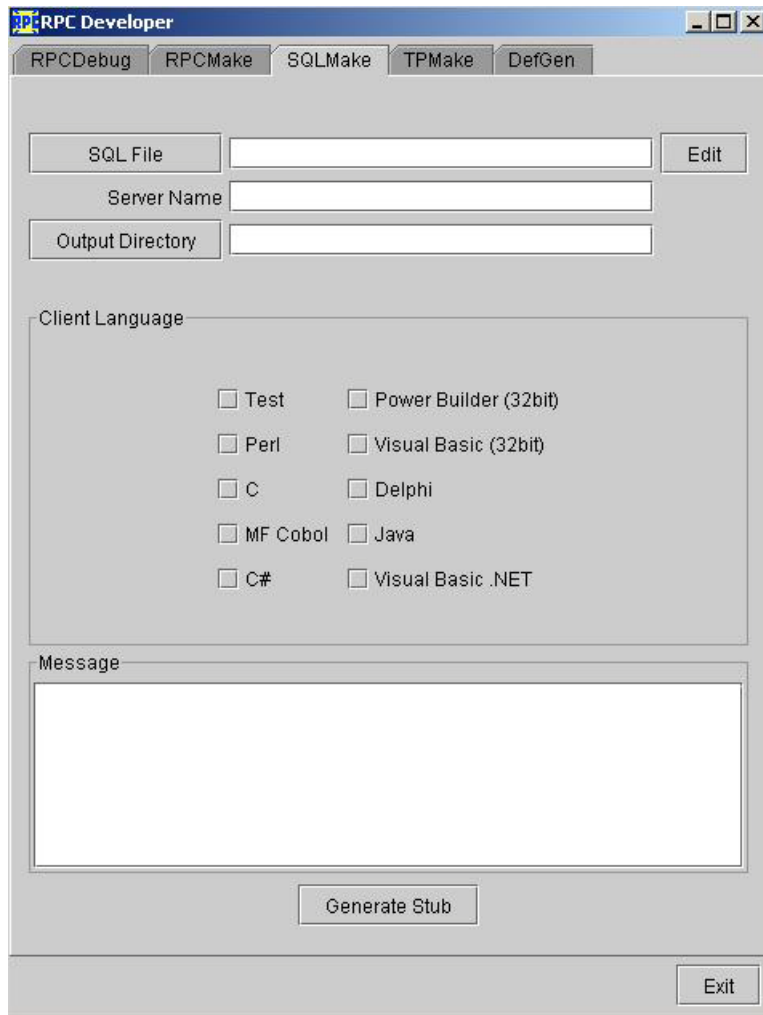


Figure 8.3: SQLMake main screen

To generate an IDL file and one or more client stubs, follow these steps:

1. In the “SQL File” field, enter the name of your SQL file name including the directory full path.
2. In the “Server Name” field, enter the name that should be specified in the interface statement in the generated IDL file.

This field also determines the prefix of the generated IDL file name. If you wish to enter a variable server name, simple precede the name with a dollar sign “\$”. Quotes are not required in the GUI field.

3. Click on one or more buttons under “Client Language” to choose the client stub language(s).

Building a client stub is not necessary at this point, but if a client stub is not built now it will have to be built later.

4. Click on the “Generate Stub” button to generate the files.

5. Click the “Exit” button to exit.

Interface options

Clicking on the “SQL File” button brings up a “File Filter” dialog box that lets you search the file tree for a filename or a filename pattern.

Clicking on the “Edit” button brings up a screen editor that allows you to edit text files without leaving the **SQLMake**.

Output files

As shown in Figure 8.4, the **SQLMake** generates an IDL file and whatever client stubs you specify.

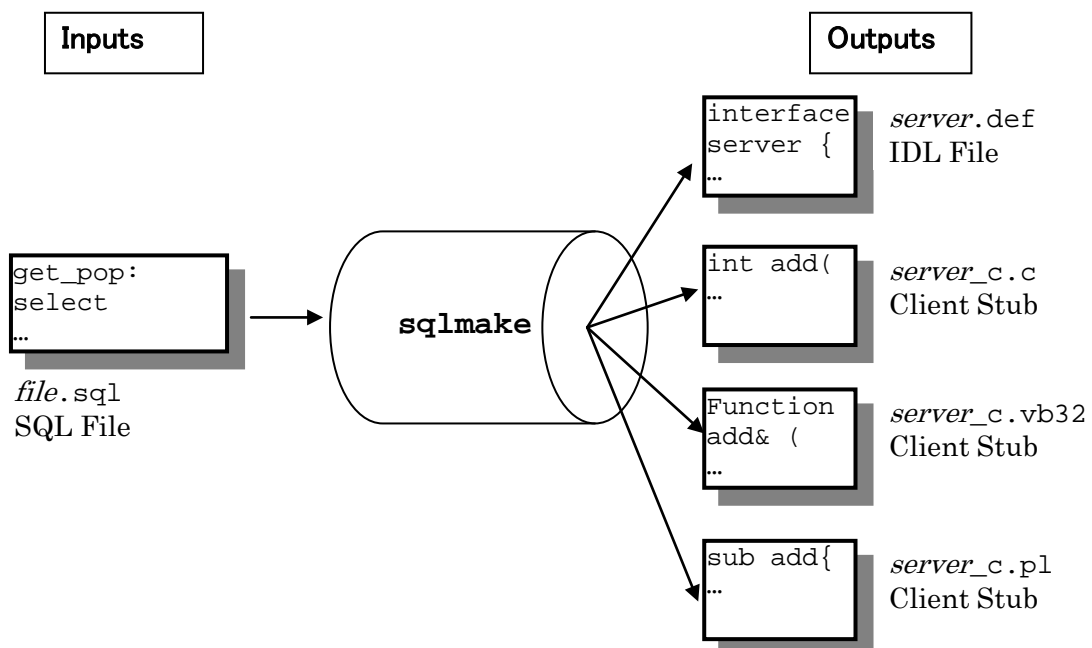


Figure 8.4: Files associated with SQLMake

Invoking SQLMake

To generate the stubs and IDL from the command line, use the following syntax:

```
> sqlmake [-q query_file] [-s ['$]server_name[']] [-c lang1] [-c lang2 ...] [-y] [-h | -help] [-v]
```

where:

<i>query_file</i>	is the path name for the Nextra-compliant SQL statement file.
<i>server_name</i>	is the server name that will be specified in the interface statement in the IDL file(must be the same as the name provided to a DB_start using the -s flag on server start-up). If preceded by a dollar sign(\$) and surrounded by single quotes, a variable server-name may be specified. You may specify multiple client stub languages at once. These flags are optional. For more information, see "RPCMake" in <i>Reference</i> .
<i>lang</i>	in an abbreviation for the language in which a client stub should be generated. SQLMake invokes RPCMake to generate this stub, passing the -c flag, the language, and the generated IDL file in to the RPCMake utility. You may specify multiple client stub languages at once. These flags are optional. For more information, see "RPCMake" in <i>Reference</i> .
-y	automatically overwrite any previously existing file(s) for the same serve by the same name.
-h, -help	causes the utility to provide a brief usage message and exit.
-v	causes the utility to provide its version number and exit.

Table 8.2: SQLMake syntax

SQLMake names the IDL file as *server_name.def*. Client stubs are named *server_name_c.lang*, *lang* is an abbreviation for the selected language.

Creating an environment file

Create or edit the environment file according to the details set down in "Creating an Environment File".

Testing a DB access server

Test your new DB access server before implementing it in a runtime distributed application. Use the debugging methods described in "Testing a Nextra Server".

1. Follow the directions in "Testing a Nextra Server" to generate a Perl client stub and start the Broker.
2. Start your DB access server.

Enter:

```
> DB_start -q query_file -d db_name -e env_file -s server_name
```

See *Reference* for a complete discussion of DB access server start-up options.

3. Follow the directions in "Testing a Nextra Server" to start the RPCDebug.

If you have included an adhoc function, select it, and then enter a complete ANSI-SQL statement as input. Click the "Execute" button and inspect the results. (Since adhoc sends your statement directly to the RDBMS, and error indicates that your input statement was incorrect.)

Distributing the server code

Move the SQL statement file to the machine upon which the DB access server will run. If a client stub was generated at this time, move the client stub to the platform where the client interface will be located. If no client stub was generated, refer to *Client Developer's Guide* for the client building process.

Using Special Features

This section converts how to use four additional features:

- non-default DB access
- dedicated server accessing multiple DBs

- stored procedure
- DB cursors (with dedicated server only)

Non-default DB access

DB_start can connect to any DB running under the appropriate RDBMS, not just the default DB (which may be specified in an environment variable).

There are no restrictions on which DB you can access with each **DB_start**. For instance, you can access an Oracle DB not specified as the default. The “default” for each DB engine is specified as follows:

Oracle	specify default DB in the ORACLE_SID environment variable.
MS SQL Server	default is the local SQL Server , if one is running. If the DSQUERY environment variable is set, it is used.
DB2	DB2INSTANCE environment variable for the default DB.

You need to set up separate instances to access different DBs.

Consult the following for details specific to each RDBMS.

Oracle

IF you want to connect to any DB other than the default DB, then all machines involved (even if there is only one) must be running Oracle*Net. Also, the name of the DB and its Oracle*Net alias, as specified in the tnsnames.ora file, does not need to be the same.

For example, suppose the default DB is specified to be *massachusetts*:

```
ORACLE_SID=massachusetts
```

To start up your DB access server with the default DB, you simply type

```
> ora_start -e data.env -s oraserver -d massachusetts -q query_file
```

Or, start up using another DB, type the following:

```
> ora_start -e data.env -s oraserver -d alias -q query_file
```

`ora_start` looks in `tnsnames.ora` file, sees the alias *alias*, and opens the DB specified for the alias.

SQL Server on Windows

You must set the environment variable `DSQUERY` if you wish to access any SQL Server except for the default (usually local) server.

This SQL Server must be defined using the SQL client configuration utility, which can be found in the SQL Server program group. Follow these steps in the SQL Client Configuration dialog box:

1. Click the “Advanced” button.

The “Advanced” dialog box pops up.

2. In the first field, enter an alias name to identify the server.

This can be any name unique among the defined SQL server.

3. In the second field, the DLL should always be `dbmsocn`.

4. In the third field, enter the hostname and port of the SQL server you wish to associate with the alias name you have specified.

The proper syntax is *hostname, portnum*.

5. Save your changes and exit the utility.

When you start a DB access server, you must specify a DB (using the `-d` option) that is available through the SQL Server specified in the `DSQUERY` environment variable.

For example, suppose you want to start up a server that accesses `bigdb`, a DB running under the default SQL Server:

```
> sql_start -e data.env -s server_name -d bigdb -q query_file
```

To access a DB under a different SQL Server, you must set `DSQUERY` to the name of that server, and then specify a similar start-up command:

```
> sql_start -e data.env -s server_name -d tech -q query_file
```

Accessing different databases with each dedicated child server

DB_start dedicated servers can connect to a different DB with each dedicated instance of the server.

If you are running a dedicated server, each client can specify what DB it wants to connect to in its `sql_prepare_interface()` RPC. The following statements come from different clients connecting to the same dedicated server called `dbserver`, requesting dedicated instances which connect to different databases.

```
sql_prepare_dbserver("purpleddb", "beth", "v98S34tJ");
sql_prepare_dbserver("reddb", "marco", "eWr8uin348");
```

Using stored procedure

Nextra tools support the calling of stored procedures for Oracle. However, no output can be returned from a stored procedure.

A function that calls a stored procedure must be defined in the SQL statement file, just like any other function that the DB access server will execute. However, stored procedures cannot be used with data type attribute clauses.

To define a stored procedure as an RPC function, you must enclose the call you would normally make to the RDBMS in braces, flag input variables with dollar signs, and declare the output variables immediately after the call.

For example, suppose you defined the following stored procedure in Oracle:

```
create procedure listname (person_age in integer, in
raise real) as
  BEGIN
    update namelist
    set salary=salary*raise where age >
    person_age;
  END listname;
```

You can call this procedure in your SQL statement file in the following manner:

```
update_namelist:
```

```
{BEGIN listname($a, $b)\; END\; \;};
```

See also “SQL Statement Files” in *Reference*.

Using cursors with dedicated server

Dedicated server allows the use of DB cursors. If a **DB_start** utility has been started with the environment file attribute `DCE_DEDICATED` set greater than 0, it creates a copy of itself for each client that connects with it, so that each server executes RPCs for only one client. In contrast to regular servers, this kind of server can maintain a “state”; that is, it can keep track of the client’s request and tailor responses appropriately. Using cursors with DB access server is one way to exploit this feature.

Cursors and Advanced DB access server

IDL files generated by **SQLMake** include two functions for us with dedicated DB access server: `sql_prepare_<interface>()` and `sql_set_max_rows_<interface>()`. These functions are for us only with dedicated DB access server and should never be called in regular DB access server. The function names are appended by the interface name to distinguish between similar functions for different servers in situations where a client accesses more than one dedicated DB access server.

Starting a session with dedicated DB access server

Dedicated DB access server can login to a DB two different ways: implicitly (default) or explicitly (override).

The default DB login and password are provided in the environment variables `DB_LOGIN` and `DB_PWD`. If `sql_prepare_interface()` is not the first RPC received by the dedicated server, then it initializes a login to the DB just like a regular server – by using the `DB_LOGIN` and `DB_PWD` environment variables in combination with the DB name specified with the `-d` option on the command line. If the environment variables are not set, then the login and password used are `NULL`. If the login fails, then the dedicated server dies without connecting to the DB.

You may override these defaults by calling `sql_preapre_<interface>()` as your first RPC to this server. The function takes three arguments: DB to access, user name, and password. Its C prototype follows:

```
long sql_prepare_<interface> (char * dbname, char * login,
char * passwd)
```

Upon receiving this function call, the server initializes a login to the specified DB using the specified name and password. If the function is successful, any further calls to this function return a Nextra error (DBALREADYCONN: function called twice). If the first call fails, the server will only accept another call to this function. It continues to accept no other RPCs (returns error code DBUSELOGINRPC: sql_preapre failed) until the sql_prepare_<interfce> () function is successful, indicated by a return value of DCPSUCCESS.

See *Reference* for more information about this function.

Setting the cursor value

Next to make use of the cursors feature, the client must call the function sql_set_max_rows<interface> ().

This function takes one argument: the maximum number of rows to retrieve per query execution. It will return the value DSPSUCCESS if *maxrows* is zero or a positive number and the function succeeds. If *maxrows* is invalid (negative), the function returns DBINVALIDPARAM. If the function fails for another reason, it returns the appropriate error code.

For example, if a query selects 500 rows total, and sql_set_max_rows_<interface> () has been called with the value 50, only the first 50 rows are returned the first time the query is made, and the rest of the rows are cached in the dedicated server child process. With each subsequent execution of the same query, the next set of 50 rows is returned. When all rows have been returned, the cache is refreshed.

See *Reference* for more information about this function.



Why only dedicated server?

Cursors will not work with regular servers, because nothing prevents another client from making the same query to the same server, which will set the cursor 50 rows too far ahead for the next time the original client

makes the query. Or, another client might be making a different query, setting the cursor back to zero before the first client finished retrieving the full selection. Since a dedicated server serves only one client, there is no possibility of another client disturbing the query series.

If `sql_set_max_rows_<interface> ()` is never invoked, or if `sql_set_max_rows_<interface> (0)` is invoked, the server returns all selected rows at the first query. Similarly, if `maxrows` is larger than the number of selected rows, all the selected rows are returned with the first query. If another query is called before the first query has returned all selected rows, the first `maxrows` rows of the second query are retrieved, and the cursor for the first query is reset to the beginning. To change the number of rows being returned at a time, you need only invoke `sql_set_max_rows_<interface> ()` with a new value.

For example, assume we have two RPCs, `rpc1` and `rpc2`. Both return 20 rows total, and `sql_set_max_rows_<interface> (4)` has been called. The first call to `rpc1` returns rows A1-A4. The next call to `rpc1` returns rows A5-A6. A call to `rpc2` returns rows B1-B4. A following call to `rpc1` returns rows A1-A4. If `sql_set_max_rows_<interface> (8)` is called, and `rpc1` is called once again, the rows A5-A12 are returned.

Note: if `rpc1` is called 6 times in a row with `sql_set_max_rows_<interface> (4)`, the 5th call returns rows 17-20, while the 6th call returns 0 rows. A 7th call refreshes the dedicated server child process cache, and returns rows 1-4.

If there were only 19 total rows, then the 5th call would return rows 17-19, and a 6th call would return rows 1-4.

If the last in a series of queries returns exactly enough rows to complete the full selection, the next execution of that query will return 0 rows. This behavior is desired because without it, situations can arise in which the number of rows returned does not necessarily indicate when the full selection has been returned. With this behavior: If a query returns fewer than `maxrows` rows, then the full selection has been returned on that query.

Example:

```
#include dceinc.h
/* This example is for regular data access servers */
.
```

```

..
...
main
{
char * db, * login, * passwd;
int rv;

get_info_from_user(db, login, passwd);
rv = sql_prepare_interface(db, login, passwd);
check_for_error(rv);
rv = sql_set_max_rows_interface(100);
...

..

.

```

Cursors and Variable named DB access server

For variable named interfaces, stubs automatically contain an extra parameter in all remote function calls. This parameter is always the first parameter in the parameter list, and it specifies the interface to which the function call should be sent.

For variable named server, interface is the variable name specified in the IDL file, with the leading '\$' stripped off. For example, if you generate a IDL file with **SQLMake** using the flag `-s '$var_name'`, the cursors functions are called `sql_prepare_var_name ()` and `sql_set_max_rows_var_name ()`.

See Also:

“Dedicated server: Overview” in *Configuration Guide*.

“`sql_set_max_rows_<interface> ()`” and “`sql_prepare_<interface> ()`” at “Nextra API” chapter in *Reference*.

Chapter 9 Building Java application server

This chapter covers the specific process for building and debugging Java application server.

We recommend you to read [“Chapter6 Building Functionality Servers”](#) before proceeding.

For the client application development, please refer to *Client Developer’s Guide*.

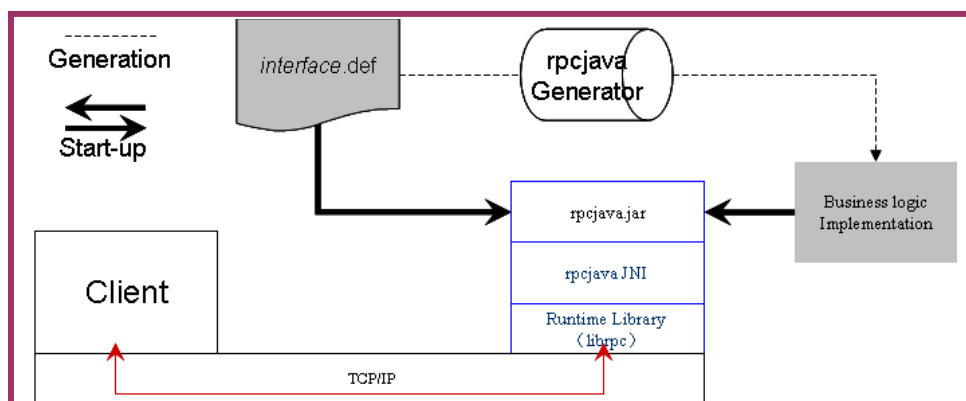
Characteristics of Java application server

Developers are able to build and deploy distributed Java application server by only implementing business logic based on the Java interface class generated by a Nextra generator.

Features of this ultra light distributed Java application server are:

1. No EJB container required. Thus, it is light weighted and more optimized than EJB.
2. Materialize application clustering without having any additional softwares and hardwares by deploying redundant Java application server processes on multiple server machines.
3. Fast data transpirations with its client processes.
4. It can communicate not only with Java client language, but also other client languages such as .NET, VB, C, COBOL, Delphi and PB.

Architecture




Part  needs to be programmed by the developers.
Other than that, Nextra provides or will generate for you.

Figure 9.1: Java application server architecture

Requirements

Platform requirements

- Nextra runtime library (librpc)
- JDK1.3.1 or above

Restrictions

Dedicated child processes of the Dedicated Server (specify DCE_DEDICATED attribute in the environment file) are only able to inherit classpath.

Development process

We need to go through seven steps to develop Java application server.

1. [Set up the environment](#)
2. [Write IDL file](#)
3. [Generate Java interface class with Java application server \(rpcjava\)](#)
4. [Implement the Java interface class](#)
5. [Prepare the environment file](#)
6. [Start up rpcjava](#)
7. [Unit test with RPCDebug](#)

Set up the environment

We assume you already did what described in 'Environment set-up after installation' of "Chapter2 Installation" in *Read Me First*.

Windows

Source the directory where we have jvm.dll to the PATH environment variable.

UNIX

Source \$ODEDIR/lib and the directory where we have `jvm.[txt]` to the Shared Library PATH environment variable.

Write IDL file

Open a text editor and write an IDL (.def) file. Please refer to 'Interface Definition Language (IDL)' of "Chapter 2 File Specifications" in *Reference* for more details.

Generate Java interface class with Java application server (rpcjava)

In order to implement the business logic, you need to generate a Java interface class first with either Nextra Development Tool or command line. `interfacename.java` is the generated file name.

Using Nextra Development Tool

>rpcmake

On the GUI, you select an IDL (.def) file and click "Java" for the Server language, then click "Generate" button.

Using command line

>rpcjava -h for HELP

>rpcjava -f *file.def* [-d *directory*] [-c *classname*] [vm options ...]

Implement the Java interface class

Please implement the `interfacename.java`. You shall also refer to "[Reminder over implementation](#)".

Prepare the environment file

Please refer to 'Environment Files' of "Chapter 2 File Specifications" in *Reference* for more details.

Start up rpcjava

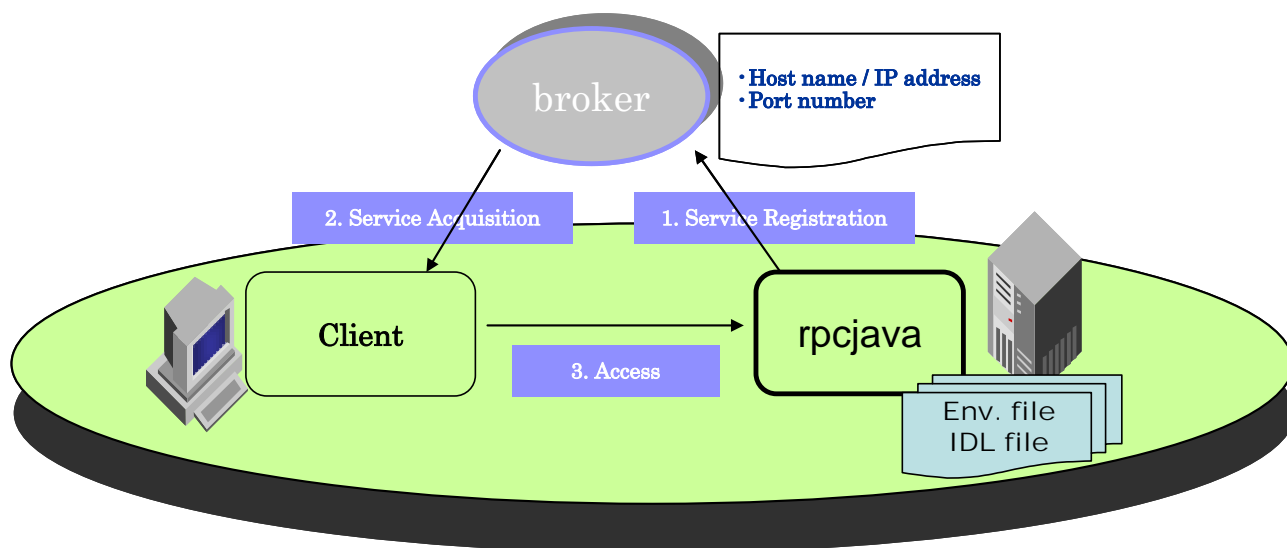


Figure 9.2: rpcjava startup image

First off, start broker as following:

```
>broker -e env_file
```

* Please refer to 'broker' of "Chapter 4 Nextera utilities" in *Reference* for more broker's options.

Then, we start up **rpcjava** as follows:

```
>rpcjava -e env_file -d file.def [-i server implement class | -f server factory] [-DMULTI=true] [server options ...] [vm options ...]
```

Get system property in dedicated child process

In case you want to get a value using System.getProperty in the dedicated child process like

```
String lDelay = System.getProperty("L2U_DELAY");
```



You must specify L2U_DELAY tailored by "-DED_" in rpcjava as:

```
i.e.) >rpcjava -DDED_L2U_DELAY=10 ...
```

Then, you are able to get the value 10 in the dedicated child process.

Unit test with RPCDebug

Our universal client **RPCDebug** lets you test the Java application server without having the client program ready. **RPCDebug** takes place the Client in the Figure9.2. Please refer to 'rpcdebug' of "Chapter 4 Next utilities" in *Reference* for more information.

Reminder over implementation

Exception

The following is the Exceptions you can use in **rpcjava**. Please also refer to examples you can find in "samples" directory.

Exception	Description	How to use
RPCException	To be thrown when running into IO errors over start up.	No need to catch in the implementation class.
IDLFormatException	To be thrown when you have incorrect syntax in the IDL file.	No need to catch in the implementation class.
ServerException	You can throw exception when fail to process. The server will exit when make the exit flag to "true".	<pre>try { //Business logic } (catch Exception e) { ServerException se = new ServerException("message", e); se.setExit(true); //Exit flag throw se; }</pre>

Variable Named Server

Having "-s" option followed by the server name to **rpcjava**, you can assign a unique server name on the fly at start-up. You can also specify the server name by assigning "DCE_SERVERNAME" attribute in the environment file. But, remind you that you must specify the interface name as *\$variable_named_server* in the IDL file.

Encoding (Locale)

You can specify the server encoding with "DCE_LOCALE" attribute in the environment file. ASCII is the default. The following is the reference URL to tell you which locales you can set to the attribute.

<http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html>

Writing to the log file

Use “com.inspire.rpc.RPCLog” class in order to write your own log into the RPC log file. In your server implementation class, you can get RPCLog object using getLog() in “com.inspire.rpc.server.ServerContext”.

```
import com.inspire.rpc.server.ServerContext;
import com.inspire.rpc.server.ServerException;

public BasicsImpl implements Basics {

    private ServerContext ctx;

    public void setServerContext(ServerContext ctx) {
        this.ctx = ctx;
    }

    public int add(int x, int y) throws ServerException {
        ctx.getLog().debug(" add", " message");
        return x + y;
    }
}
```

“-classpath” requirement when running through AppMinder

Kicking off rpcjava required to give “-classpath” on Windows through AppMinder, you must set “-classpath” enclosed in the double-quotation (“”) as following:

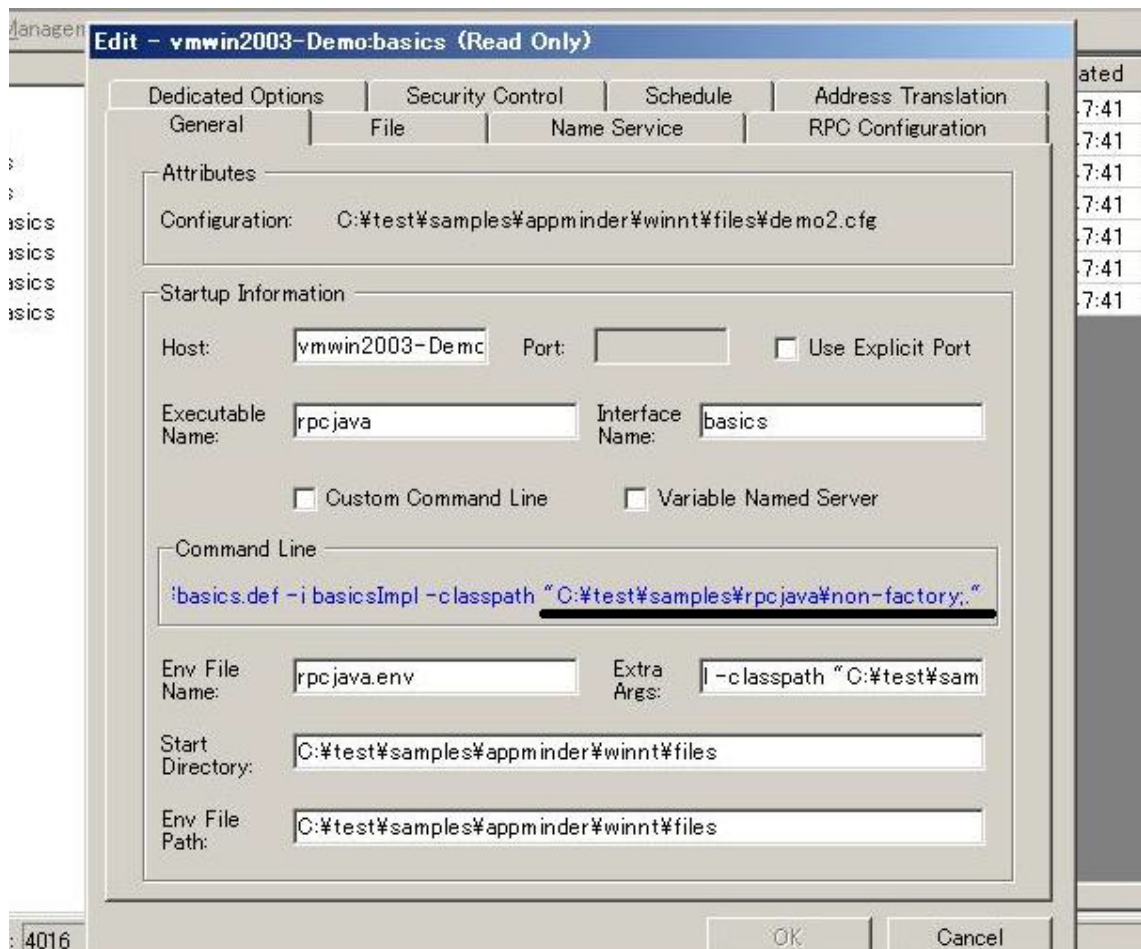


Figure 9.3: “-classpath” set through the Viewer (AmViewer)

OS environment variable

DCE_LISTEN_QUEUEUS

The client's RPC requests falls into the TCP/IP backlog queue stack at the server process if a RPC request is already executed by the server program. Once the RPC request is processed, then the request in the backlog queue in the stack is popped out and processed by the server program. Nextera can let you define the number of backlog queues with DCE_LISTEN_QUEUEUS environment variable at run time. The range is from 1 to INT_MAX and 5 is the default. You can also specify it in the environment file.

API

Please refer to “docs/api/rpcjava” directory.

Sample program

We have sample files in “samples/datatest/server/java” directory which covers all the data types Nextra supports.

Data type mapping

Table9.1 describes supported data types in simple data types and data types able to declare in the IDL file. Please refer to Table9.2 to understand each data types.

Table9.1: Data type mapping

Java application server	IDL file
short	short
int	int
int	long
float	float
double	double
char	char
void	void
Object	object

Table9.2: Each data types usage

Dimension	Array	Type	Sample functions in IDL file	Please refer to the following methods you can find in the sample files in order to understand how to deal with parameters.	
1 Dimensional	Simple	short	<code>short FuncShort([in] short shVar, [out] short shVarOut);</code>	<code>public short FuncShort(short shVar, ShortOut shVarOut)</code>	
		long	<code>long FuncLong([in] long lVar, [out] long lVarOut);</code>	<code>public int FuncLong(int lVar, IntOut lVarOut)</code>	
		int	<code>int FuncInteger([in] int nVar, [out] int nVarOut);</code>	<code>public int FuncInteger(int nVar, IntOut nVarOut)</code>	
		float	<code>float FuncFloat([in] float fVar, [out] float fVarOut);</code>	<code>public float FuncFloat(float fVar, FloatOut fVarOut)</code>	
		double	<code>double FuncDouble([in] double dVar, [out] double dVarOut);</code>	<code>public double FuncDouble(double dVar, DoubleOut dVarOut)</code>	
		char	<code>char FuncChar([in] char cVar, [out] char cVarOut);</code>	<code>public char FuncChar(char cVar, CharOut cVarOut)</code>	
		Object	<code>object FuncObj([in] object oVar, [out] object oVarOut);</code>	<code>public Object FuncObj(Object oVar, ObjectOut oVarOut)</code>	
	Constrained array	Constrained array	short	<code>void FuncConstrainedShortArray([in] short nRowDim1, [in] short nRowDim2, [in] short shArrVar[nRowDim1], [out] short shArrVarOut[nRowDim2]);</code>	<code>public void FuncConstrainedShortArray(short nRowDim1, short nRowDim2, short[] shArrVar, ShortArrayOut shArrVarOut)</code>
			long	<code>void FuncConstrainedLongArray([in] long nRowDim1, [in] long nRowDim2, [in] long lArrVar[nRowDim1], [out] long lArrVarOut[nRowDim2]);</code>	<code>public void FuncConstrainedLongArray(int nRowDim1, int nRowDim2, int[] lArrVar, IntArrayOut lArrVarOut)</code>
			int	<code>void FuncConstrainedIntArray([in] int nRowDim1, [in] int nArrVar[nRowDim1], [in] int nRowDim2, [out] int nArrVarOut[nRowDim2]);</code>	<code>public void FuncConstrainedIntArray(int nRowDim1, int[] nArrVar, int nRowDim2, IntArrayOut nArrVarOut)</code>
			float	<code>void FuncConstrainedFloatArray([in] int nRowDim1, [in] float fArrVar[nRowDim1], [in] int nRowDim2, [out] float fArrVarOut[nRowDim2]);</code>	<code>public void FuncConstrainedFloatArray(int nRowDim1, float[] fArrVar, int nRowDim2, FloatArrayOut fArrVarOut)</code>
			double	<code>void FuncConstrainedDoubleArray([in] int nRowDim1, [in] double dArrVar[nRowDim1], [in] int nRowDim2, [out] double dArrVarOut[nRowDim2]);</code>	<code>public void FuncConstrainedDoubleArray(int nRowDim1, double[] dArrVar, int nRowDim2, DoubleArrayOut dArrVarOut)</code>
			char	<code>void FuncConstrainedCharArray([in] int nColDim1, [in] char cArrVar[nColDim1], [in] int nColDim2, [out] char cArrVarOut[nColDim2]);</code>	<code>public void FuncConstrainedCharArray(int nColDim1, String cArrVar, int nColDim2, StringOut cArrVarOut)</code>
			void	<code>void FuncConstrainedVoidArray([in] int nRowDim1, [in] void byteArrVar[nRowDim1], [out] int nRowDim2, [out] void byteArrVarOut[nRowDim2]);</code>	<code>public void FuncConstrainedVoidArray(int nRowDim1, byte[] byteArrVar, IntOut nRowDim2, ByteArrayOut byteArrVarOut)</code>
Fixed array		Fixed array	short	<code>void FuncFixedLengthShortArray([in] short shArrVar[10], [out] short shArrVarOut[10]);</code>	<code>public void FuncFixedLengthShortArray(short[] shArrVar, ShortArrayOut shArrVarOut)</code>
			long	<code>void FuncFixedLengthLongArray([in] long lArrVar[10], [out] long lArrVarOut[10]);</code>	<code>public void FuncFixedLengthLongArray(int[] lArrVar, IntArrayOut lArrVarOut)</code>
			int	<code>void FuncFixedLengthIntArray([in] int nArrVar[10], [out] int nArrVarOut[10]);</code>	<code>public void FuncFixedLengthIntArray(int[] nArrVar, IntArrayOut nArrVarOut)</code>
	float		<code>void FuncFixedLengthFloatArray([in] float fArrVar[10], [out] float fArrVarOut[10]);</code>	<code>public void FuncFixedLengthFloatArray(float[] fArrVar, FloatArrayOut fArrVarOut)</code>	
	double		<code>void FuncFixedLengthDoubleArray([in] double dArrVar[10], [out] double dArrVarOut[10]);</code>	<code>public void FuncFixedLengthDoubleArray(double[] dArrVar, DoubleArrayOut dArrVarOut)</code>	

		char	void FuncFixedLengthCharArray([in] char cArrVar[10], [out] char cArrVarOut[10]);	public void FuncFixedLengthCharArray(String cArrVar, StringOut cArrVarOut)
		void	void FuncFixedLengthVoidArray([in] void byteArrVar[5973], [out] void byteArrVarOut[5973]);	public void FuncFixedLengthVoidArray(byte[] byteArrVar, ByteArrayOut byteArrVarOut)
	NT*) array	char	void FuncNullTerminatedArray([in] char cArrVar[], [out] char cArrVarOut[]);	public void FuncNullTerminatedArray(String cArrVar, StringOut cArrVarOut)
2 dimensional	Constrained array	char	void FuncConstrainedCharArray([in] int nRowDim1, [in] int nColDim1, [in] char sVar[nRowDim1][nColDim1], [in] int nRowDim2, [in] int nColDim2, [out] char sVarOut[nRowDim2][nColDim2]);	public void FuncConstrainedCharArray(int nRowDim1, int nColDim1, String[] sVar, int nRowDim2, int nColDim2, StringArrayOut sVarOut)
	Fixed array	char	void FuncFixedCharArray([in] char sVar[10][20], [out] char sVarOut[10][30]);	public void FuncFixedCharArray(String[] sVar, StringArrayOut sVarOut)
	NT*) array	char	void FuncNullTerminatedArray([in] char sVar[][], [out] char sVarOut[][]);	public void FuncNullTerminatedArray(String[] sVar, StringArrayOut sVarOut)

Nextra
Server Developer's Guide

2008/10/15	v5 2 nd Edition
2007/7/10	Chapter 9 Building Java application server
2007/4/16	3 rd Edition
2006/11/20	Added Environment variable setup
2005/11/11	Minor description change as to COBOL
2004/7/19	2 nd Edition
2003/4/18	First Edition

Author: Inspire International Inc.

Copyright © 1998-2010 Inspire International Inc.
Printed in Japan