

Nextra Configuration Guide

Version 5
2nd Edition



Contents

Chapter 1 Introduction	2
Using this Book	2
Format Conventions.....	3
Chapter 2 Administrative Strategies Overview.....	5
Nextra runtime features	5
Strategies Using Nextra runtime	7
Modifying Production Applications	8
Chapter 3 Configuring Your Application.....	10
When using Nextra	10
Increasing an Application's Fault Tolerance	10
Using Sub-Broker and local servers	13
Multi thread server	17
Dedicated server.....	17
Variable named server	22
Improving performance.....	29

Chapter 1 Introduction

Using this Book

Welcome to this Book. This book contains discussions of important topics with step-by-step instructions to introduce you to some of the more complex features you can use when building or managing distributed applications. Take a minute to read these two pages and ensure you have the background that we assume you have, and make sure you are reading the right book.

Who should use it

This book is designed for all developers and administrators who work with three-tiered applications. Whether you are responsible for building servers, scripting GUI clients or configuring an application, this book provides a detailed understanding of how you can manage and customize applications in an open distributed environment. You should not attempt to build a custom three-tiered application until you have read this book.

What you should already know

In general, you should have a basic understanding of client/server computing and of the limitations inherent in two-tiered architectures. You should have read *Read Me First* and *Server Developer's Guide* before delving into this guide.

When to use it

Through task-oriented instructions and plain-language discussions, this book provides you with expanded knowledge and skills to supplement the fundamentals you learned from *Server Developer's Guide*.

Refer to this book when you need to know more about certain Nextra features, such as Security and other administrative issues. This book and the Reference should provide all the information you need to understand the more advanced features or uses of the Nextra developer's package.

Topics

This book is divided into the following parts.

- Administrative Strategies Overview

- Adding robustness to your applications
- Strategically configuring applications including:
 - Robustness
 - Sub-Brokers and local servers
 - Special servers
- Security

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> -e <i>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 Administrative Strategies

Overview

As a system administrator, you are concerned about many issues regarding your applications or system. Reliability and performance may be just the beginning. This chapter explains how to address several administrative issues using Nextra.

If you will be administering an application in an open distributed environment, you will need to be familiar with the material in this chapter. You should already be familiar with system administration in general.

Nextra runtime features

The following items are descriptions of Nextra runtime features which can help you resolve administrative issues.

Parallel Brokers

By running multiple identical Master Brokers, you protect against Broker failure. If one Broker fails, its clients will instantly switch to the next Broker, with no lapse in performance.

Parallel servers

You can run multiple identical servers on one machine, or on different machines. These servers can work together, or if one goes down, the other(s) can compensate by taking on higher loads until the failed server is restarted.

Plus, you can distribute the work load to other servers so that the overall application performance will be improved.

Clustering

Cell is a unit comprising Broker(s) and its servers. There must be one Master Broker at least in the Cell. Having equal or more than two cells, we call Clustering which guarantees no application down time when running into hardware/network failures at one Cell.

Sub-Brokers and local servers

Sub-Brokers report to a Master Broker to get information. Clients using a Sub-Broker can connect to all servers that the Master Broker can connect to, but the reverse is not necessarily true: clients of the Master Broker may not be able to connect to servers reporting to a Sub-Broker. You can decide whether Sub-Brokers should share information about local servers registered under them, thus creating local servers in domains.

Multi thread server

In contrast to a normal server, a multi thread server is not a shared resource. It provides each client with its own thread copy within its process. Since v5.2, we support this feature in C and Java languages.

Dedicated server

In contrast to a normal server, a dedicated server is not a shared resource. It provides each client with its own server copy; that server copy takes requests only from that one client. This server copy does not need to return to its starting state at the end of each RPC. Thus, it can keep track of its client's RPC and tailor to its responses appropriately.

For DB access modules using dedicated server, you can retain its own cursor in its memory, not the cursor provided by RDBMSs.

Choose either Multi thread server or Dedicated server



You cannot select the both feature on in a server. In other words, you need to pick either Multi thread server or Dedicated server when you deploy your server.

Variable named server

Variable named server is a server equivalent except for its interface names (thus “variable named”), and its resources of data. You insert a variable as the interface name in the IDL file; you specify the value of each server's name at runtime. This type of server is useful if you will access multiple databases which have an identical structure. You do not need to code individual servers for each database.

When accessing from clients outside of private network or having multiple IP addresses assigned to server machine

What if you have a requirement that the client PCs sit outside of the network where the server machines located? Server processes are registered to the Broker with their local IP addresses which are invisible for the Nextra clients accessing from outside of the local network.

Nextra provides a feature to translate the server's IP address to the IP address visible to the Nextra clients. You only need to specify `DCE_EXTCLIENT` and `DCE_TRANSCRIPT` attributes to the Broker's environment file. Please refer to "Environment file attributes" in *Reference*. Additionally, those attributes are useful in case like two or more IP addresses assigned to a server machine.

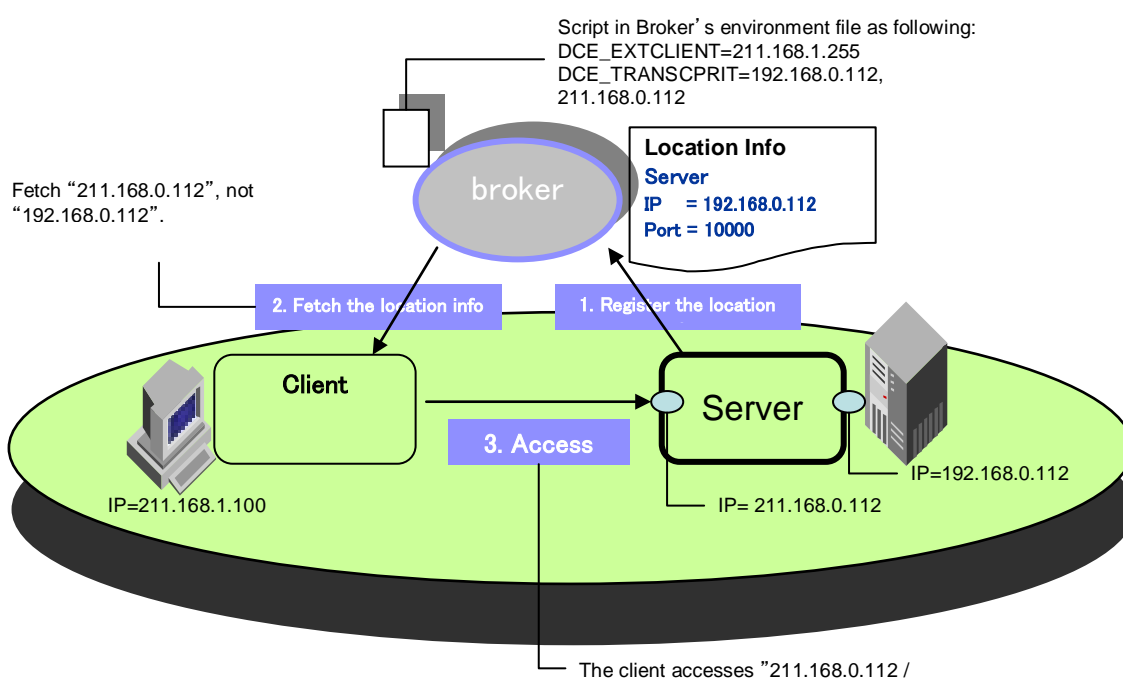


Figure 2.1: Mechanisms of `DCE_EXTCLIENT` and `DCE_TRANSCRIPT`

Strategies Using Nextra runtime

You are familiar with the features incorporated into Nextra, but how exactly can they address administrative issues? Some features can help solve problems in more than one area.

Guaranteeing reliability

Key buzzwords here are robustness and fault tolerance. To help minimize the effects of a failure on subsequent RPCs and user productivity, Nextra offers a scheme of active monitoring combined with system redundancy. Redundancy of application elements allows an application to continue operation, unaffected by any type of failure.

We strongly recommend adopting features like Parallel servers, Parallel Brokers and Clustering.

Improving performance

Nextra runtime features can help improve performance by increasing efficiency (load distribution in particular), and adding flexibility. Please refer to “Improving performance“ in “Chapter 3 Configuring Your Application”.

Modifying Production Applications

The following guidelines have been assembled to help you create a system by which you can keep track of application modifications and distribute changes efficiently.

Controlling versions

The objective of version control in a distributed environment is to make sure that your client stub and server skeleton are always based on identical definitions. Since the Nextra communications libraries are compatible across versions, the actual version numbers of the libraries is less important.

Note that you do not need to bring down an entire application in order to upgrade servers or stub/skeleton versions. Simply bring down and replace the modules that need to be replaced, while the rest of the application continues to run.

The following case should help you decide when to rename interfaces and how to keep track of modifications.

If your server code changes, but your IDL file does not change:

That is, if your new server functions receive and return the same number and the same types of input and output parameters as the old functions, then use the same interface name. Keep the old stub/skeleton. Recompile the server, and bring down the old servers and bring up the new ones one at a time, while the rest of the application continues to run.

If you add server functions without changing the definitions of the old functions:

Use the same interface name, but remember to edit the IDL file to include the prototypes of the new functions. Generate new stub/skeleton, recompile the server, and bring down the old servers and bring up the new ones one at a time, while the rest of the application continues to run. Distribute new client stubs to client programs that need the new functions, or distribute the stubs gradually when individual client programs are not in use.

If your original function definitions change, or if you subtract functions:

Rename the interface to ensure that old stubs do not try to send now-incompatible arguments into newly modified functions. You will likely want to include some kind of version information in the interface name, such as `cserver10` versus `cserver11`, etc.

Generate new skeleton(s), recompile the server, and bring up the new servers, while the rest of the application continues to run. Distribute client stubs as needed, or when individual clients are not in use. When all clients have the new stubs, bring down the old servers, eliminating the old interface.

If you must entirely prevent old clients from accessing old servers once the modified server is ready to run, then you need to bring down all clients at once, integrate their new skeletons, bring down all the old server instances, and replace them with the new server instances.

Alerting Users. It may be useful to leave an old server running after all the new servers are up. Modify this old server so that it sends an error code to any clients contacting it. Since you gave your application plenty of foresight, clients will recognize this error code and send their users a message informing them that they need to get new stub version from the application administrator.

Chapter 3 Configuring Your Application

This chapter covers the technical details of the different types of Brokers, servers and clients supported in the open distributed environment.

You may choose to read through this chapter, use it as reference, or both. You will need to understand the concepts presented in the chapter “Administrative Strategies Overview”.

When using Nextra

Configuring Backlog Queue

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. Nextra can let you define the number of backlog queues with **DCE_LISTEN_QUEUES** environment file attribute at run time. Additionally, you can also source it through OS environment variable. The range is from 1 to INT_MAX and 5 is the default.

For the parallel processing in order to improve the application performance, please consider to utilize [Parallel servers](#) and [Dedicated server](#) or [Multi thread server](#).

Increasing an Application’s Fault Tolerance

Nextra offers a scheme of active monitoring combined with system redundancy.

- Parallel Brokers allow instance recovery from machine dependent Broker failure.
- Parallel servers allow instant recovery from machine dependent failure.

Parallel Brokers

With Nextra, you can use parallel Brokers for increased fault tolerance. Servers attempt to register with each Broker, and clients use information from the first available Broker in their list. Once clients have a list of

servers, they rarely need to contact a Broker again. In the event that a client needs to contact a Broker which has gone down, it will gain look on its Broker list, and try to contact the next Broker in line.

Environment files

The syntax for specifying redundant Broker locations within an environment file for a client or a server is as follows:

```
DCE_BROKER=
  Broker_host, port_num
  Broker_host2, port_num2
  . . .
DCE_EXTSEARCH=1 ← Required for the client
```

There is no limit to the number of redundant Brokers you may have in the list.

Broker start-up

When starting up the Brokers, each Broker only parses the first host/port value in the list when choosing its port number, so the environment files for these Brokers cannot be identical.

For example, if your server's environment file specifies

```
DCE_BROKER=
  173.21.4.21, 12989
  173.21.4.22, 12988
  173.21.4.23, 12987
```

then you would need three different environment files for the Brokers,. You would need to start the first Broker with

```
DCE_BROKER=173.21.4.21, 12989
```

the second Broker with

```
DCE_BROKER=173.21.4.22, 12988
```

and the third Broker with

```
DCE_BROKER=173.21.4.23, 12987
```

Note



In this example, you would need to issue each Broker start-up command on a different machine corresponding to the difference in IP addresses.

-f option



Using `-f` option along side with *filename*, you can recover the broker's ServerList (Servers' location information) into its cache on the fly before server programs start. With this option, in the event of restarting only the broker, you no need to restart the server programs as well. Needless to say, you can use this option along with the AppMinder Management tool.

Parallel servers

By issuing a server's start-up command multiple times, you can run parallel instances of a server. (The OS chooses a unique PID and port for each instance, so you don't need to worry about parallel servers interfering with one another.) This feature is helpful in situations where service availability is critical or in cases where RPC volume overwhelms a single server.

Thus, if instances are running on different machines, the failure of one machine does not interrupt the application.

If multiple instances of a server are running, the Broker handles instance failure by redirecting client requests to another instance of the server. In addition, clients automatically distribute load by routing RPCs round-robin to all servers capable of fulfilling a particular RPC.

To start multiple servers, perform the following step:

- 1. Use a different environment file for each server instance.**

The environment file for each server instance must contain the same Broker list, but you should specify a different debugging log in each environment file, or else you will find the resulting group log, file extremely difficult to interpret. In a run-time situation where debugging has already been completed, this may not be a factor. In fact, you may not even have logging turned on.

2. Issue the start-up command once for each instance you wish to use

Session time out for communication failure

What if your application is running in WAN environment and needs to be ready for sudden communication disruption. Nextra RPC library is equipped with features preventing your application from hanging. What you need to do is to specify the following attributes in the environment files.

Default value	Broker	Client	Server
DCE_CLN_TIMEOUT	10 seconds*	LONG_MAX	
DCE_SVR_TIMEOUT	10 seconds*		LONG_MAX
DCE_RECEIVETIMEOUT	10 seconds*	LONG_MAX	LONG_MAX

*)LONG_MAX for Nextra3.6

DCE_CLN_TIMEOUT



Set in the client environment file. Define the duration, for this attribute specified the number of seconds that a client will wait for the return of an RPC sent by a server. If the client is idle for *duration* seconds, then the client will close the connection to the server.

DCE_SVR_TIMEOUT



Set in the server environment file. Define the, duration, for this attribute specifies the number of seconds that a server will remain idle waiting for an RPC after a client has connected to it. If no RPC arrives for *duration* seconds, then the server will close the connection to the client, preventing it from hanging.

Please refer to “Environment file attributes” in *Reference*.

Using Sub-Broker and local servers

Sub-Broker

Sub-Broker may be used to divide your application into logical zones, and to off-load naming request from a heavily-used Broker to one or more assisting Brokers.

A normal or Master Broker must be started first, just as in a one Broker application. Then, when you start a Sub-Broker, it registers with that Master Broker. During run-time, it informs the Master Broker of each server that registers with it. While the Master Broker knows of every server in the application, the Sub-Broker knows only a sub-set of that – only those servers which explicitly registered with that Sub-Broker.

If a Broker is started as a Sub-Broker, it scans the environment file for the host and port of the parent Broker with which it should register. Then, it establishes a connection at the port specified on the command line. Finally, the Sub-Broker connects with its parent Broker to verify that the parent Broker exists. The Sub-Broker then closes its connection to the parent Broker and waits until another application component connects with it.

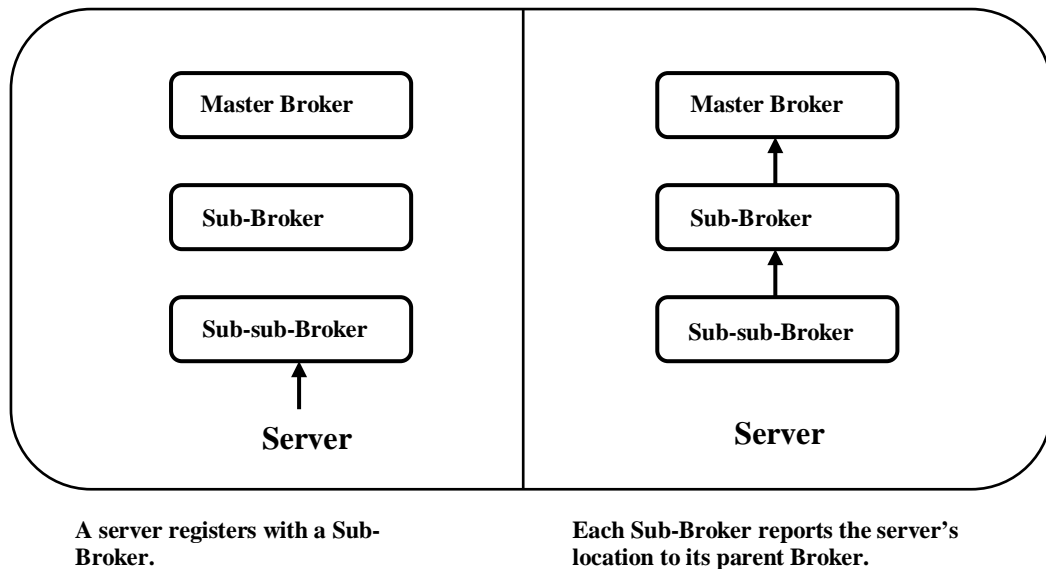


Figure 3.1: The Sub-Broker hierarchy during server start-up

Each time a server registers with a Sub-Broker, the Sub-Broker informs its parent Broker of the new server. After all servers in an application have registered, the Master Broker knows of all of them while each Sub-Broker knows only of those servers registered directly with it, and with those Sub-Brokers below it in the hierarchy. However, any Broker can find out about any server if a client so desires, except for servers designated as “local” or set DCE_LOCAL=1 attribute in the environment file.

If a parent Broker dies, its Sub-Brokers continue to run as though the parent Broker still existed. Each Sub-Broker keeps trying to contact the parent every time a new server registers because, if the parent Broker is

restarted in the same location, it keeps track of any new servers started after it. Note that any servers which register with a Sub-Broker while the parent Broker is down are never registered with any Broker above that Sub-Broker.

Starting Sub-Broker

To start a Sub-Broker, you must perform the following steps:

1. **Specify the location of the intended parent Broker using the `DCE_BROKER` attribute in the environment file.**

2. **Invoke the Broker utility with an extra argument specifying the port at which the Sub-Broker will establish a connection and listen for request. Type this command:**

```
broker -e env_file -p port_num
```

where *env_file* specifies the environment file containing the parent Broker's location, and *port_num* specifies the port at which the Sub-Broker should establish a connection.

If you want a server to register with this Sub-Broker, the server's environment file must include this Sub-Broker's host and port as one of the entries for `DCE_BROKER`.

Multiple Sub-Brokers

If a Sub-Broker does not know the address of the server that the client is looking for, it petitions its parent Broker for the server's location. If the parent Broker does not know the server's location, it petitions its parent Broker in turn, until a server has been found, or until the pinnacle of the Sub-Broker hierarchy has been reached. If the server is located, all Brokers between the server and the requesting client are alerted to the location of the server. In this manner, over time, the locations of all servers (except for "locale server") in the hierarchy are eventually propagated to all Sub-Brokers.

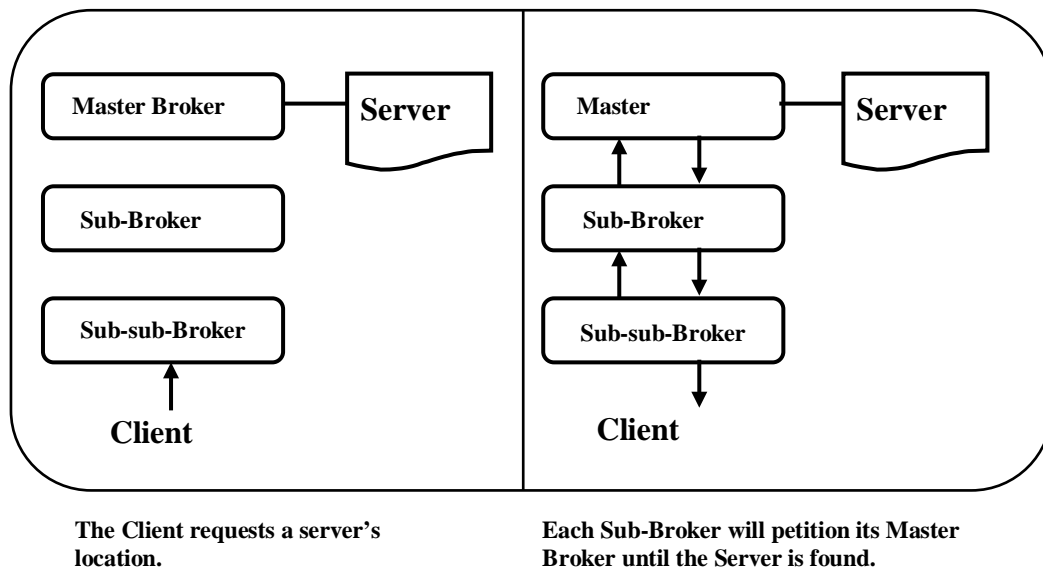


Figure 3.2: The Sub-Broker hierarchy during a client request

Distributing load with Sub-Brokers and local servers

The RPC routing load can be distributed over several Brokers through the use of Sub-Brokers.

As a Sub-Broker begins receiving client requests, it routes the request to any servers registered with it. If a client requests a different server, the Sub-Broker connects to its parent Broker and asks for the specified server. It then keeps the new server information itself, routing subsequent request directly to that server without referring to its parent Broker. The only other time that a Sub-Broker connects to its parent Broker is when it must report a new server's location.

By starting Sub-Brokers that register with other Sub-Brokers, you can build a hierarchy to distribute the RPC routing load as widely as you wish.

Local server

A local server is a server whose binding information is never propagated to any Broker higher in the hierarchy than Broker directly managing the local server. Figure 3.3 illustrates this feature.

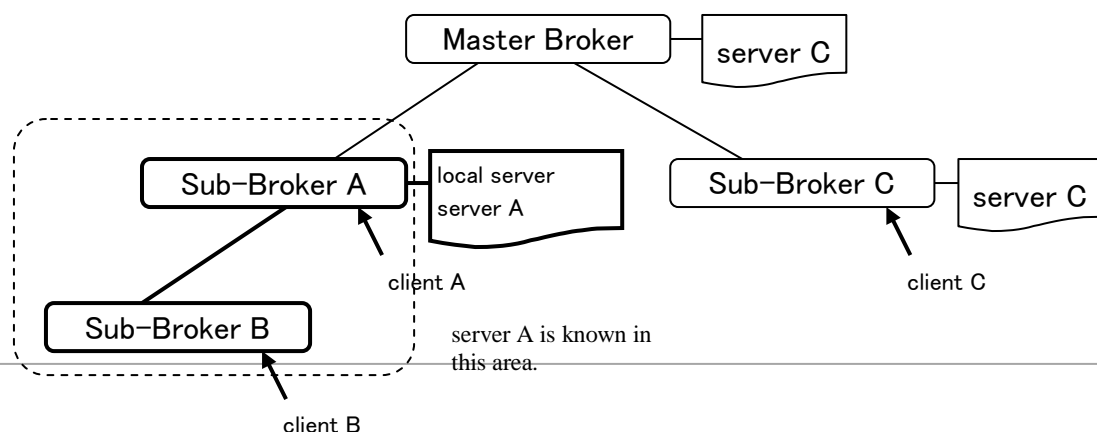


Figure 3.3: Sub-Brokers and local servers

As shown in the figure, the location of server A is only known by client A and client B, but never to client C.

Such configurations can be useful in situations involving a WAN, where the use of geographically remote resource is discouraged for performance or other reasons.

Starting local server

To make a server local, you need only add the following attribute to the server's environment file:

```
DCE_LOCAL=1
```

Multi thread server

A new feature Multi Threading Technology (MTT) available since v5.2 lets the server create and assign a thread for each client. The number specified with `DCE_THREADED` environment file attribute sets the upper limit for the number of clients that may connect at one time. For example, if `DCE_THREADED=4`, the server will create and assign up to four separate threads for the four clients at one time. If more than the allowed number of clients makes requests, the server will return the Nexera runtime error number of `DCE_MAXRPCEXCEEDED` to the extra clients. You can catch the Nexera runtime error with `dce_errnum ()` API in the procedure languages and Exception in the object languages.

In the event if both `DCE_DEDICATED` and this attribute are set, then the value for `DCE_DEDICATED` takes precedence.

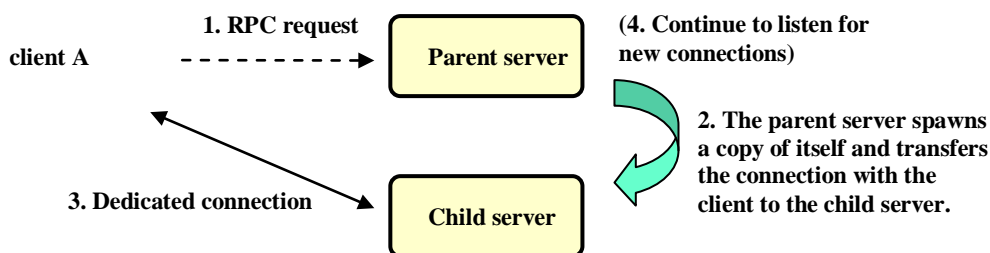
Dedicated server

Dedicated server: Overview

While a regular server fulfills any RPC request that it receives, regardless of the source of the request, a dedicated server fulfils requests only for a single client. The most important consequence of this is that a server can have a sense of “state”, so that it can, in effect, remember where it left off with the last RPC.

Typically, a regular server listens at a port for connection. When a client makes a request, the connection is completed. While a Nextra server is connected to a client, it does not accept other connections. After the RPC executes, the connection is broken and the server once again listens for new connections.

A dedicated server starts in a similar manner by listening for connections, but when first connected by a client, it starts up a new copy of itself: a “child” process. The child copy opens a dedicated connection with the client, and never even registers with the Broker. It only communicates with its client, while the “parent” server continues listening for further connections. The dedicated server’s connection to the client remains intact until the client sends the server a disconnect message, or the connection is destroyed by a physical network disconnection. If the dedicated server child process detects that its connection with the client has been destroyed or the client process has ended, it shuts itself down. Meanwhile, the original parent server process continues to listen for new clients, starting additional child copies as needed until it is killed. This arrangement ensures that one copy of the server is always available, and that a distinct copy is started for each client.



Shut down when the client A ends.

Figure 3.4: Dedicated server at runtime

If a client prematurely loses its connection to a dedicated server, the current RPC registers an error, and the subsequent RPC will connect with a new copy of the same server. The dedicated server that lost its connection shuts down.

Dedicated server: Technical details

Major benefits of dedicated server

There are two basic reasons why you would want to use a dedicated server.

1. Static information in the server between RPCs

This is the primary benefit of dedicated server. If your application would be better designed so that each client can be sure that no other clients are being serviced by the same server, then dedicated server is a good idea.

Sometimes a client making an RPC only wants a small piece of data, but the server must go through a lot of effort (e.g. gathering a large chunk of data) in order to get this small piece. Frequently, the next RPC that this particular client makes is a request for the “next” small piece of data which is right next to the first piece in the large chunk. It is obviously wasteful for the server to have to keep gathering that large chunk each time.

Using cursors and RDBMS security with different users and passwords both need a sense of state in order to work efficiently. You can also break up a transaction over multiple RPCs.

2. Quicker response from RPCs which take a long time.

There are some RPCs which simply take a long time (longer than the average RPC) to run, no matter what else has happened before. Large database queries, for instance, can take a minute or more to run. This means that the minimum response time a client experiences is going to be fairly long. However, the maximum response time can easily become far longer than this if you are using non-dedicated server, because there may be number of clients in the “queue” at the server. The sixth client in line at the server takes 30 seconds for each RPC would have a response time of 180 seconds.

By making the server dedicated, you reduce the maximum response time to almost exactly the same amount as it takes for a single RPC, because the client never has no wait in line. Recall that the way a dedicated server works is to have a “parent” server registered with the broker, and waiting for clients to make their first RPC. When each client makes its first RPC, the parent creates a “child” server to serve that client. This creation process is very quick relative to the actual RPC. Then the

child fulfills the RPC. The next RPC that this particular client makes automatically goes back to the child, which is guaranteed to be serving on only this client, so no one else can be in line in front of this client. Thus the response time is always going to be the amount of time that the actual function that the server is running takes.

Disadvantages of dedicated server

The major drawback of dedicated server is that the potentially large number of processes, one for each client, can be a performance hit on the machine on which the dedicated server and its child server processes are running.

If more than a few clients are to use copies of a dedicated server, it is a good idea to start up the server on two or more different machines to distribute the load generated by a large number of open servers.

Coding dedicated server and the client

The development process, compilation, and execution of the server are identical to those same processes for a regular server. To make a server dedicated, all you need to do is the following:

- 1. Set DCE_DEDICATED=N in the environment file, while N is the maximum number of live children the dedicated server can have at one time.**
- 2. Optionally, use the environment file attribute DCE_DEDICATEDLOGFILE to specify whether the child servers should create their own log files. See “DCE_DEDICATEDLOGFILE” in Reference for more information.**

Through this would make any server a functional dedicated server, you can likely write your server to take advantage of the fact that it always talks to the same client.

To communicate properly with dedicated server, the client should have one change:

- 3. The client must disconnect with, or “hang up on”, its dedicated server child process by calling `dce_dedDisconnect()` API when it is completely done using the server (not after each RPC).**

All other functions involved with using dedicated server are coded automatically in the stub. As with any client, you should use a DCE error checking APIs after each RPC to verify that it was completed properly. If the dedicated connection fails and the client connects to a new instance of the server, the new server will not be in the same state at the instance that died. The DCE error checking APIs provide the client with this

information. The client must then deal with the situation accordingly, perhaps restarting the current function at the beginning, or providing a warning to the user.

Write the server before the other parts of the application, and make sure you are confident of this piece before beginning on the result. In particular, make sure you're very confident of your design: dedicated server is a more complex construct to administer than non-dedicated server. Make sure that you try out your new dedicated server locally on the server machine before you alter any front-end clients. You will want to be familiar with the behavior of the server before you start worrying about network problems and other pieces working with the dedicated server.

See the chapter “Nextra API” in *Reference* for more information on the following topics: `dce_dedDisconnect()`, `dce_isDedicated()` and `dce_server_is_ded()`.

Initializing dedicated server

The initialization function option in **RPCMake** \mathcal{O} (the `-i` option) works with dedicated server. Initialization means preparing the environment for the server to work in before it actually handles any RPCs from clients. You can use `-i` option of the **RPCMake** to specify the name of an initialization function that the server should run after it sets up the Nextra environment, but before it starts accepting RPCs from clients. A typical example would be for a remote connectivity server to log into the remote machine and start up the legacy application, before accepting RPCs from clients.

With dedicated server, the `-i` option in the **RPCMake** works so that the parent server executes neither initialization nor exit functions. Instead, each child server executes the initialization functions immediately upon start-up, and the exit function immediately before shutting down.

Verifying that a server is dedicated

Imagine that you want to use a particular server which is already running, and you want to make sure that it is a “stateful” server. You can verify this by invoking `dce_isDedicated()` API function with your `interface/server` name as the argument. If the function returns “true”, then you know that the server is dedicated, and therefore holds state.

Dedicated server log files

With dedicated server, you can log data either only for the parent server, or for the parent and every child. If you set child server logging on, each child server that starts creates a new log file in a directory name like *<logfile_name>_d*. The individual file names are based on the servers' process ids.

- Use the `DCE_DEDICATEDLOGFILE` attribute in the environment file to set child server process logging on.
- The `DCE_LOG` attribute in the environment file names the parent server's log file. Child server process log files are saved in a directory named with the value of `DCE_LOG`, with “_d” as a suffix. Each log file name includes a number which is the PID of the child server process that created it.

Which log file is the right one?



If you do not know which dedicated instance your client is connected to, you can fine out by running the client at `DEBUG` debugging level. At this level, the client prints the PID of the server instance it is talking to in its log file. You can then match this PID with the name of a child log file on the server machine.

Loss of server

If a dedicated server dies when connected to a client, the next RPC to the server returns `PEERERROR`, and the RPC after that connects to a new dedicated server.

This behavior is desired because if a new server is provided immediately, it will not necessarily have the same state as the server that died. The `PEERERROR` message allows you to decide how the client should deal with this contingency, either recovering the previous server's state in some way, or informing the user to restart the transaction.

Trapping SIGCHILD

You cannot trap the `SIGCHILD` interrupt in a dedicated server, because the server library traps the signal and does not chain it.

Variable named server

Variable named server: Overview

Nextra allows the user to direct RPCs to any one of a set of parallel servers chosen at runtime; the servers in this set must be generated from the same IDL file. Because they offer the same functionality and interface, the parallel servers are conceptually “the same”, but are logically distinguished by their interface names.

As an example of using variable named server consider the following situation: the data fro a national corporation is kept in distinct regional DBs, all of which have the same data model (so that a query that works on one regional DB works on all the others). It is important for the end-users to be able to select which DB they are accessing. If the regions are divided into Northeast, Southeast, Southwest, etc., it is inefficient to have to create multiple IDL files and client stubs to access these servers that have identical functionality. Variable named server allows you to create one IDL file and stub, name each different server instance at start-up, and have the client p4rogram direct each RPC via an extra argument. Thus, the scenario resembles the diagram below, in which a client stub routes a specific query to the appropriate regional DB server.

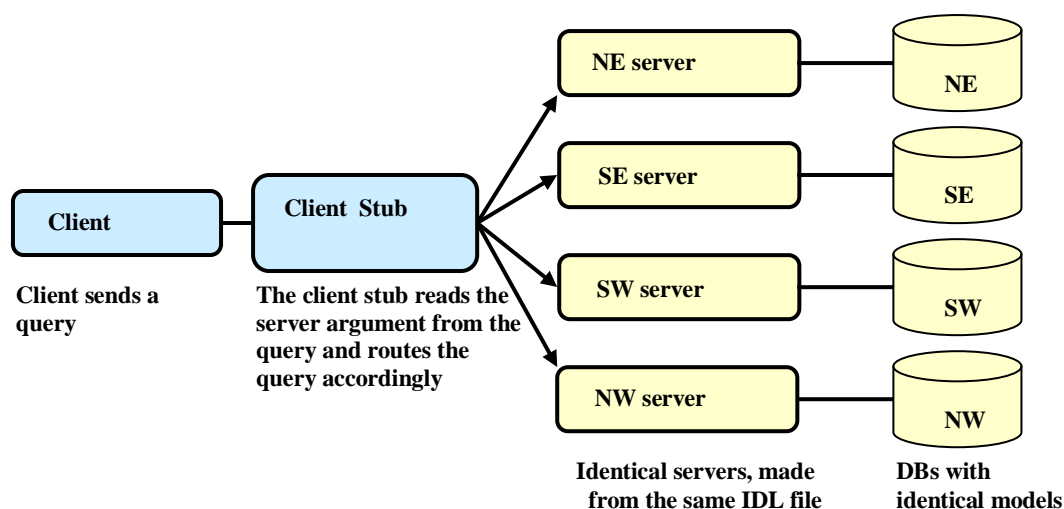


Figure 3.5: A RPC to multiple identical servers

Note that it is the same basic query that is routed to any of the regional servers. The difference in routing depends on one argument of the RPC, the name of the server the query needs to contact.

Since the regional DBs all follow the same model, the servers used to access them can call be generated from the same IDL file, provided that the interface name – the name of the server – is designated as a variable. Because servers generated this way are the same program, just with different names; they can all handle the same query.

When to use variable named server

Variable named server is useful when you have servers identical in functionality, which differ only in their name (and probably in the DB to which they connect), especially when the name is an arbitrary division like a region or department.

Do not use this capability unless the interface's function names and parameters are identical. Do not use it when the server's data sources are significantly different. In general, the more differences there are among servers of different names, the less suited this feature is for the implementation. For servers that accept identical inputs and return identical outputs, however, this feature prevents the problem of maintaining separate servers, allowing you to reuse one server under many different names. It simplifies the client program as well, because only one stub is necessary per set of variable named server.

Variable named server: technical details

Creating a variable interface name

To create a variable named functionality server, the IDL file must contain a variable in the interface statement. This may be done by hand by inserting a dollar sign (\$) in front of the interface name, making the interface name a variable:

```
interface $varname
```

To create a DB server, you must use the **SQLMake**. You can either use the **SQLMake GUI**, or type a command on the command line. If you use the GUI, just insert the name of the query file in the first text field, and *\$varname* in the "Server" field. When using the **SQLMake** on the command line to generate a IDL file, the variable server name should be used as an argument to the `-s` flag. On UNIX platforms, the variable server name must contain the initial dollar sign and be enclosed in single quotes:

```
sqlmake -q qfile -s '$varname' ...
```

On Windows, do not use quotes:

```
sqlmake -q qfile -s varname ...
```

When the client stubs are generated, you will notice that each stub function has an additional parameter (the first one on the parameter list),

used to specify the interface name of the actual server to which the client should connect.

Example IDL file

Here is a sample IDL file, defining a server with a variable interface name. Some parts of the example file.

```
# RPC interface definition file for: $server
[uuid(b8bd90e1-016a-11ce-b4b8-08002b2ff44f)
version(1.0)]

interface $server {
    int region_stats (
        [out] int revenues,
        [out] int expenses,
        [out] char manager);
}
```

IDL file comparison

The following comparison illustrates the difference between code generated from a fixed name contained IDL file and from a variable name contained IDL file. After this comparison, we illustrate the C client stubs and their differences.

First, the IDL file for the fixed name server:

```
# RPC interface definition file for: cserver
[uuid(00.06.09.00.00.00.06) version(1.1)]

interface cserver {
    int region_stats (
        [out] int revenues,
        [out] int expenses,
        [out] char manager);
}
```

and the variable named server:

```
# RPC interface definition file for: $server
[uuid(00.06.09.00.00.00.07) version(1.1)]

interface $server {
    int region_stats(
```

```

        [out] int revenues,
        [out] int expenses,
        [out] char manager);
    }

```

Client stub comparison

This example illustrates part of the C client stub generated by the **SQLMake** for the fixed name server (note that the argument in the call to `dce_findserver()` is the fixed string “`cserver`”).

```

/*#####
# Client Proxy Procedure Code
# generated by RPCMake
# on Thursday, April 8, 1993 at 11:4:2
#
# interface: cserver
# */

#ifdef __mpexl
#include "dceinc.h"
#else
#include <dceinc.h>
#endif

int region_stats(int *revenues, int *expenses, char
*manager)
{
    int rv = 0;
    int Socket;
    struct table *dce_table = NULL;

    if ((Socket = dce_findserver("cserver")) >= 0)
    {
        dce_table = dce_submit("cserver",
            "region_stats", Socket);
    }
    rv = dce_pop_int(dce_table, "dce_result");
    *revenues = dce_pop_int (dce_table, "revenues");
    *expenses = dce_pop_int (dce_table, "expenses");
    *manager = dce_pop_char (dce_table, "manager");
    if (dce_table) dce_table_destroy(dce_table);
    return(rv);
}

```

The following example shows the C client stub generated by the **SQLMake** for the variable named `server` (the argument in the call to `dce_findserver()` is the variable `ode_server` with the differences highlighted in bold type:

```

/*#####
# Client Proxy Procedure Code
/*#####
# Client Proxy Procedure Code
# generated by RPCMake
# on Thursday, April 8, 1993 at 11:2:23
#
# interface: server
# */

#ifdef __mpexl
#include "dceinc.h"
#else
#include <dceinc.h>
#endif

int region_stats(char *ode_server, int *revenues,
int *expenses, char *manager)
{
    int rv = 0;
    int Socket;
    struct table *dce_table = NULL;

    if ((Socket = dce_findserver(ode_server)) >= 0)
    {
        dce_table = dce_submit(ode_server,
            "region_stats", Socket);
    }
    rv = dce_pop_int(dce_table, "dce_result");
    *revenues = dce_pop_int (dce_table, "revenues");
    *expenses = dce_pop_int (dce_table, "expenses");
    *manager = dce_pop_char (dce_table, "manager");
    if (dce_table) dce_table_destroy(dce_table);
    return(rv);
}

```

Your client code will need to call each RPC with an additional argument: the name of the server to access. This parameter must be the first parameter in each remote function call, as shown in the stub above.

Bad serve name



If a client sends an unregistered server name in – if it sends a name for which there is no server – it will encounter an error just as for any client for which there is no server.

Parallel variable named server

Consistent with other situations in the open distributed environment, you may have several copied of each server (three SW servers, fifteen NEservers, etc.). A RPC will connect to nay server with the name specified in the call; all of the SWservers are exactly the same.

Variable named server with the same name



Variable named server does not require that they be named uniquely. But, if you name two dissimilar variable named servers by the same name, your application will not run properly. Variable named servers offering different interfaces must be named differently if the application is expected to function properly.

Starting parallel variable named servers manually

Continuing with the example, to stat there same serves manually, you would type the following line for each NE DB server:

```
DB_start -s NEserver -e multNE.env -q mult.qfile -d Nedatabase
```

To start the other regional server, again, probably on another machine (linked to a different DB), you would type the following line for each SW server:

```
DB_start -s SWserver -e multSW.env -q mult.qfile -d Swdatabase
```

Since all servers user the same IDL file, that means they should also use the same SQL statement file, or at least a statement file that generates an identical IDL file.

Instead of the –s command line option, you may specify the server name by setting an attribute in the environment file. Set `DCE_SERVERNAME=server_name`...in the server’s environment file



Naming server

If you try to name a fixed name server using the `–s` command line option or the environment file option, it will not work. These options are for variable named server only.

Improving performance

We will explain techniques how you improve the application performance over run time.

Multi threaded (Multi thread server)

Since v5.2, you can make your business logic multi threaded. Please also consider adopting [Parallel servers](#) as well in order to improve your overall application performance. If you have any reason unable to use Multi thread server feature, then consider to adopt to use [dedicated server](#) as the alternative.

Improve server response speed: When too many backlog queue at the server

Please consider to adopt [Multi thread server](#) or [Dedicated server](#) so that no backlog queue would be created at the server.

Reduce the amount of time to wait to cut off the session to the server when there is a network problem in between

Set `DCE_CLN_TIMEOUT` attribute in the client environment file. For instance, if you want to set the time out to 10, then specify as following in the client environment file:

```
DCE_CLN_TIMEOUT=10
```

The following is a C sample code how you retry when the time out takes place:

```
#define RETRY 3

for (i=0;i<RETRY;i++){
    result=RPC_function(argument);

    if (dce_errnum()==0) break;
    else if (dce_errnum()!=DCE_RPC_TIMEOUT){
        printf("Error: %s¥n", dce_errstr());
        exit(1);
    }

    if (i==RETRY-1){
        printf("Error: %s¥n", dce_errstr());
        exit(1);
    }
    sleep(20);
}
```

Nextra network locale

Nextra supports ASCII in the user's business logic and also uses ASCII as the network locale. However, you can choose UTF8 and other locales, but please consider using the same locale on your network and in your business logic so that you would not lose the amount of time for local translation in between.

Log file DEBUG level

After rolling out your application and being fairly stabilized, you shall consider lowering your application log levels: the lower the file I/O. Pleaser refer to "DCE_DEBUGLEVEL" in *Reference*.

Error file when using AppMinder

After rolling out your application and being fairly stabilized, you shall consider not generate error files with AppMinder. On the Viewer (AmViewer), please specify as following:

For Windows: Set "Error File Name" as "**nul**" and anything for "Error File Directory".

For Unix: Set "Error File Name" as "**null**" and "**/dev**" for the "Error File Directory".

You will see NO error file created by AppMinder.

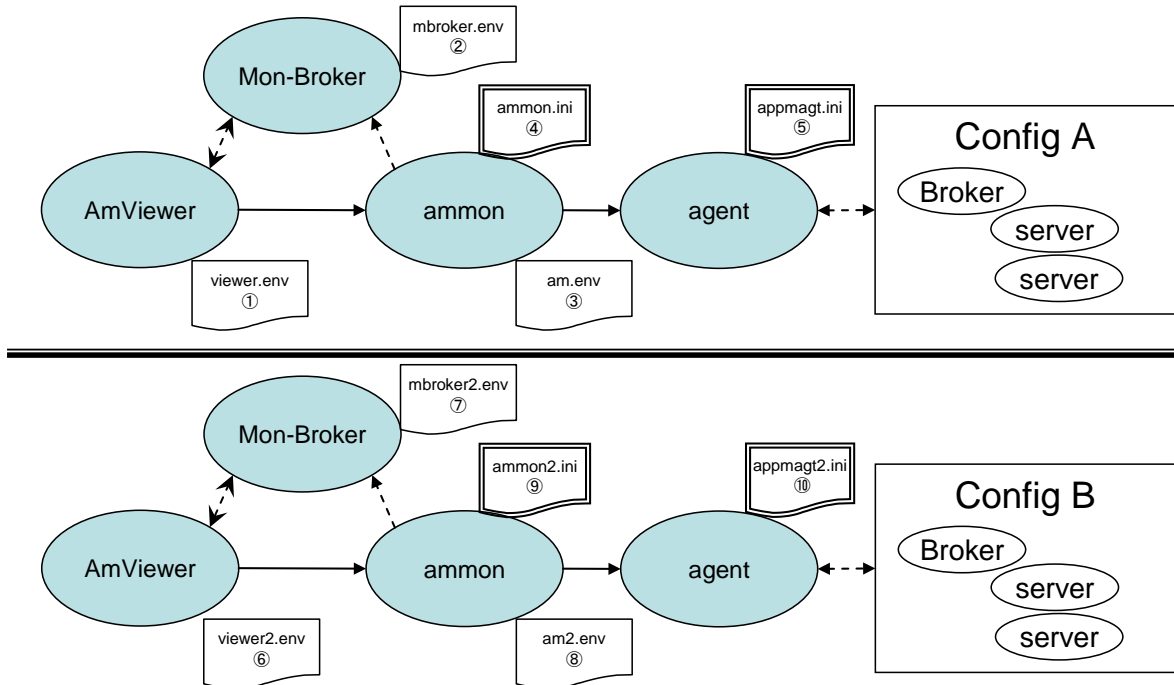
How to change the amount of time you get to see “Pinged” status on AppMinder

Modify `AMAGENT_PROC_START_PAUSE` attribute in the Agent initialization file (`appmagt.ini` is the default name). The default value is 100 seconds.

How to shorten the Agent (appmagt) management cycle

Modify `AMAGENT_MGMTINTERVAL` attribute in the Agent initialization file (`appmagt.ini` is the default name). The default value is 120 seconds.

How to manage multiple configurations on a single platform



Sample for each configuration

- 1st configuration

1. viewer.env

```
DCE_BROKER=localhost,9090
DCE_DEBUGLEVEL=NONE,DEBUG
```

2. mbroker.env

```
DCE_BROKER=localhost,9090
DCE_DEBUGLEVEL=NONE,DEBUG
```

3. am.env

```
DCE_BROKER=localhost,9090
DCE_DEBUGLEVEL=NONE,DEBUG
DCE_SERVERPORT=8001
DCE_CLN_TIMEOUT=<X> seconds
```

4. ammon.ini

```
AMMON_AGTPORT=7001
AMMON_SERVERARGS=-e am.env
AMMON_TRPCLOGLEVEL=NONE,DEBUG
```

5. appmagt.ini
 AMAGENT_PORT=7001

- 2nd configuration

6. viewer2.env
 DCE_BROKER=localhost,9091
 DCE_DEBUGLEVEL=NONE,DEBUG

7. mbroker2.env
 DCE_BROKER=localhost,9091
 DCE_DEBUGLEVEL=NONE,DEBUG

8. am2.env
 DCE_BROKER=localhost,9091
 DCE_DEBUGLEVEL=NONE,DEBUG
 DCE_SERVERPORT=8002
 DCE_CLN_TIMEOUT=<X> seconds

9. ammon2.ini
 AMMON_AGTPORT=7002
 AMMON_SERVERARGS=-e am2.env
 AMMON_TRPCLOGLEVEL=NONE,DEBUG

10. appmagt2.ini
 AMAGENT_PORT=7002

* Port number for each file is arbitrary. Set value in DCE_CLN_TIMEOUTsuit in your application environemtn.

Windows sample start-up batch commands

```
@echo off
start broker -e mbroker.env
start appmagt -c appmagt.ini -p mypassword -recover
start ammon -c ammon.ini -p mypassword

start broker -e mbroker2.env
start appmagt -c appmagt2.ini -pname pmon2 -p mypassword -recover
start ammon -c ammon2.ini -p mypassword
```

UNIX sample start-up shell commands

```
#!/bin/sh
broker -e mbroker.env &
appmagt -c appmagt.ini -p mypassword -recover &
ammon -c ammon.ini -p mypassword &

broker -e mbroker2.env &
appmagt -c appmagt2.ini -p mypassword -recover &
ammon -c ammon2.ini -p mypassword &
```

Tuning the packet size

DCE_PACKETSIZE defines the number of bytes that should be put into and received from the TCP layer. Please set this value bigger if your application deals many times with larger data than the default size (1460 bytes) so that you will experience the speed up in the data transfer.

Tuning send/receive socket buffer size

DCE_SO_RCVBUF_LEN environment file attribute specifies the maximum size, in bytes, of the receive socket buffer. For SOCK_DGRAM sockets, the receive buffer size may limit the maximum size of messages that the socket can receive. Default and maximum values are OS dependent; see individual protocol manual entries.

DCE_SO_SNDBUF_LEN environment file attribute specifies the maximum size, in bytes, of the send socket buffer. For SOCK_STREAM sockets, the send buffer size limits how much data can be queued for transmission before the application is blocked. For SOCK_DGRAM sockets, the send buffer size may limit the maximum size of messages that the application can send through the socket. Default and maximum values are OS dependent; see individual protocol manual entries.

"lingers" (waits) if there are untransmitted data

DCE_SO_LINGER environment file attribute defaulted as 10 milliseconds controls whether or not an application "lingers" (waits) if there are untransmitted data in the send socket buffer when the socket is closed.

Disabling the Nagle algorithm

Setting 1 to DCE_TCP_NODELAY environment file attribute defaulted as 0 disables the Nagle algorithm.

Sending data no fragmentation

When the data comprising 1 RPC is bigger than the size defined with DCE_PACKETSIZE environment file attribute, you may improve the speed to send the all data at once by setting DCE_PACKET4NOFRAGMENT environment file attribute to 1.

TCP layer tune up for a mission critical system

For future information, please refer to *Nextra Tuning Guide*.

Nextra Configuration Guide

2008/08/12	v5 2 nd Edition
2007/04/09	3 rd Edition
2006/12/13	Reduce the amount of time to wait to cut off the session to the server when there is a network problem in between
2006/11/14	How to manage multiple configurations on a single platform
2006/10/20	Added Backlog Queue explanation
2006/08/28	Added Improving performance
2005/05/19	When accessing from clients outside of private network or having multiple IP addresses assigned to server machine
2004/07/19	2 nd Edition
2003/04/18	1 st Edition

Author: Inspire International Inc.

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