Nextra
運用／設定ガイド

Version 6.5

目　次

第 1 章 はじめに

本書の利用方法

本書『運用／設定ガイド』では、分散アプリケーションを構築、管理するときに使用できる、より複雑な機能を説明するために、重要な項目について、手順を追って説明します。まず、対象読者に該当し、本書が適切なマニュアルであることを確認してください。

対象読者

本書は、3 層分散アプリケーションを扱う全ての開発者と管理者のために書かれています。サーバの構築、GUI クライアントのスクリプト化、またはアプリケーションの設定を行う場合に、オープン分散環境でのアプリケーションの管理とカスタマイズについて、本書は詳細な情報を提供します。本書を読んでから、3 層分散アプリケーションを構築してください。

前提知識

本書は、読者がクライアント／サーバコンピューティングと 2 層構造アーキテクチャの制限について基本的に理解していることを前提にしています。本書を読む前に、『はじめにお読みください』と『サーバ開発者ガイド』をお読みください。

本書の使用方法

タスクに沿った手順と簡単な言語の説明を通じて、『運用／設定ガイド』では、『サーバ開発者ガイド』で学んだ基本事項を補足、拡張する知識を提供します。
本書と『リファレンス』は、より上級の機能や Nextra ランタイムの使用方法について理解するために必要な全ての情報を提供します。
データアクセスの情報については、『トランザクション制御ガイド』を参照してください。

本書の内容

本書では、次の項目について説明しています。

☐管理戦略の概要
☐アプリケーションの堅牢性、耐障害性の向上
☐戦略的なアプリケーションの構成

表記規則

文中の表記規則

本書で使用する規則を理解しておくと、ユーティリティの使用方法などを容易に理解できます。

形式	説明	例
sub-text	ユーザが指定する必要がある値を示します。	*text*.def
bold	本文中では Nextra ユーティリティを示します。サンプル中では、強調される部分を示します。	**broker**
[brackets]	| がない場合は、オプションテキストを示します。| がある場合は、いずれか 1 つを選択することを示します。	[NONE | ERROR | WARN | DEBUG]

次の形式で区別されているパラグラフは、コード例です。

```
#include <stdio.h>

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

本書で使用するシンボル

本書では、次のようなシンボルを使用しています。

警告メッセージ

このシンボルに続くメッセージに、特別な注意を払う必要があることを示しています。このメッセージには重要な情報が含まれており、この情報を正しく理解してから先に進んでください。

ヒントメッセージ

このシンボルに続く本文は、必須ではありませんが状況に応じて役立つ手順であることを示しています。

オプションメッセージ

このシンボルに続く本文はオプションであることを示しています。内容は、追加機能または代替手法の概要、ある概念を理解するために役立つプロセスステップの詳細などです。

デバッグのヒント

このシンボルに続く本文は、プロジェクトの現在のステップをデバッグする手順が含まれていることを示しています。この方法はあくまで参考であり、別の有効なデバッグ方法の使用を妨げるものではありません。

第 2 章 管理戦略の概要

システム管理者は、アプリケーションまたはシステムについて、さまざまな問題に直面します。まず、信頼性とパフォーマンスが問題になります。この章では Nextra を使用して、管理上の問題を解決する方法について説明します。

オープン分散環境アプリケーションを管理するには、この章の内容に精通する必要があります。この章は、システム管理全般についての知識を前提とした内容になっています。

Nextra ランタイムの機能

以下は、Nextra ランタイム機能の説明です。この機能を使用することで、管理上のさまざまな問題を解決することができます。

並列ブローカ

複数のブローカを起動すると、複数サーバの実行と同じような方法で、耐障害性を向上させます。クライアントが環境ファイル中に記載されている最初の使用可能なブローカに問い合わせをします。サーバは環境ファイル中に記載されている全てのブローカに自身の情報を登録します。1 つのブローカが使用不可能になった場合、クライアントは環境ファイルに記載されている次のブローカに問合せを行います。

並列サーバ

単一のマシン、または複数の異なるマシン上で、複数の同じサーバを実行することができます。これらのサーバは同時稼動が可能です。あるサーバプロセスがダウンしても、そのサーバプロセスが再起動するまで、他のサーバプロセスが過負荷を受け持ち、埋め合わせる機能を果たします。

堅牢性の向上に加え、複数の同一サーバを起動することは、サーバへの負荷を分散するために有効です。この方法は、アプリケーションの処理速度をかなり向上させます。

クラスタリング構成

クラスタリング構成を使用するということは、少なくとも 2 つ以上の同一セルを複数の場所で実行させるということを意味します。あるセルを構成するハードウェアやネットーワークに障害が発生した場合、別のセルにあるサービスにアクセスすることにより、アプリケーションユーザには継続してサービスを提供することができます。

サブブローカとローカルサーバ

ローカルサーバは、それが登録されたブローカ／サブブローカと、そのブローカ以下の階層のブローカのみに位置を知られているサーバです。

サブブローカを使用することにより、アプリケーション内にドメインを作成することができます。サブブローカにローカルサーバを登録した場合、マスタブローカへサーバ位置情報の問合せを行うクライアントは、そのローカルサーバの位置情報を得ることができません。
ローカルサーバを登録したサブブローカ以下の階層にアクセスするクライアントは、そのローカルサーバの位置情報を得ることができます。

これは、必要なとき、つまり、サーバがローカルドメイン内にないときにのみ、クライアントがローカルドメインの外に出るということを意味します。

たとえば、国中の主要都市を結ぶ、高価な専用回線により接続された LAN を使用する場合、応答が遅くて料金の高い電話回線でアプリケーションを使う頻度を極力押さえたいとは、誰もが思うことです。ローカルサーバを持つサブブローカは、こういう場合に役立ちます。

マルチスレッド・サーバ（Multi Thread Server）

v5.2 より、マルチスレッド・サーバが使用できるようになりました。これにより、それぞれのクライアントのリクエストはサーバプロセス内の各スレッドにて処理されます。現在サポートされている言語は、C と Java です。

デディケイテッド・サーバ（Dedicated Server）

環境ファイル中の DCE_DEDICATED 属性を指定することにより、サーバを簡単にデディケイテッド・サーバにすることができます。クライアントがデディケイテッド・サーバに要求したときは、そのクライアントのためだけに子プロセスを作成します。これによって、各クライアントは、子サーバプロセスを専有できます。子サーバプロセスは、1 つのクライアントからの要求しか受け付けません。

DB アクセス・モジュールをデディケイテッド・サーバにした場合、カーソルなどの状態を保持することができます。ただし、ここでのカーソルは DB が提供するカーソルではなく、DB アクセス・モジュールが持つカーソル機能を指します。

マルチスレッド・サーバとデディケイテッド・サーバを同時に使用できません。

よって、実際にご利用する場合には、どちらかの機能を選択することになります。

バリアブル・ネームド・サーバ（Variable Named Server）

バリアブル・ネームド・サーバとは、そのインタフェース名(つまり「可変名」)以外のプログラムが同じサーバのことです。IDL ファイルにはインタフェース名として変数を指定し、実行時にサーバ名の値を指定します。同じ構造を持つ複数の DB にアクセスする際に、この種のサーバが役立ちます。個々の DB ごとに独自のサーバをコーディングする必要がないからです。 バリアブル・ネームド・サーバを DB アクセス・モジュールに使用すると、新しいサーバを作成せずに、同じ構造持つ、異なる DB にアクセスすることができます。これにより、開発者の時間を確実に節約し、柔軟性と有用性を提供します。

プライベートネットワークの外にあるクライアントからのアクセスや、サーバマシンに 2 つ以上の IP アドレスが割り振られている場合

プライベートネットワークの外から、Nextra のサービスを使用したい要求があった場合にどのような運用構成にしたら良いでしょうか？プライベートネットワーク内でサービスを起動したサーバプロセスには、ローカル IP アドレスが割り振られ、そのローカル IP アドレスがブローカに登録されています。このままでは、たとえクライアントプログラムがサーバ位置情報をブローカより取得したとしても、プライベートネットワーク内に位置するサーバプロセスにはアクセスできません。そういった要望にお答えするためのブローカ用環境ファイル属性をご用意しました。DCE_EXTCLIENT と DCE_TRANSCRIPT です。これらの属性の詳細については『リファレンス』「環境ファイル属性」を参照してください。
そのほかに、サーバプロセスを起動するサーバマシンに 2 つ以上の IP アドレスが割り振られていた場合にも、これらの環境ファイル属性が有効です。

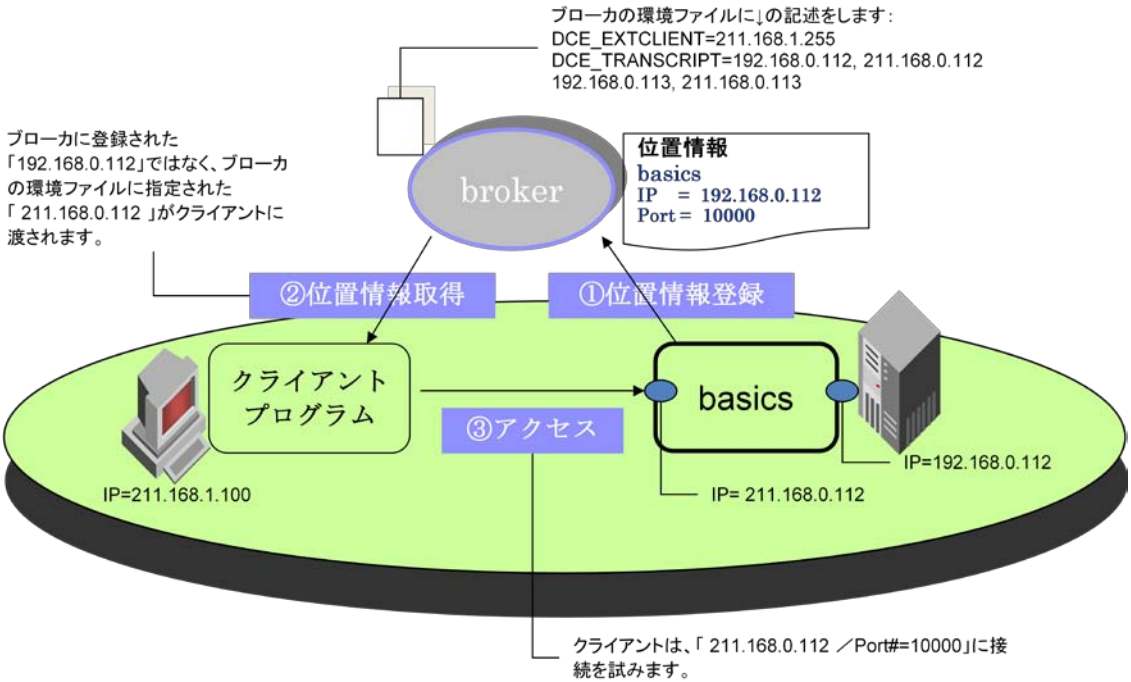


図 2.1 ： ブローカ用環境ファイル属性（DCE_EXTCLIENT、DCE_TRANSCRIPT）を指定した場合の、サーバプロセス IP アドレス情報について

Nextra ランタイムを使用した戦略

Nextra に組み込まれている機能に精通している方でも、管理システムがどのようになっているかをご存じでしょうか？ 機能のいくつかは、1 つ以上の局面で問題を解決するのに役立ちます。

信頼性の保証

ここでのキー用語は、堅牢性と耐障害性です。アプリケーションの構成要素の冗長性によって、どのような故障にも影響されず、アプリケーションの継続運転が可能になります。運用の際、「並列サーバ」、「並列ブローカ」、「クラスタリング構成」にすることをお勧めします。

パフォーマンスの向上

Nextra ランタイムの機能を用いると、効率を上げ(特に負荷分散)、柔軟性を加えることにより、パフォーマンスを向上することができます。『第 3 章アプリケーションの設定』の「パフォーマンス向上」を参照してください。

製品アプリケーションの修正

アプリケーション修正の記録をとり、変更点を効率的に配布するシステムを作成するためのガイドラインを次に示します。

バージョンの管理

分散環境におけるバージョン管理の目的は、常にクライアント・スタブとサーバ・スケルトンが必ず同じ定義に基づくようにすることです。Nextra 通信ライブラリはバージョンによっては互換性がありませんのでご注意ください

サーバやスケルトンのバージョンをアップグレードする場合に、アプリケーション全体を終了させる必要がない点に注意してください。交換するモジュールだけを停止して、交換しますが、そのとき、残りのアプリケーションは実行を続けることができます。

インタフェースをリネームして、修正の記録をどのようにとるかを決定する際に、次の説明が役立つでしょう。

IDL ファイルはそのままで、サーバプログラムを変更する場合：

つまり、新しいサーバ関数が古い関数と同じ数、同じ型の入出力引数を受信し、返す場合は、同じインタフェース名を使用します。スケルトンも古いものを使用できます。サーバをリコンパイルし、古いサーバを停止し、新しいものを起動します。この間、残りのアプリケーションは実行を続けることができます。

古い関数の定義を変えずに、サーバファンクションを追加する場合：

同じインタフェース名を使用しますが、忘れずに新しい関数のプロトタイプを含むように、IDL ファイルを編集してください。新しいスケルトンを生成し、サーバをリコンパイルし、古いサーバを停止し、新しいものを起動します。この間、残りのアプリケーションは実行を続けることができます。新しい関数を必要とするクライアントプログラムに新しいクライアント・スタブを配布するか、あるいは各クライアントプログラムを使用していないときに、逐次スタブを配布します。

関数の定義を変更する場合、またはファンクションを減らす場合：

現時点で不適合になっている引数を、古いスケルトンが新たに修正した関数に送らないように、インタフェースをリネームします。cserver10 と cserver11 のように、インタフェース名にバージョン情報を含めることも 1 つの方法です。

新しいスケルトンを生成し、サーバをリコンパイルし、新しいサーバを起動します。この間、残りのアプリケーションは実行を続けることができます。必要に応じて、あるいは各クライアントが使用していないときに、クライアント・スタブを配布します。全てのクライアントが新しいスタブを持つ場合、古いインタフェースを削除して、古いサーバを停止します。

変更したサーバが起動できる状態になった後、古いクライアントが古いサーバに完全にアクセスできないようにする場合は、即座に全てのクライアントを終了し、新しいスケルトンを統合し、古いサーバプロセスを全て停止して、新しいサーバプロセスと交換しなければなりません。

第 3 章 アプリケーションの設定

この章では、オープン分散環境において提供される各種のブローカ、サーバおよびクライアントの使用テクニックの詳細について説明します。

この章を読む前提事項として、「管理戦略の概要」の章で紹介した概念を理解している必要があります。

利用に際して

Backlog Queue（バックログ・キュー）の設定

Broker に大量のサーバを同時に登録する場合や、Nextra アプリケーションサーバに対して大量の同時アクセスが発生した場合に、TCP/IP Port で Backlog Queue（バックログ・キュー）に入ることができなかったリクエストは、Queue（キュー）溢れ（Time Out）になります。これが発生した場合、Nextra/RPC ライブラリでは、並列サーバなどに迂回する機能を備えておりますが、Queue 溢れになる状況は好ましくありません。

参考までに、Queue 溢れが起こった場合には、Unix では”ETIMEOUT”、Windows で”WSATIMEOUT”のシステムエラーとなり、Nextra 環境ファイルで指定したログファイルに記録されます。

これを回避するために、Nextra ランタイムでは環境ファイル属性「DCE_LISTEN_QUEUES」をご提供しております。更に、環境変数としてソースすることも可能です。実行する環境に併せて適切に設定してください。設定有効値は 1～200（Nextra5 以降は INT_MAX）でデフォルト値は 5 となっています。

「DCE_ISTEN_QUEUS」で指定した値は、Queue のサイズを指定するもので、実際の Backlog Queue の数ではありません。各プラットフォームでの Backlog Queue 数は、「LISTEN_QUEUES」指定値のおおむね 1.5～2 倍になります。

同時並列処理が必要なアプリケーションの場合には、並列サーバを利用しながら、デディケイテッド・サーバ（Dedicated Server）、またはマルチスレッド・サーバ（Multi Thread Server）を利用してください。

アプリケーションの耐障害性の向上

Nextra は、ブローカ、サーバの冗長性により耐障害性を提供します。

- 並列ブローカによって、クライアントは、あるマシンで起動するブローカにアクセス不能になった場合、別マシンで起動するブローカよりサービスを受けることが可能になります。
- 並列サーバによって、クライアントは、あるマシンで起動するサーバにアクセス不能となった場合、別マシンで起動するサーバよりサービスを受けることが可能になります。

並列ブローカ

Nextra では、耐障害性の向上のために、並列ブローカを使用することができます。サーバは、各ブローカに登録され、クライアントは、リスト内の最初の利用可能なブローカから受け取った情報を使用します。クライアントはサーバ位置情報を獲得すると、ブローカに再接続する必要はありません。あるクライアントが、ブローカに接続しようとして、そのブローカが停止していた場合に、クライアントは環境ファイル中のブローカのリストをチェックし、リスト中の次の利用可能なブローカに接続します。

環境ファイル属性

クライアント／サーバは、ブローカの位置を環境ファイル属性で指定します。

```
DCE_BROKER=
        Broker_host,  port_num
Broker_host2, port_num2
...
DCE_EXTSEARCH=1   ← クライアントに必須
```

リストに加えるブローカの数に制限はありません。

ブローカの起動

ブローカ起動時、各ブローカは、ポート番号を選ぶ際に、リスト中の最初のホスト／ポート番号だけを解析します。これにより、複数のブローカを起動する場合、各ブローカの環境ファイルが同じになることはありません。

たとえば、サーバの環境ファイルが次のように指定しているとします。

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

この場合、3 つのブローカを起動することになります。その為、3 つの異なる環境ファイルが必要になります。

最初のブローカ環境ファイル中の DCE_BROKER 属性は次の通りです。

```
DCE_BROKER=173.21.4.21, 12989
```

2 番目のブローカ環境ファイルは次のように設定します。

```
DCE_BROKER=173.21.4.22, 12988
```

3 番目のブローカ環境ファイルは次のように設定します。

```
DCE_BROKER=173.21.4.23, 12987
```

注意

この例では、IP アドレスの違いに合わせて、異なるマシン上で各ブローカの起動コマンドを発行する必要があります。

-f オプション

ブローカ起動時、-f オプションを指定して *filename* で指定されたファイルを読込むことにより、サーバ起動を待たずして、前回までのブローカキャッシュ内サーバリスト(サーバ位置情報)を復元します。これにより、ブローカのみの再起動が必要な場合に、サーバプログラムを再起動する必要がありません。 管理ツール AppMinder と併用している場合にも利用可能です。

並列サーバ

サーバ起動コマンドを複数回発行することによって、サーバの並列プロセスを実行することができます(OS は、個々のプロセスごとに別の PID とポートを選択するので、並列サーバが互いに干渉し合うことはありません)。サービスが処理の限界を超えるような場合に、この機能が有効になります。

ただし、同じ環境ファイルを使用して立ち上げる場合、そのサーバは同じ属性を使用していることに注意してください。たとえば、ログファイルなどを複数のサーバが共有することになります。これを避ける為には、サーバ毎に環境ファイルを用意することをお勧めいたします。

複数のプロセスが異なるマシンで実行されている場合には、1 台のマシンでの障害によって、アプリケーション全体が影響を受けることはありません。

複数のサーバプロセスが実行されていて、そのうち 1 つのサーバプロセスで障害が発生した場合は、クライアントは使用可能な別のサーバプロセスにアクセスします。さらに、クライアントは、RPC 毎にラウンドロビンにより別のサーバプロセスにアクセスします。これにより、負荷が分散されることになります。

複数サーバを起動するには、以下のように行ってください。

1. **各サーバプロセスに、異なる環境ファイルを割り当てます。**

   それぞれのプロセスの環境ファイルには、同じブローカリスト(DCE_BROKER 属性で指定)が含まれていますが、もし同じディレクトリから各サーバを起動する場合は、それぞれの環境ファイルには異なるログファイル名を指定してください。

2. **使用する各プロセスに対して、起動コマンドを 1 度だけ発行します。**

通信障害に備えたタイムアウト(TO)の設定

WAN 環境で利用の場合、PC クライアントとの接続が突然切れる場合などが考えられます。Nextra/RPC ライブラリでは、そういった状況に対応するために、以下の環境ファイル属性をご提供しております。

デフォルト値	Broker	クライアント	サーバ
DCE_CLN_TIMEOUT	10 秒*	LONG_MAX	
DCE_SVR_TIMEOUT	10 秒*		LONG_MAX
DCE_RECEIVETIMEOUT	10 秒*	LONG_MAX	LONG_MAX

*) Nextra3.6 では、LONG_MAX になります。

DCE_CLN_TIMEOUT は、クライアント側 RPC にて、データを受信する際に通信が TO する時間を指定します。DCE_SVR_TIMEOUT は、サーバ側 RPC にて、データを受信する際に通信が TO する時間を指定します。DCE_RECEIVETIMEOUT に値を設定すると、

DCE_CLN_TIMNEOUT と DCE_SVR_TIMEOUT 両方にこの値が設定されます。詳しくは『リファレンス』「環境ファイル属性」を参照してください。

その他、サーバマシンの TCP/IP Kernel パラメータにおける、TCP/IP メッセージ送信のトータルの再送タイムアウト(RTO)の値の設定が必要な場合があります。この再送時間の設定は、TCP Kernel パラメータチューニングとなり、各 OS により異なります。Solaris/HP-UX では、「tcp_ip_abort_interval」がそのパラメータになります。その他の OS については、各 OS のマニュアルをご確認ください。

サブブローカとローカルサーバの使用

サブブローカ

サブブローカは、アプリケーションを論理的なゾーンに分けるために、また、頻繁に使われるブローカから 1 つ以上のサブブローカに、サーバ位置情報要求の負荷を減らすために使用されます。

最初に、マスタブローカを起動する必要があります。その後にサブブローカを起動すると、サブブローカはマスタブローカに登録されます。実行中、サブブローカは登録されている各サーバ位置情報をマスタブローカに知らせます。マスタブローカはアプリケーション中の全てのサーバを認識しますが、サブブローカはマスタブローカのサブセットだけ、つまり、そのサブブローカに明示的に登録されたサーバだけを認識します。

あるブローカがサブブローカとして起動されると、このサブブローカは、自分自身を親ブローカに登録するのに必要なホスト名とポート番号を取得するために、環境ファイルをスキャンします。それから、コマンドラインに指定されたポートで接続を確立します。最後に、このサブブローカは親ブローカに接続して、親ブローカが存在することを確認します。それから親ブローカとの接続を切り、別のアプリケーションからの要求があるまで待機します。

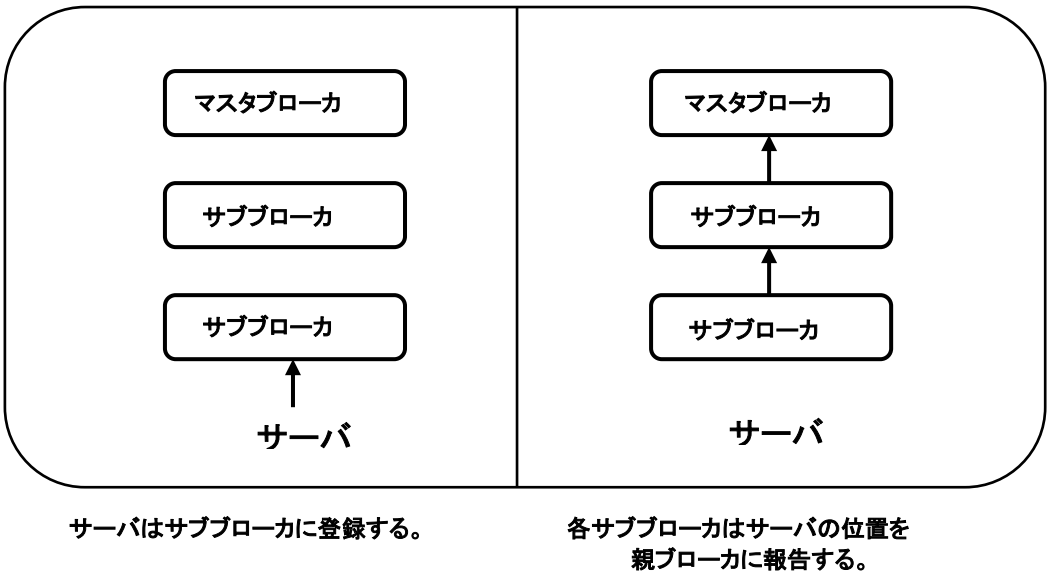


図 3.1： サーバ起動中のサブブローカ階層

あるサーバがサブブローカに登録される度に、このサブブローカは親ブローカに新しいサーバが登録されたことを通知します。アプリケーション中の全てのサーバが登録されると、マスタブローカは全てを認識するようになりますが、各サブブローカは、直接登録されたサーバや、その階層レベル以下のサブブローカに登録されたサーバだけを認識します。どのブローカも、「ローカル」(環境ファイル DCE_LOCAL 属性で設定)と設定されたサーバを除き、クライアントが望む限り、全てのサーバを見つけることができます。

親ブローカが停止しても、サブブローカは親ブローカが存在しているかのように処理を続けます。各サブブローカは、新しいサーバを登録する度に親ブローカに接続しようとします。親ブローカが動いていないときに、サーバをサブブローカに登録する際は、そのサブブローカの階層レベル以上のブローカには登録できない点に注意してください。

ただし、AppMinder を使用している場合は、上位ブローカが再起動されると、全ての下位のブローカ、サーバは再登録されます。

サブブローカの起動

サブブローカを起動するには、次のように行ってください。

1. 環境ファイルの DCE_BROKER 属性を使用して、目的の親ブローカの位置を指定します。

2. サブブローカが接続を確立し、要求を待つポートを指定する引数を追加して、ブローカを起動します。

次のコマンドでブローカを起動します。

> broker -e *env_file* -p *port_num*

ここで、*env_file* は親ブローカの位置が入っている環境ファイル、*port_num* はサブブローカが接続を確立するポートです。

サーバをこのサブブローカに登録するには、サーバの環境ファイルにこのサブブローカのホストとポートが DCE_BROKER 属性のエントリの 1 つとして含まれていなければなりません。

複数のサブブローカ

クライアントが探しているサーバのアドレスをサブブローカが認識しない場合、サブブローカは親ブローカにサーバの位置を要求します。親ブローカがサーバの位置を認識しない場合は、サーバが見つかるまで、あるいはサブブローカ階層の頂点に達するまで、次々にその親ブローカに要求していきます。サーバの位置情報が確認されると、このサーバと、そのサーバを要求しているクライアントの間にある全てのブローカには、サーバの位置情報が伝播されます。このようにして、いずれは全てのサーバ（ただし「ローカルサーバ」を除く）の階層内での位置は、全てのサブブローカに伝播されます。

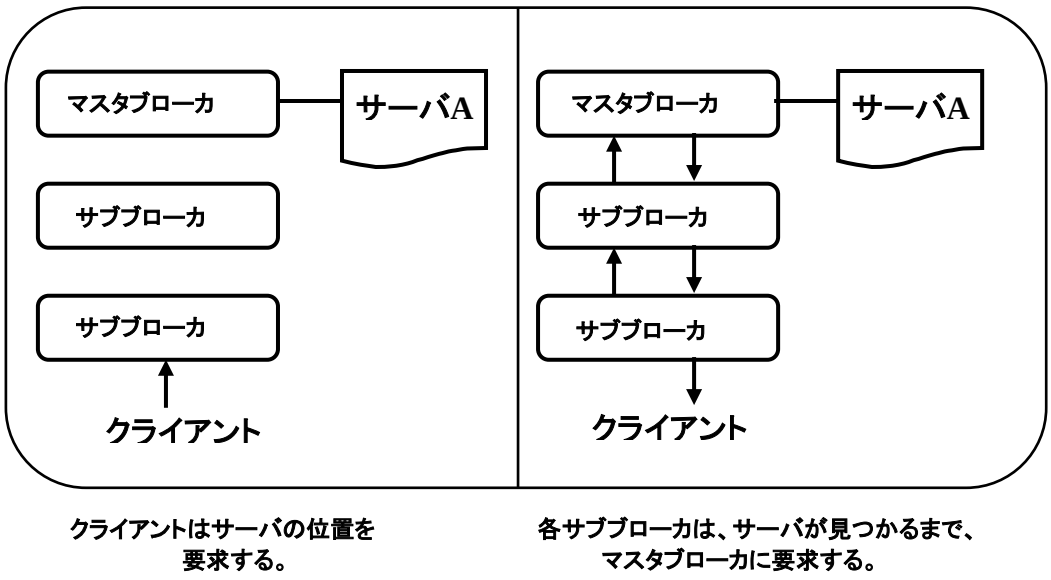


図 3.2： 親ブローカへのサーバ位置情報の要求

サブブローカの負荷分散

RPC 通信の負荷は、サブブローカを使用することによって分散することができます。

サブブローカがクライアントの要求を受けると、登録されているサーバ位置情報をクライアントに返します。クライアントから、サブブローカが保持していないサーバ位置情報の要求があ

れば、サブブローカはその親ブローカに接続し、指定されたサーバを問い合わせます。それ以降、サブブローカはそのサーバの情報を保持し、親ブローカに照会せずに、クライアントに位置情報を返します。これ以外にサブブローカが親ブローカと接続するのは、新しいサーバの位置を報告するときだけです。

他のサブブローカに登録されたサブブローカを起動することによって、階層を作り、RPC ルーティングの負荷を、必要な範囲に分散することができます。

ローカルサーバ

ローカルサーバとは、自分を直接管理しているブローカより上位の階層にあるブローカには位置情報を伝播しないサーバです。図 3.3 にその様子を示します。

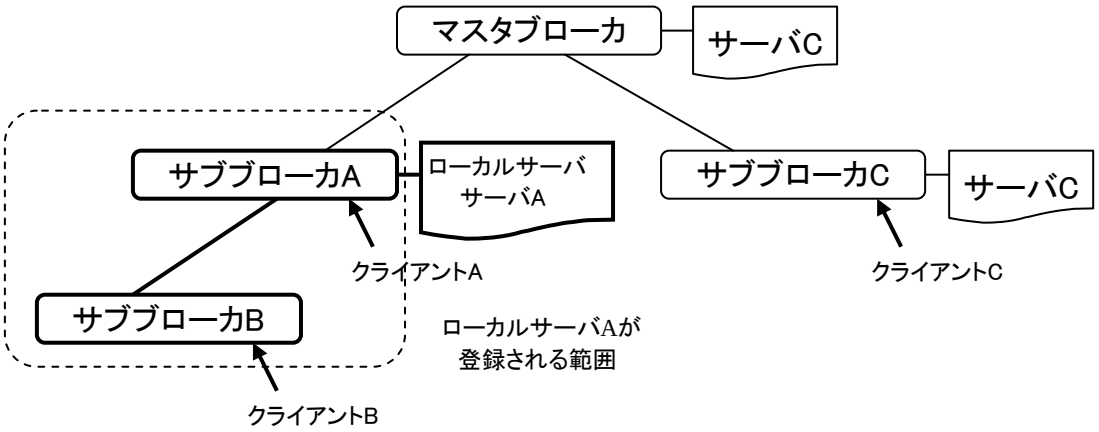


図 3.3 ： サブブローカとローカルサーバ

ローカルサーバの位置は、クライアント A とクライアント B にだけ認識されることになります。クライアント C には認識されません。なぜなら、ローカルサーバは、サブブローカ A より上位にあるマスタブローカに登録しないからです。

このような設定は、パフォーマンスやその他の理由から、地理的に離れているリソースの使用が望ましくないような WAN を起動する状況で便利です。

ローカルサーバの設定

サーバをローカルサーバにするには、サーバの環境ファイルに次の属性を追加するだけです。

```
DCE_LOCAL=1
```

マルチスレッド・サーバ(Multi Thread Server)

通常のサーバは、どの RPC 要求でも受け付けて実行しますが、マルチスレッド・サーバは 1 クライアントの要求を、1 スレッド内にて実行します。v5.2 より、新機マルチ・スレッド・テクノロジー (MTT) が搭載され、マルチスレッド・サーバは 1 クライアントの要求を、1 スレッド内にて実行することができます。 DCE_THREADED 環境ファイル属性の指定のみで実現し、この属性に指定した値がスレッド数最大値となり、同時可能クライアントアクセス数となります。例えば、DCE_THREADED=4 と指定した場合、スレッドサーバは最大4スレッド生成が可能であり、よって 4 クライアントからの同時アクセスを処理できます。許される数以上のクライアントが要求した場合、そのクライアントに対し、DCE_MAXRPCEXCEEDED の Nextra ランタイムエラー番号を返答します。 この Nextra ランタイムエラーは、C のようなプロシジャー言語では dce_errnum() API を使用して、またオブジェクト言語では Exception を利用して判定することができます。

DCE_DEDICATED 属性とこの属性両方が設定されている場合は、DCE_DEDICATED 属性の値が優先されます。

.

デディケイテッド・サーバ(Dedicated Server)

デディケイテッド・サーバの概要

通常のサーバは、どの RPC 要求でも受け付けて実行しますが、デディケイテッド・サーバは 1 クライアントの要求だけを実行します。これにより、サーバは「状態」を保持できるので、実際に最後の RPC がどこで終わったかを覚えておくことができるのです。

一般に、通常のサーバはポートで接続を待ちます。クライアントから要求があると接続が完了します。サーバがクライアントに接続されている間は、他の接続は受け付けられません。クライアントからの要求完了後に接続が切られると、サーバは別の接続を待つようになります。

デディケイテッド・サーバも同じように接続を待ちますが、クライアントに最初に接続されたときに子プロセスを作成します。子プロセスは、クライアントとのデディケイテッド接続をオープンしますが、ブローカには登録されません。親サーバがさらに接続を待つのに対して、子プロセスはそのクライアントと通信するだけです。クライアントと子プロセス間の接続は、クライアントがサーバに切断メッセージを送信するか、物理的なネットワークの切断によって接続が切られるまで続きます。子プロセスがクライアントとの接続の切断、またはクライアントプロセスの終了を検出すると、子プロセスは停止します。その間、オリジナルの親プロセス(デディケイテッド・サーバ)は、新しいクライアントからの接続を待ちます。

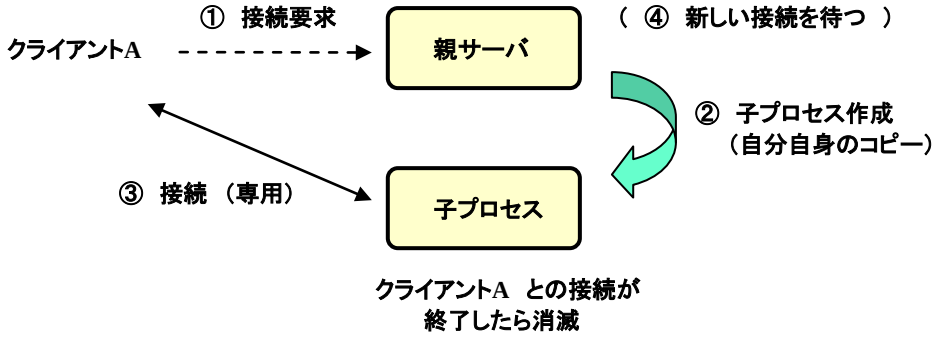


図 3.4 : デディケイテッド・サーバ使用時の処理の流れ

この方法を使用すると、親プロセスは常にクライアントからの接続を待ち、クライアントからの処理要求があり次第、子プロセスを生成し、その子プロセスがクライアントからの要求を処理します。

クライアントと子プロセス間の接続が、RPC の処理が終わらないうちに切れてしまうと、その時点で処理されていた RPC はエラーとなり、次の RPC は新しい子プロセスに接続されます。接続を失った子プロセスは停止します。

デディケイテッド・サーバの技術詳細

デディケイテッド・サーバの主な長所

デディケイテッド・サーバを使用するのは、次の 2 つの基本的な理由によります。

1. RPC におけるサーバの静的情報

これが、デディケイテッド・サーバの 1 番目の利点です。個々のクライアントが、他のクライアントと同一のサーバでサービスを受けることがないようにアプリケーションを設計したい場合に、デディケイテッド・サーバは良い方法です。

たとえば、特定の顧客の注文日付を検索する DB アクセス・サーバがあるとします。しかし、日付を全データの中から探すため、時間が 1 分以上かかってしまうとします。非デディケイテッド・サーバでは、1 回の RPC ごとに、少なくてもこの位の時間を要することになります。デディケイテッド・サーバでは、クライアントが特定の顧客の注文日付を問い合わせると、全てのデータを子プロセス内にキャッシングするので、2 回目以降はより敏速な応答を得ることができます。

DB アクセス・モジュールでデディケイテッド・サーバを使用することにより、カーソル機能を使用できます、ただし、これは DB カーソルではなくデディケイテッド・サーバ内でのカーソルになります。

2. 長時間を要する RPC からのより速い応答

RPC には、その前に何が起こっていても、処理に長い時間がかかるものもあります。たとえば、大量のデータ照会では、1 分以上もかかることがあります。さらに、非デディケイテッド・サーバを使用すると、複数クライアントからのサーバへの要求はキューが溜まることになり、各クライアントへの処理レスポンスが悪化します。RPC 毎に 30 秒かかるサーバに 6 つのクライアントが同時にアクセスすれば、最後のクライアントは 180 秒後にレスポンスを受けることになります。

サーバをデディケイテッド・サーバにすることによって、クライアントはサーバでのキュー待ちがなくなるので、全くアクセスがない非デディケイテッド・サーバへのリクエストに要する時間とあまり変わらなくなることになります。デディケイテッド・サーバでは、ブローカに親サーバ（デディケイテッド・サーバ）が登録してあり、クライアントの最初の RPC 実行を待っていることを思い出してください。各クライアントが、最初の RPC を実行すると、親サーバはそのクライアントのために、子（サーバ）プロセスを作成します。子プロセス作成に要する時間は非常に短いものです。作成された子プロセスが、クライアントからの RPC に対応するのです。同一クライアントが実行する次の RPC は、自動的に同じ子プロセスにアクセスします。この子プロセスは、このクライアントのみに使用され、他のクライアントに使用されることはありません。したがって、応答時間は、常にサーバが実際に実行する時間だけになります。

デディケイテッド・サーバの短所

デディケイテッド・サーバの主な短所は、各クライアント毎に 1 つ生成されるプロセスが、全体として非常に多くなる可能性があるため、親サーバと子サーバを実行しているマシンにとっては、パフォーマンスの点で影響が大きいことです。

Nextra の以前のリリースで、発生するもう 1 つの問題は、各子プロセスごとにログファイルが 1 つ作成されて、全体として非常に多くなり、ファイルシステムを占有してしまう可能性があるということです。しかし、現在のリリースではログファイルの数と詳細を設定できるため、もはや問題にはなりません。詳細については、「デディケイテッド・サーバのログファイル」を参照してください。

デディケイテッド・サーバとそのクライアントのコーディング

現在、デディケイテッド・サーバがサポートされているのは、アプリケーションサーバと DB アクセス・サーバです。その他の言語のサポートは、今後追加される予定です。デディケイテッド・サーバの開発、コンパイル、および実行の手順は通常のサーバの手順と同じです。

サーバをデディケイテッド・サーバにするには、次の手順を行います。

1. 環境ファイルに属性 DCE_DEDICATED=*N* を設定する。*N* はデディケイテッド・サーバが同時に持つことのできる、実行状態の子プロセスの最大数である。

2. オプションで、環境ファイルに属性 DCE_DEDICATEDLOGFILE を使用して、子プロセスが独自のログファイルを作成するかどうかを指定できる。

   詳細については、『リファレンス』の「DCE_DEDICATEDLOGFILE」を参照。

   この手順によって、どのサーバでもデディケイテッド・サーバにすることができますが、常に同じクライアントと通信するという長所を利用するためにサーバをコーディングすることも可能です。

   デディケイテッド・サーバと適切に通信するには、クライアント側は 1 箇所の変更が必要となります。

3. そのサーバの使用が完全に終了したときに(各 RPC 終了後ではなく)、dce_dedDisconnect()を呼び出して、クライアントはデディケイテッド・サーバを切断しなければならない。

   デディケイテッド・サーバを使用するための他の機能は、スケルトンに自動的にコーディングされます。どのクライアントについても、各 RPC の後で DCE エラーチェック機能を使用して、RPC が正常に終了したことを検証する必要があります。もしデディケイテッド・サーバへの接続に失敗し、クライアントが新しい子プロセスに接続した場合でも、新しい子プロセスは失敗した子プロセスと同じ状態にはなりません。DCE エラーチェック機能は、クライアントにこの情報を提供します。そして、クライアントは、最初から現在の関数を処理するか、またはユーザに警告を与えて、このエラーに対して適切に対応しなければなりません。

   デディケイテッド・サーバは、非デディケイテッド・サーバよりも管理が複雑な構造になっているので、設計の際には慎重に検討しましょう。

   dce_dedDisconnect(), dce_isDedicated(), dce_server_is_ded()の詳細については『リファレンス』の「Nextra API」の章を参照してください。

デディケイテッド・サーバの初期化

rrpcmake の初期化関数オプション(-i オプション)は、デディケイテッド・サーバでも実行できます。初期化とは、サーバがクライアントからの RPC を実際に受け付ける前に、サーバの実行環境を準備することです。 **rpcmake** の-i オプションを使用すると、サーバが Nextra 環境を設定した後、クライアントから RPC を実際に受け付け始める前に、サーバが実行する初期化関数の名前を指定することができます。たとえば、クライアントからの RPC を受け付ける前に、サーバがリモートマシンにログインして、別アプリケーションにアクセスするなどです。

デディケイテッド・サーバでは、**rpcmake** の-i オプションは、親サーバに初期化関数も終了関数も実行させません。その代わりに、それぞれの子サーバが起動直後に初期化関数を実行し、停止の直前に終了関数を実行します。

サーバがデディケイテッドであることの確認

既に実行されている特定のサーバを使用するときに、それがデディケイテッド・サーバであることを確認する場合を検討してみましょう。現在のサーバ名を引数にして API 関数(dce_isDedicated())を呼び出すことにより確認できます。関数が true を返した場合には、サーバがデディケイテッド・サーバであるということです。

デディケイテッド・サーバのログファイル

デディケイテッド・サーバでは、データを親サーバ(デディケイテッド・サーバ)についてのみロギングするか、または親サーバとそれぞれの子サーバについてロギングすることができます。子サーバのロギングをオンにした場合には、起動するそれぞれの子サーバは、<サーバのログファイル名>_d というディレクトリに、新しいログファイルを作成します。個別のファイル名は、サーバのプロセス ID に基づきます。

- 子サーバのログインを設定するために、環境ファイル中の DCE_DEDICATEDLOGFILE 属性を使用する。
- DCE_LOG 属性は親サーバのログファイルの名前を付ける。子サーバログファイルは接尾に_d が付き、DCE_LOG 属性の値の名前の付いたディレクトリに保存される。各ログファイル名は、作成した子サーバのプロセス ID 番号を含む。

どのログファイルが正しいか?

クライアントが接続しているデディケイテッド・サーバのプロセスが不明な場合には、DEBUG のデバッグレベルでクライアントを実行すると、それを発見することができます。このレベルでは、クライアントは、現在通信しているサーバプロセスのプロセス ID をログファイルに出力します。その後、このプロセス ID と、サーバマシン上の子サーバのログファイルの名前とを照合することができます。

サーバの喪失

クライアントとの接続時に、デディケイテッド・サーバが停止すると、次の RPC は PEERERR0R を返し、その後の RPC で新しい子プロセスに接続されます。

新しい子プロセスがすぐに提供され、停止した子プロセスと同じ状態を持つ必要がない場合、この処理は望ましいものになります。PEERERR0R メッセージをクライアントが受け取った場合、前のサーバの状態を何らかの方法で回復するか、トランザクションを再実行することをユーザに通知するかについて、クライアントが整合性を扱う方法を決定することができます。

SIGCHILD のトラッピング

子プロセスは、親プロセス(デディケイテッド・サーバ)と全く別のプロセスとして起動されている為、SIGCHILD 割り込みをデディケイテッド・サーバでトラッピングすることはできません。

バリアブル・ネームド・サーバ

バリアブル・ネームド・サーバの概要

Nextra ユーザは、並列サーバのセットに属するどのサーバにでも、RPC を使って直接接続することができます。ただし、このセットにあるサーバは、同じ IDL ファイルから生成されたものでなければなりません。並列サーバは同じ機能とインタフェースを提供するため、概念的には「同一」と考えられますが、論理的にはインタフェース名で識別されます。

バリアブル・ネームド・サーバの例として、次のような状況を検討してみましょう。ある国営企業のデータは、同じデータモデルで地域ごとの DB に格納されています。このため、ある地域の DB に対して実行できる照会は、他の全ての DB に対しても実行できます。
このとき、エンドユーザがどの DB にアクセスするか選択できるということは重要です。地域が NE、SE というように分類されている場合には、複数の IDL ファイルとクライアント・スタブを作成して、同じ機能を持つこれらのサーバにアクセスすることは非効率です。
バリアブル・ネームド・サーバを使用すれば、IDL ファイルとスケルトンをそれぞれ 1 つだけ用意すればよく、起動時にそれぞれのサーバ名を指定できます。サーバ名は、追加引数によって指定され、クライアントプログラムから直接呼び出すことができます。つまり、この例は図 3.5 で示すような流れになり、クライアントは特定の照会を適切な地域の DB アクセス・サーバに送ります。

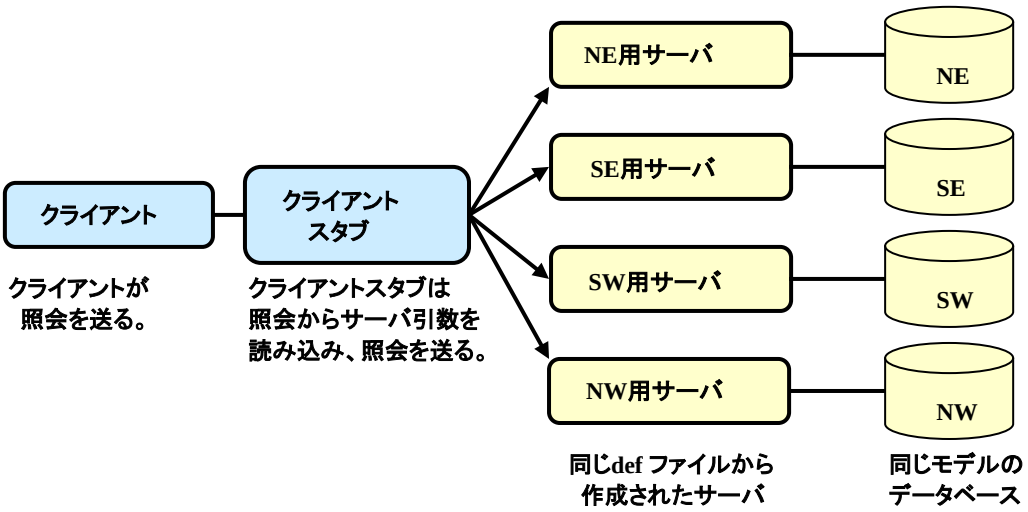


図 3.5： 複数の同一サーバに対する RPC

どの地域のサーバに送られる要求も、同一の形式をもつ照会であることに注目してください。要求の接続先は、RPC の引数の 1 つにサーバ名を指定することにより選択できます。 地域の DB は、全て同一のモデルにしたがっているため、これらの DB にアクセスするために使用するサーバは、インタフェース名、つまりサーバの名前が変数として指定されていれば、全て同じ IDL ファイルから生成することができます。このようにして生成されるサーバは、名前が異なるだけで同一のプログラムであるため、全て同一の照会を処理することができます。

バリアブル・ネームド・サーバを使用する場合

名前(および接続する DB)が異なるだけで、機能が同等のサーバがあり、特に名前が地域や部門のように任意の分類である場合には、バリアブル・ネームド・サーバが便利です。

この機能は、インタフェースの関数名とパラメータが同じである場合以外には、使用してはいけません。同じ入力を受け付け、同じ出力を返すような複数のサーバでは、この機能を使用すれば個々のサーバを保守する問題を回避し、1 つのサーバを多くの異なる名前で再利用することができます。この機能を使用すれば、バリアブル・ネームド・サーバ 1 組について必要なスタブは 1 つだけなので、クライアントプログラムも簡潔になります。

バリアブル・ネームド・サーバの技術詳細

可変インタフェース名の作成

バリアブル・ネームド・サーバを作成するには、IDL ファイルで interface 文に変数を指定しなければなりません。そのためには、インタフェース名の前にドル記号($)を挿入して、インタフェース名を変数にします。

```
interface $varname
```

DB アクセス・サーバを作成するには、**SQLMake** を使用しなければなりません。**SQLMake** GUI を使っても、コマンドラインでコマンドを起動してもかまいません。GUI を使用する場合は、最初のテキストフィールドに SQL ファイル名を、サーバ名のテキストフィールドに$*varname* を入力します。IDL ファイルを作成するために、コマンドラインで **SQLMake** を使用する場合は、-s フラグを付けた引数として可変サーバ名を使用します。UNIX プラットフォームでは、可変サーバ名はドル記号で始め、シングルクォーテーションで囲みます。

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

Windows ではクォーテーションは使用しません。

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

クライアント・スタブが生成されるときに、各スタブ関数には追加引数(パラメータリストの最初のもの)があり、クライアントが接続する実際のサーバのインタフェース名を指定するために使用されます。

IDL ファイルの例

可変インタフェース名によってサーバを定義する IDL ファイルのサンプルを示します。

```
# 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 ファイルの比較

次の比較例で、固定インタフェースの IDL ファイルから生成されたコードと、可変インタフェースの IDL ファイルから生成されたのコードの違いを説明します。このページでは、2 つの IDL ファイルそのものを比較します。次のページでは C クライアント・スタブとその違いを説明します。

まず、固定名サーバの IDL ファイルを示します。

```
# RPC interface definition file for: cserver
[uuid(00.06.09.00.00.00.00.06) version(1.1)]

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

}
```

可変インタフェースの IDL ファイルは次のようになります。

```
# RPC interface definition file for: $server
[uuid(00.06.09.00.00.00.00.07) version(1.1)]

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

}
```

クライアント・スタブの比較

この例では、固定名サーバのために SQLMake が生成した C クライアント・スタブの一部を示します(dce_findserver()に対する呼び出しで、引数が固定文字列 cserver である点に注意してください)。

```
/*#########################
# 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);

}
```

この例では、バリアブル・ネームド・サーバのために RPCMake が生成した C クライアントプログラム(dce_findserver()への呼び出しの引数は ode_server)について、違いをボールド体で示します。

```
/*#########################
# 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);

}
```

クライアントプログラムでは、追加引数(アクセスするサーバ名)を指定して、各 RPC を呼び出す必要があります。このパラメータは、上記のスタブが示すように、各 RPC の最初のパラメータでなければなりません。

不適切なサーバ名

クライアントが未登録のサーバ名を送った場合、つまり、送った名前のサーバがない場合は、サーバが存在しない場合と同じエラーとなります。

並列バリアブル・ネームド・サーバ

オープン分散環境の他の場合と同様、各サーバのコピーをいくつか作っておくことができます。(たとえば、SW サーバ 3 つ、NW サーバ 15 など。) 呼び出し時に名前を指定することによって、RPC はどのサーバにも接続できます。

同じ名前のバリアブル・ネームド・サーバ

バリアブル・ネームド・サーバは、ユニークな名前を付ける必要はありません。ただし、2 つの異なった可変名を持つサーバに同じ名前を付けると、アプリケーションが適切に作動しなくなります。異なるインタフェースを提供するバリアブル・ネームド・サーバについては、アプリケーションが適切に機能するためには、異なる名前を付けなくてはなりません。

手動による並列バリアブル・ネームド・サーバの起動

例の続きとして、同じサーバを起動してみましょう。

各 NE サーバについて、次の行を指定します。

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

各 SW サーバについて、次の行を指定します。

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

全てのサーバは同じ IDL ファイルを使用するため、サーバは全て同じ SQL ファイルを使用しなくてはならないことになります。

-s コマンドラインオプションの代わりに、サーバの環境ファイルに、DCE_SERVERNAME=*server_name* 属性を設定することによって、サーバ名を指定してもかまいません。

ネーミングサーバ

-s コマンドラインオプション、または環境ファイルの DCE_SERVERNAME 属性を使用して固定名サーバの名前を付けることはできません。これらのオプションはバリアブル・ネームド・サーバ用のものです。

バリアブル・ネームド・サーバを最初から作成する場合

可変名を使用するサーバを開発者自身が作成する場合は、名前の異なるサーバ間で、目に見える違いがないようにすべきです。違いは非常に小さいものでなくてはなりません。

パフォーマンス向上

運用上において、パフォーマンスを向上させるテクニックについて説明します。

マルチスレッド(Multi Thread Server)

v5.2 より、マルチスレッド・サーバが使用できるようになりました。更なるパフォーマンスを重視するアプリケーションの場合、並列サーバ(複数サーバプロセス起動)機能を同時に利用することをお勧めしております。また、何らかの理由でマルチスレッド・サーバが利用できない場合は、Dedicated Server(デディケイテッド・サーバ)機能を利用することをお勧めします。

サーバプロセスへのリクエストの際、TCP Backlog Queue(キュー)待ちになりレスポンスが上がらない場合に、レスポンススピードを上げる方法

サーバプロセスが TCP ポートにて Queue(キュー)待ちを行うことにより、パフォーマンスが上がらないことがあります。これを解消するために、Nextra ではマルチスレッド・サーバ(Multi Thread Server)、または Dedicated Server(デディケイテッド・サーバ)を利用することができます。これにより、クライアントプログラムからのリクエストにサーバプロセスが瞬時に反応、処理を行うことができます。

クライアントがサーバを認識した後、ネットワークの不調により通信が切断される。このとき、クライアントがタイムアウトを判定するまでの待ち時間があるが、これを短くする方法

クライアント環境ファイル(client.env)において以下の環境ファイ属性を設定します。例えばタイムアウトまでの判定を 10 秒としたい場合、

DCE_CLN_TIMEOUT=10

のように設定します。

DCE_CLN_TIMEOUT は、クライアントがサーバに送った RPC の応答を待つ時間(単位:秒)を設定します。サーバからのレスポンスが設定した時間(上の例の場合は 10 秒)までに得られない場合、クライアント側から接続を切ります。

上記現象発生時に再度接続を試みる方法(C 言語のプログラムでの例)

```
#define RETRY 3 /* 3 回接続を試みる場合 */

for (i=0;i<RETRY;i++){
     result=RPC_function(argument); /* RPC の結果を result に入れる */

     /* 正常に終了したら再接続のループから抜ける */
     if (dce_errnum()==0) break;

    /* RPC のエラーのうちサーバに接続できたが応答がないというエラー
```

```
            以外 のときはエラーメッセージを出力して終了 */
        else if (dce_errnum()!=DCE_RPCTIMEOUT){
             printf("Error: %s\n", dce_errstr());
             exit(1);
        }

        if (i==RETRY-1){ /* RETRY 回接続を試みてもだめだった場合 */
             printf("Error: %s\n", dce_errstr());
             exit(1); /* エラーメッセジを出して終了 */
        }
        sleep(20); /* 20 秒待って再接続。秒数はサーバの処理時間を
                      考慮して調整してください。 */
    }
```

＊上記プログラム中の「DCE_RPCTIMEOUT」はサーバに接続はしたが、応答が帰ってこない場合の RPC エラーのエラー番号を示します。

Nextra のネットワーク言語

Nextra は、ロケールとして EUC（Unix のみ）と SJIS をサポートしております。内部的には、Nextra のプロセス間では SJIS をネットワーク言語として採用しております。もし、EUC 環境で Nextra プロセスを起動していて、データを N/W に書き出す場合は、EUC->SJIS にデータ言語変換を行います。逆に、データを N/W から取り出す場合は、SJIS->EUC へのデータ言語変換を行います。

よって、SJIS 環境で Nextra プロセスを起動している場合には、データ言語変換は起こらないこととなり、よって、多少なりともデータスルーのパフォーマンスが向上することになります。

データ言語変換、非変換は、LANG 環境変数指定を元に、Nextra のランタイムライブラリ内にて行われます。

なお、Nextra5 以降では、UTF8 もネットワーク言語としてサポートされます。

ログファイルの DEBUG（デバッグ）レベル

運用に入ったクライアントやサーバアプリケーション、そして Broker のログファイルの DEBUG （デバッグ）レベルを下げることによりパフォーマンスを挙げることができます。『リファレンス』「第 2 章ファイル仕様」のログファイルを参照してください。

AppMinder 利用時のエラーファイル

運用に入ったアプリケーションを AppMinder を利用して管理する場合、もはやエラーファイルを生成する必要は殆んどありません。AmViewer より、Broker やアプリケーションに対して、"Edit"ボタンから "File"をクリックして、"Error File Name"に「nul」と指定してください。"Error File Directory"は任意で構いません。 指定先が Unix の場合は、"Error File Name"に「null」と指定し、"Error File Directory"には、「/dev」と指定してください。

これにより、エラーファイルが生成されなくなり、リソース確保に貢献できます。

AppMinder 利用時に、全てのプロセスが「Pinged」になるまでの時間を変更する方法

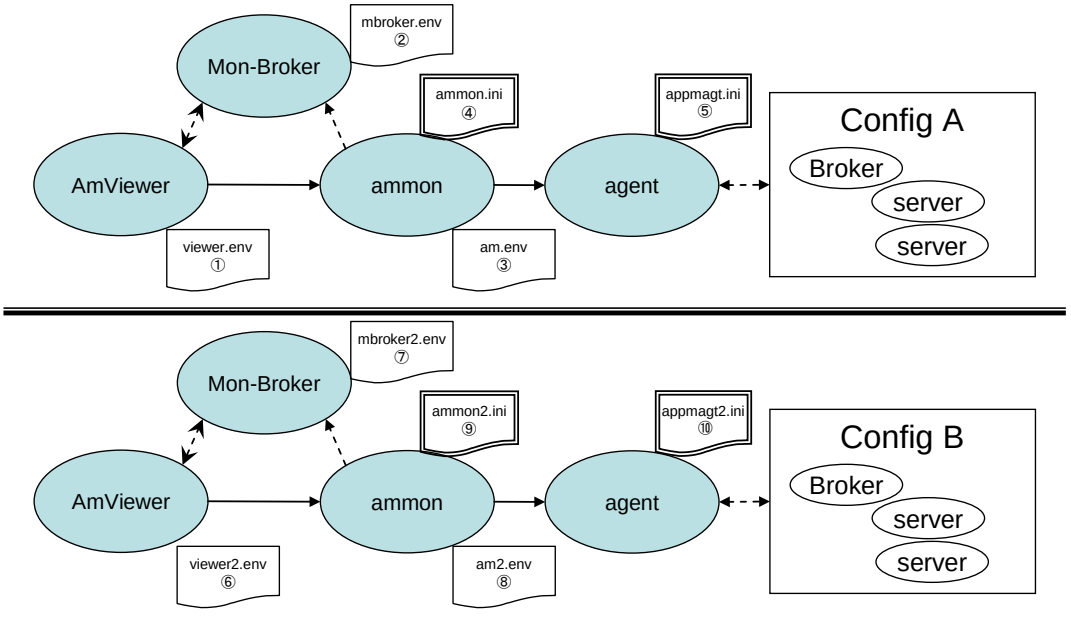
appmagt.ini ファイルの中の AMAGENT_PROC_START_PAUSE の値を変更してください。初期値は 100（秒）です。

AppMinder 利用時に、エージェント（appmagt）の管理サイクルを短くする方法

appmagt.ini ファイルの中の AMAGENT_MGMTINTERVAL の値を変更してください。初期値は 120（秒）です。

複数セットのアプリケーションを 1 システム上で管理する方法

以下の図を参考にしてください。



各ファイルにおける注意点

- 1 つ目のコンフィギュレーション

①viewer.env
```
DCE_BROKER=localhost,9090
DCE_DEBUGLEVEL=NONE,DEBUG
```

②mbroker.env
```
DCE_BROKER=localhost,9090
DCE_DEBUGLEVEL=NONE,DEBUG
```

③am.env
```
DCE_BROKER=localhost,9090
DCE_DEBUGLEVEL=NONE,DEBUG
DCE_SERVERPORT=8001
DCE_CLN_TIMEOUT=<X>秒
```

④ammon.ini
```
AMMON_AGTPORT=7001
AMMON_SERVERARGS=-e am.env
AMMON_TRPCLOGLEVEL=NONE,DEBUG
```

⑤appmagt.ini
```
AMAGENT_PORT=7001
```

- 2 つ目のコンフィギュレーション

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

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

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

⑨ammon2.ini
```
AMMON_AGTPORT=7002
```

```
AMMON_SERVERARGS=-e am2.env
AMMON_TRPCLOGLEVEL=NONE,DEBUG
```

⑩appmagt2.ini

```
AMAGENT_PORT=7002
```

注意点としては、各ファイルの PORT 番号は任意です。
DCE_CLN_TIMEOUT は、環境により適切な値を指定してください。

Windows の場合のスタートバッチファイルの例

```
@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 の場合のスタートバッチファイルの例

```
#!/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 &
```

パケットサイズの変更

DCE_PACKETSIZE 環境ファイル属性により、TCP レイヤーとやりとりするネットワークパケット送受信バイト数を指定します。送受信メッセージがデフォルトパケットサイズ（1460 バイト）を超えるデータを多く扱う場合は、値を変更することにより、送受信スピードが速くなります。

受送信ソケットバッファーサイズの動的指定

DCE_SO_RCVBUF_LEN 環境ファイル属性により、受信ソケットバッファーの最大サイズをバイト単位で指定します。SOCK_DGRAM ソケットでは、受信バッファーのサイズによっては、ソケットの受信可能な最大メッセージサイズが制限されることがあります。デフォルトおよび最大値は OS 依存。

また、DCE_SO_SNDBUF_LEN 環境ファイル属性にて、送信バッファーの最大サイズをバイト単位で指定します。SOCK_STREAM ソケットでは、送信バッファーのサイズによっては、アプリケーションがブロックされる前に送信のためにキューに入れることができるデータの量が制限されることがあります。SOCK_DGRAM ソケットでは、送信バッファーのサイズによっては、アプリケーションがソケットを通じて送信可能なメッセージの最大サイズが制限されることがあります。デフォルトおよび最大値は OS 依存。

リンガする（待機する）かどうか

DCE_SO_LINGER 環境ファイル属性（デフォルト=10 ミリ秒）により、ソケットがクローズされたときに送信ソケットバッファーに未転送データがある場合、アプリケーションが「リンガ」する（待機する）かどうかを制御します。

Nagle アルゴリズムを無効にする

DCE_TCP_NODELAY 環境ファイル属性（デフォルト=0）に 1 を指定すると、Nagle アルゴリズムを無効にします。

送信データの非分割について

1RPC に含まれる送信データが DCE_PACKETSIZE 環境ファイル属性で指定されたバイト数よりも大きい場合に、分割せずに送信することにより送信スピードを短縮することが可能となります。DCE_PACKET4NOFRAGMENT 環境ファイル属性（デフォルト=0）に 1 を指定することで有効にするこができます。

ミッションクリティカルシステムの為の、TCP 層のチューニング

Nextra Tuning Guide（英語版のみ）をご参照ください。

ご注意

商標権に関する注意

Nextra 製品は、全て Inspire International Inc. の商標または登録商標です。その他記載のブランドおよび製品名は、該当する会社の商標または登録商標です。

著作権に関する注意

Nextra
運用／設定ガイド

2015 年 11 月 8 日　v6.5 1st Edition
2013 年 3 月 5 日　図 2.1 DCE_TRANSCRIPT へ変更
2011 年 9 月 15 日　v6 1st Edition
2008 年 8 月 12 日　v5 2nd Edition
2007 年 4 月 18 日　第 3 版発行
2006 年 12 月 13 日　通信障害に備えたタイムアウト（TO）の設定追加
2006 年 11 月 14 日　複数セットのアプリケーションを 1 システム上で管理する方法の追加
2006 年 10 月 20 日　Backlog Queue（バックログ・キュー）の設定
2006 年 8 月 28 日　パフォーマンス向上追加
2005 年 5 月 19 日　プライベートネットワークの外にあるクライアントからのアクセスや、サーバマシンに 2 つ以上の IP アドレスが割り振られている場合についての追加
2004 年 7 月 19 日　第 2 版発行
2003 年 4 月 18 日　初版発行

著者　Inspire International Inc.