# サーバー・クラスター対応の Java アプリケーションを構築する

## Apache ZooKeeper と LinkedIn の Norbert によって、分散型エンタープライズ Java アプリケーションでのサーバー·グループのコーディネーションを容易にする

Mukul Gupta
Senior Technical Architect
CGI

Paresh Paladiya
Senior Technical Architect
CGI

2014年 2月 20日
(初版 2012年 12月 13日)

極めてスケーラブルな Java エンタープライズ・アプリケーションを開発する場合、サーバーをクラスタリングするのが当たり前のことになってきています。けれども今のところ、Java EE ではアプリケーション・レベルでサーバー・クラスターに対応できるようにはなっていません。この記事では、分散型エンタープライズ Java アプリケーションにおいて、サーバー・グループのコーディネーションをするために、Apache ZooKeeper と LinkedIn の Norbert という 2 つのオープンソース・プロジェクトを利用する方法を Mukul Gupta と Paresh Paladiya が紹介します。
2014年 2月 20日 — developerWorks の新しい記事「ZooKeeper fundamentals, deployment, and applications」を参考文献の項目に追加しました。

最近のエンタープライズ・アプリケーションの多くは、分散された一連のプロセスとサーバーが協調するという形で実現されています。よく使われている Java エンタープライズ・サーバーのほぼすべてにおいて、例えば、サーバーのクラスタリング機能を Web リクエストに対して使用できるようになっています。さらに、サーバーの重み付けや構成の再ロードなどといった限定的な構成オプションが用意されている場合もあります。

大抵の Java エンタープライズ・サーバーにはクラスタリングのサポートが組み込まれています。けれども、そのサポートは、アプリケーション・レベルでのカスタム・ユース・ケースで直ちに使用できる状態にはなっていません。ソフトウェア開発者としては、分散されたタスクのコーディネーションが必要となるユース・ケースや、マルチテナント・アプリケーションのサポート

が必要となるユース・ケースにどのように対処すればよいのでしょう (「マルチテナント・アプリケーション」とは、サーバー・クラスターやサーバー・グループ全体のサブセットで個々のインスタンスを分離しなければならないアプリケーションのことです)。このようなタイプのユース・ケースの場合、アプリケーション・ソフトウェアのレベルで、そしてできれば高い抽象化レベルで、グループのコーディネーション機能を使用可能にする方法を見つけなければなりません。

この記事では、グループ・メンバーシップ機能とグループ管理機能を分散型 Java アプリケーションに組み込む際の指針を紹介します。Spring Integration をベースにシミュレートした Java アプリケーションを出発点に、Apache ZooKeeper と LinkedIn の Norbert という 2 つのオープンソース・プロジェクトをベースとしたサーバー・クラスター抽象化レイヤーを統合します。

### Apache ZooKeeper および LinkedIn の Norbert について

Apache ZooKeeper は、分散型アプリケーションにサーバー・グループ・コーディネーション機能を提供するオープンソースのプロジェクトです (「参考文献」を参照)。LinkedIn によって開発された Norbert は、ZooKeeper のサーバー・グループ管理機能とリクエストのルーティング機能をさらに高い抽象化レベルで公開します。これにより、Java アプリケーション開発者はこれらの機能をより利用しやすくなります。

## サーバー・クラスタリングの概要

一般に、サーバー・クラスター対応のアプリケーションには、少なくとも以下の機能のいくつかが必要になります。

- **状態の維持管理機能と照会機能を備えたグループ・メンバーシップ**: アクティブな一連のサーバーに処理を分散するには、リアルタイムのグループ・メンバーシップが必要です。そしてグループ・メンバーシップを管理するには、アプリケーションでプロセス/サーバー・グループを設定して、そのグループ内のすべてのサーバーの状態を追跡できなければなりません。さらに、あるサーバーが停止または起動したときに、アクティブ・サーバーに対してそれを通知できることも必要となります。アプリケーションは高い可用性のサービスを確保するために、クラスター内のアクティブなサーバーでのみサービス・リクエストのルーティングおよびロード・バランシングを行います。
- **プライマリー・プロセスまたはリーダー・プロセス**: クラスター内の 1 つのプロセスだけがグループをコーディネートする役割を担い、サーバー・クラスター全体での状態同期を維持します。リーダー・プロセスを選択するメカニズムは、「分散合意」として知られる一連の問題の特殊なケースです (分散合意の有名な問題には、2 相コミットや 3 相コミットがあります)。
- **タスクのコーディネーションとリーダー・サーバーの動的選出**: アプリケーション・レベルでは、「リーダー・サーバー」がタスク・コーディネーションを担当し、クラスター内の他の (フォロワー) サーバーにタスクを配布します。リーダー・サーバーを決めることで、サーバー間で競合が発生しないようにするというわけです。サーバー間で競合が発生すると、適格なタスクが実行されるようにするために、何らかの形の相互排除機能やロック機能が必要になります (このよう競合が発生するのは、例えば、サーバーが共通データ・ストアのタスクをポーリングしている場合です)。さらに、動的にリーダーを選出することで、分散処理に信頼性がもたらされます。リーダー・サーバーがクラッシュしても、動的なリーダー選出によって新しいリーダーを選出することにより、アプリケーション・タスクの処理を続行することができます。

- **グループ通信**: クラスター対応アプリケーションは、サーバー・クラスター全体で構造化データおよびコマンドの効率的なやりとりを促進できなければなりません。
- **分散ロックおよび共有データ**: 分散型アプリケーションは、必要に応じて分散ロックや共有データ構造体 (キューやマップ) などの機能にアクセスできなければなりません。

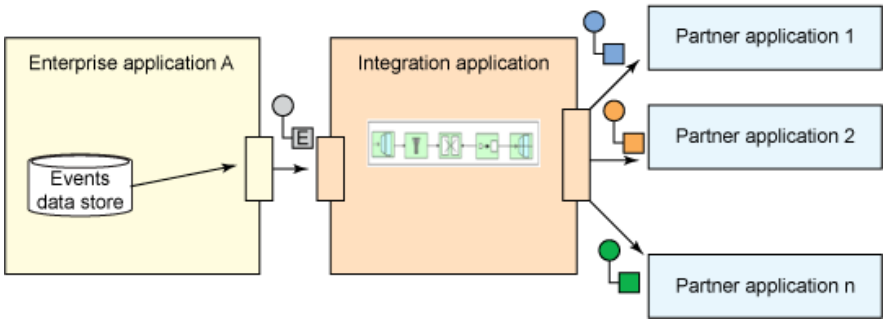
## サンプル・ユース・ケース: Spring Integration

この記事で取り上げる代表的なユース·ケースは、エンタープライズ・アプリケーション統合 (Enterprise Application Integration: EAI) のシナリオです。これから、Spring Integration をベースにシミュレートしたアプリケーションでこのユース・ケースに対処します。このアプリケーションの特徴および要件は以下のとおりです。

1. シミュレートしたソース・アプリケーションは、通常のトランザクション処理の一環として統合関連のイベントとメッセージを生成し、それらをデータ・ストアに保管します。
2. 統合イベントとメッセージは、一連の分散された Java プロセス (サーバー・クラスター) で処理されます。これらのプロセスは同じサーバー・マシンで実行することも、十分な性能を持つネットワークに接続された複数のマシンに分散することもできます。サーバーのクラスタリングは、スケーラビリティーのためにも高可用性のためにも必要です。
3. それぞれの統合イベントは、クラスター・メンバー (つまり、所定の JVM) によって一度だけ処理されます。出力メッセージは、イントラネットまたはインターネットのいずれか該当する経路を介してパートナー・アプリケーションとの間で通信されます。

図 1 に、シミュレートしたソース・アプリケーションから送信される統合イベントおよびメッセージ処理フローを示します。

### 図 1. Spring Integration ベースのサンプル・アプリケーションの概略図




## ソリューションを設計する

このユース・ケースのソリューションを開発する際の最初のステップは、実行する統合アプリケーションをサーバー・クラスターに分散させることです。これにより、処理のスループットが向上し、高可用性とスケーラビリティーの両方が確保されます。1 つのプロセスに障害が発生しても、アプリケーション全体の処理は停止しません。

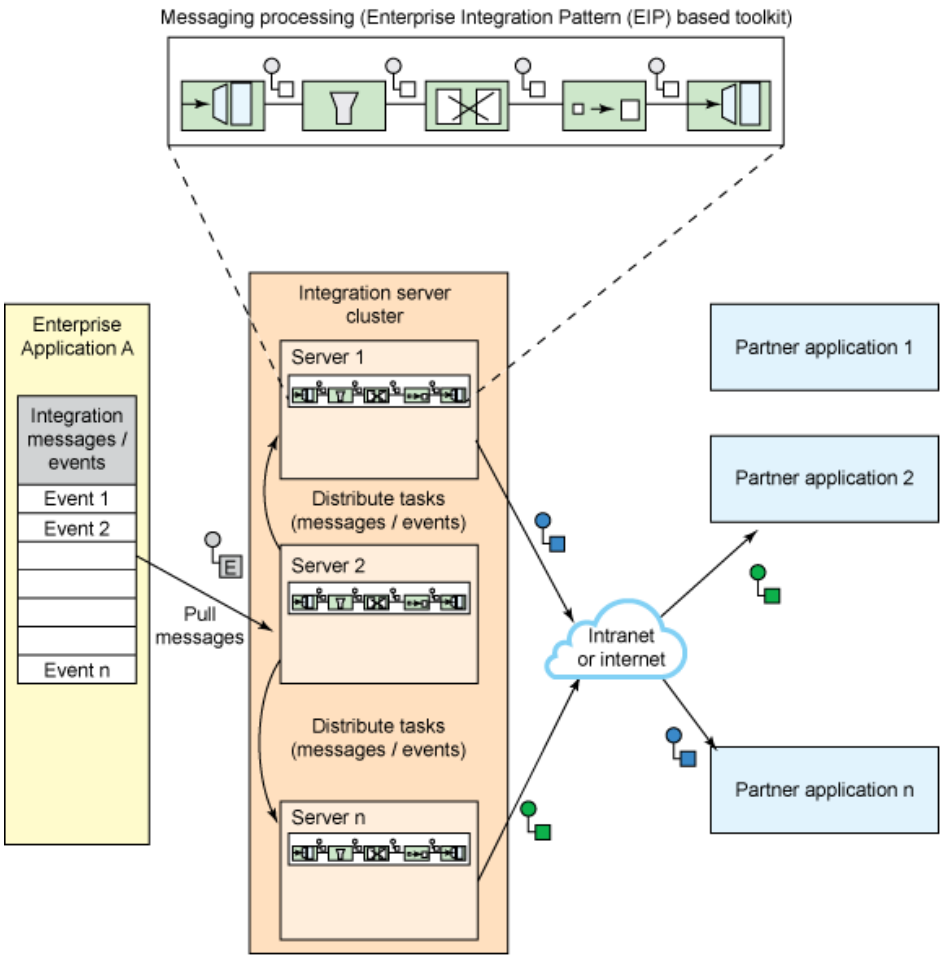
分散された統合アプリケーションは、アプリケーションのデータ・ストアから統合イベントを取り出します。サーバー・クラスターを構成するサーバーのうち、1 つのサーバーだけが、適切な

アプリケーション・アダプターを介してイベント・ストアからアプリケーション・イベントを抽出し、これらのイベントをクラスター内の残りのサーバーに配布して処理させます。つまり、この 1 つのサーバーが、クラスターを構成する残りのサーバーにイベントを配布する (タスクを処理させる) 「リーダー・サーバー (タスク・コーディネーター)」の役割を担うということです。

統合アプリケーションをサポートするサーバー·クラスターのメンバーは、その構成時に既知となります。新しいサーバー・メンバーの起動時や、サーバーがクラッシュしたり、停止したりした時点で、稼働中のすべてのサーバーにクラスター・メンバーシップ情報が動的に配布されます。同様に、タスク・コーディネーターの役割を担うサーバーも動的に選出されます。タスク・コーディネーター役のサーバーがクラッシュしたり、使用不可になったりすると、残りの稼働中サーバーのなかから協調的に代わりのリーダー・サーバーが選出されます。統合イベントを処理するには、エンタープライズ統合パターン (Enterprise Integration Patterns: EIP) をサポートしている多くのオープンソース Java フレームワークのうちのいずれかを使用することができます (「参考文献」を参照)。

図 2 に、このユース・ケースのソリューションの概略図およびコンポーネントを示します。コンポーネントについては、次のセクションで詳しく説明します。

## 図 2. ユース・ケースのソリューションの概略図とサーバー・クラスターのコンポーネント

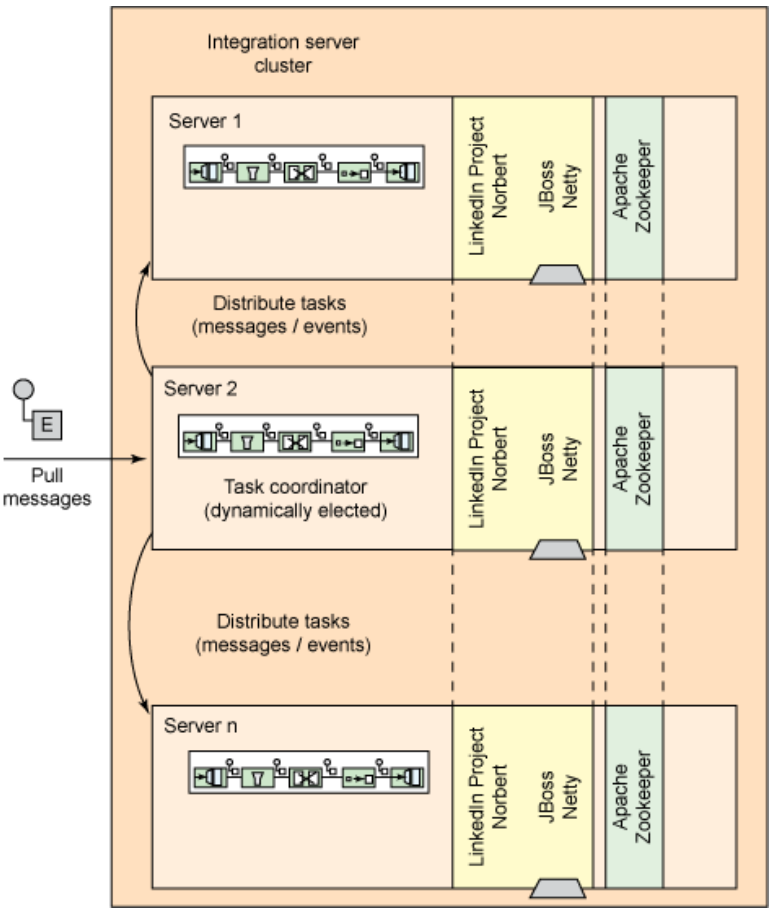


## サーバー・クラスター

この統合アプリケーションに必要なサーバー・グループ関連の機能は、Java SE (Standard Edition) にも Java EE (Enterprise Edition) にも用意されていません。これらの機能の例としては、サーバー・クラスタリング機能や、動的サーバー・リーダーの選出機能などがあります。

図 3 に、この EAI ソリューションを実装するために使用するオープンソースのツールを示します。具体的には、イベント処理には Spring Integration を使用し、クラスター対応にするために Apache ZooKeeper および LinkedIn の Norbert を使用します。

### 図 3. サーバー・クラスターの技術マッピング




## シミュレートするアプリケーションについて

ここでシミュレートするアプリケーションの目的は、Java ベースのサーバー・クラスターを開発する際の共通の課題を解決するために、Apache ZooKeeper と LinkedIn の Norbert を使用する方法を具体的に説明することです。このアプリケーションは以下のような動作をします。

- アプリケーションのイベント・ストアは、統合サーバー・クラスター内のすべてのサーバーがアクセスできる共有フォルダーでシミュレートします。
- 共有フォルダーに格納されるファイル (およびそこに含まれるデータ) は、統合イベントをシミュレートするために使用します。外部スクリプトを使用して継続的にファイルを作成し、それによってイベントの作成をシミュレートすることもできます。

- Spring Integration ベースのファイル・ポーリング用コンポーネントである、ファイル・ポーラー (インバウンド・イベント・アダプター) は、共有ファイル・システム・フォルダーによってシミュレートされたアプリケーションのイベント・ストアからイベントを取り出します。すると、ファイルのデータがサーバー・クラスターの残りのメンバーに配布されて処理されます。
- イベント処理をシミュレートするために、ファイルに含まれるデータの先頭には簡潔なヘッダー・タイプ情報 (`server id`、`timestamp` など) を追加します。
- パートナー・アプリケーションをシミュレートするために、さらに固有の共有ファイル・システム・フォルダーをいくつか使用します。これらの共有ファイル・システム・フォルダーのそれぞれが 1 つのパートナー・アプリケーションを表します。

サンプルとして使用するユース・ケース、私たちが提案するソリューション・アーキテクチャー、そしてシミュレートするアプリケーションの概要は以上のとおりです。次は、このサーバー・クラスターとタスク分散ソリューションで主役となる 2 つのコンポーネント、Apache ZooKeeper と LinkedIn の Norbert を紹介します。

## Apache ZooKeeper と LinkedIn の Norbert

元々 Yahoo Research によって開発された ZooKeeper は、Apache Software Foundation に採用された当初は Hadoop のサブプロジェクトでしたが、現在はトップ・レベルのプロジェクトとして分散グループ・コーディネーション・サービスを提供しています。この記事では ZooKeeper を使用して、シミュレートしたアプリケーションをホストするサーバー・クラスターを作成します。このアプリケーションに必要なサーバー・リーダー選出機能も、ZooKeeper で実装します (リーダー選出は、ZooKeeper が提供するその他すべてのグループ・コーディネーション機能に必須です)。

ZooKeeper サーバーは、サーバーのコーディネーションを可能にする手段として、インメモリーの複製された階層型ファイルシステムのような「znode」(ZooKeeper データ・ノード) データ・モデルを使用します。znode はファイルのようにデータを格納できるだけでなく、ディレクトリーのように子 znode を格納することもできます。

znode には 2 つのタイプがあります。1 つは、クライアント・プロセスによって明示的に作成、削除される「通常の znode」、もう 1 つは、作成元のクライアント・セッションが終了すると自動的に削除される「一時 znode」です。シーケンシャル・フラグを設定した、通常の znode または一時 znode が作成されると、10 桁の単調に増加する接尾辞が znode 名に付加されます。

ZooKeeper には、以下の特徴もあります。

- ZooKeeper により、各サーバーが起動時にグループ内の他のサーバーのリスナー・ポートを認識することが確実になります。「リスナー・ポート」は、リーダー選出、ピアツーピア通信、サーバーへのクライアント接続を容易にするサービスをサポートします。
- ZooKeeper はグループ合意アルゴリズムを使用してリーダーを選出します。リーダーが選出された後の残りのサーバーは「フォロワー」と呼ばれます。サーバー・クラスターが機能するには、最低限必要な数のサーバー (クォーラム) がクラスター内で稼働していなければなりません。
- クライアント・プロセスには、使用可能な操作のセットが定義されます。クライアント・プロセスはこれらの操作を使用して、znode に基づくデータ・モデルの読み取り、更新のオーケストレーションを行います。

- すべての書き込み操作はリーダー・サーバーを通じてルーティングされるため、書き込み操作のスケーラビリティーは制限されます。リーダー・サーバーは、ZAB (ZooKeeper Atomic Broadcast) と呼ばれるブロードキャスト・プトロコルを使用してフォロワー・サーバーを更新します。ZAB が更新順序を守ることによって、インメモリー・ファイルシステムに似たデータ・モデルは、グループまたはクラスター内のすべてのサーバーで同期されることになります。また、永続スナップショットにより、このデータ・モデルは定期的にディスクに書き込まれます。
- 書き込み操作に比べ、読み取り操作には遥かにスケーラビリティーがあります。フォロワーは、クライアント・プロセスによって行われる、このデータ・モデルの同期コピーからの読み取り操作に対処します。
- znode は、クライアントに対して 1 回限りのコールバック・メカニズムをサポートします。これは、「ウォッチャー」と呼ばれます。ウォッチャーは、監視対象の znode で変更が行われたことを知らせるシグナルを監視クライアント・プロセスでトリガーします。

### Norbert によるグループ管理

LinkedIn の Norbert は、Apache ZooKeeper ベースのクラスターに接続して、アプリケーションがサーバー・クラスター・メンバーシップを認識できるようにします。Norbert は実行時にこの処理を動的に行います。また、Norbert は JBoss Netty サーバーをラップして対応するクライアント・モジュールを提供し、アプリケーションがメッセージのやりとりをできるようにします。Norbert の前のバージョンでは、Google Protocol Buffers のオブジェクト・シリアライズ・ライブラリーを使用してメッセージをシリアライズしなければなりませんでしたが、現在のバージョンでは、カスタムのオブジェクト・シリアライズをサポートしています (詳細については「参考文献」を参照)。

#### Spring Integration

Spring Integration は、EAI の課題に対処するオープンソースのフレームワークです。宣言型コンポーネントによってエンタープライズ統合パターンをサポートする Spring Integration は、Spring プログラミング・モデルをベースとして作られています。

## サーバー・クラスターの構築

すべてのコンポーネントについて説明したところで、これから早速、イベント処理サーバー・クラスターの構成に取り掛かります。クラスターを構成する際の最初のステップは、サーバー・クォーラムを設定することです。サーバー・クォーラムが設定されると、新しく選出されたリーダー·サーバーが自動的にそのローカル・ファイルのポーリング・フローを開始します。ファイルのポーリングは、インバウンド・アプリケーション・イベント・アダプターをシミュレートする Spring Integration を介して行われます。アプリケーション・イベントをシミュレートするファイルがポーリングされると、それらのファイルはラウンドロビン・タスク分散ストラテジーに従って、使用可能なサーバーに配布されます。

ZooKeeper は、「サーバー・プロセスの大多数」を有効なクォーラムとして定義することに注意してください。したがって、3 つのサーバーで構成される最小のクラスターで 2 つ以上のサーバーがアクティブになると、クォーラムが確立されます。さらに、この例でシミュレートするアプリケーション内のサーバーには、それぞれに 2 つの構成ファイルが必要です。そのうち、一方のプロパティー・ファイルは、サーバー JVM 全体をブートストラップするドライバー・ロ

ジックによって使用されます。もう一方のプロパティー・ファイルは、(そのサーバーが属する) ZooKeeper ベースのサーバー・クラスターを構成するために必要です。

## ステップ 1: プロパティー・ファイルを作成する

この分散型 EAI アプリケーションを起動するコントローラー兼エントリー・クラスは、`Server.java` (「参考文献」を参照) です。このアプリケーションの初期パラメーターは、リスト 1 に記載するプロパティー・ファイルから読み取られます。

## リスト 1. サーバー・プロパティー・ファイル

```
# Each server in group gets a unique id:integer in range 1-255
server.id=1

# zookeeper server configuration parameter file -relative path to this bootstrap file
zookeeperConfigFile=server1.cfg

#directory where input events for processing are polled for - common for all servers
inputEventsDir=C:/test/in_events

#directory where output / processed events are written out - may or may not be shared by
#all servers
outputEventsDir=C:/test/out_events

#listener port for Netty server (used for intra server message exchange)
messageServerPort=2195
```

この最小構成のサーバー・クラスターに含まれるサーバーのそれぞれに、一意の `server id` (整数値) が必要であることに注意してください。

入力イベント・ディレクトリーは、すべてのサーバーが共有します。パートナー・アプリケーションをシミュレートする出力イベント・ディレクトリーも、オプションで全サーバーが共有するようにできます。ZooKeeper ディストリビューションには、サーバー・クラスターを構成するサーバー (「クォーラム・ピア」と呼ばれます) ごとの構成情報を構文解析するためのクラスが用意されています。サンプル・アプリケーションではこのクラスを再利用するので、ZooKeeper 構成を同じフォーマットにする必要があります。

もう 1 つ注意する点として、`messageServerPort` は、(Norbert ライブラリーによって起動、管理される) Netty サーバーのリスナー・ポートです。

## ステップ 2: 構築中の ZooKeeper サーバーの構成ファイルを作成する

## リスト 2. ZooKeeper の構成ファイル (server1.cfg)

```
tickTime=2000
dataDir=C:/test/node1/data
dataLogDir=C:/test/node1/log
clientPort=2181
initLimit=5
syncLimit=2
peerType=participant
maxSessionTimeout=60000
server.1=127.0.0.1:2888:3888
server.2=127.0.0.1:2889:3889
server.3=127.0.0.1:2890:3890
```

リスト 2 に示されているパラメーター (そして、変更しない限りデフォルト値が使用されるいくつかのオプション・パラメーター) については、ZooKeeper のドキュメントに説明が記載されています (「参考文献」を参照)。各 ZooKeeper サーバーが、3 つのリスナー・ポートを使用することに注意してください。clientPort (上記の構成ファイルでは 2181) は、クライアント・プロセスがサーバーに接続するために使用します。2 番目のリスナー・ポートは、ピアツーピア通信を可能にするためのポートです (サーバー 1 の場合の値は 2888)。そして、3 番目のリスナー・ポートで、リーダー選出プロトコルを使用できるようにします (サーバー 1 の場合の値は 3888)。各サーバーがクラスターのサーバー・トポロジー全体を認識することから、`server1.cfg` にはサーバー 2 とサーバー 3、およびそれぞれのピアツーピア・ポートも記載されています。

## ステップ 3: サーバー起動時に ZooKeeper クラスターを初期化する

コントローラー・クラス `Server.java` は別個のスレッド (`ZooKeeperServer.java`) を起動し、それによって ZooKeeperベースのクラスター・メンバーをラップします (リスト 3 を参照)。

## リスト 3. ZooKeeperServer.java

```
package ibm.developerworks.article;
…
public class ZooKeeperServer implements Runnable
{

   public ZooKeeperServer(File configFile) throws ConfigException, IOException
   {
      serverConfig = new QuorumPeerConfig();
      …
      serverConfig.parse(configFile.getCanonicalPath());
    }

      public void run()
   {
      NIOServerCnxn.Factory cnxnFactory;
      try
      {
         // supports client connections
         cnxnFactory = new NIOServerCnxn.Factory(serverConfig.getClientPortAddress(),
               serverConfig.getMaxClientCnxns());
         server = new QuorumPeer();

         // most properties defaulted from QuorumPeerConfig; can be overridden
         // by specifying in the zookeeper config file

         server.setClientPortAddress(serverConfig.getClientPortAddress());
         …
         server.start(); //start this cluster member

         // wait for server thread to die
         server.join();
      }
      …
   }


    …
   public boolean isLeader()
   {
      //used to control file poller.  Only the leader process does task
      // distribution
      if (server != null)
      {
         return (server.leader != null);
      }
```

```
    return false;
}
```

## ステップ 4: Norbert ベースのメッセージング・サーバーを初期化する

サーバー・クォーラムの設定が完了したら、Norbert ベースの Netty サーバーを起動することができます。このサーバーは、サーバー内部での高速なメッセージングをサポートしています。

## リスト 4. MessagingServer.java

```
public static void init(QuorumPeerConfig config) throws UnknownHostException
{
    …
   // [a] client (wrapper) for zookeeper server - points to local / in process
   // zookeeper server
   String host = "localhost" + ":" + config.getClientPortAddress().getPort();

   //[a1] the zookeeper session timeout (5000 ms) affects how fast cluster topology
   // changes are communicated back to the cluster state listener class

   zkClusterClient = new ZooKeeperClusterClient("eai_sample_service", host, 5000);

   zkClusterClient.awaitConnectionUninterruptibly();
   …
   // [b] nettyServerURL - is URL for local Netty server URL
   nettyServerURL = String.format("%s:%d", InetAddress.getLocalHost().getHostName(),
         Server.getNettyServerPort());
   …

   // [c]
   …
   zkClusterClient.addNode(nodeId, nettyServerURL);

   // [d] add cluster listener to monitor state
   zkClusterClient.addListener(new ClusterStateListener());

   //  Norbert - Netty server config
   NetworkServerConfig norbertServerConfig = new NetworkServerConfig();

   // [e] group coordination via zookeeper cluster client
   norbertServerConfig.setClusterClient(zkClusterClient);

   // [f] threads required for processing requests
   norbertServerConfig.setRequestThreadMaxPoolSize(20);

   networkServer = new NettyNetworkServer(norbertServerConfig);

   // [g] register message handler (identifies request and response types) and the
   // corresponding object serializer for the request and response
   networkServer.registerHandler(new AppMessageHandler(), new CommonSerializer());

   // bind the server to the unique server id
   // one to one association between zookeeper server and Norbert server
   networkServer.bind(Server.getServerId());

}
```

Norbert ベースのメッセージング・サーバーには、ZooKeeper クラスターのクライアントが組み込まれることに注意してください。このクライアントを、ローカルの (構築中の) ZooKeeper サーバーに接続するように構成した後、ZooKeeper サーバーのクライアントを作成します。セッション・タイムアウトは、クラスター・トポロジーでの変更が、どの程度迅速にアプリケーションに伝わるかに影響します。これにより実質的には、新しいサーバーが起動されたり、既存のサー

バーがクラッシュしたりしたときに、記録されているクラスター・トポロジーの状態が実際のクラスター・トポロジーの状態と同期しなくなる短い時間帯が作り出されることになります。したがって、アプリケーションはメッセージをバッファーに入れるか、あるいはこの時間帯におけるメッセージ送信失敗に対する再試行ロジックを実装しなければなりません。

`MessagingServer.java` (リスト 4) の処理内容は以下のとおりです。

- Netty サーバーのエンドポイント (URL) を構成します。
- 構成した Netty サーバーにローカルの `node Id` または `server Id` を関連付けます。
- クラスター状態リスナー (この後すぐに説明します) のインスタンスを関連付けます。Norbert はこのインスタンスを使用して、クラスター・トポロジーの変更をアプリケーションにプッシュします。
- ZooKeeper クラスター・クライアントを、設定中のサーバー構成インスタンスに割り当てます。
- リクエスト/レスポンス・ペアに、固有のメッセージ・ハンドラー・クラスを関連付けます。リクエスト・オブジェクトとレスポンス・オブジェクトをマーシャリングおよびアンマーシャリングするためのシリアライザー・クラスも必要です (GitHub に用意されているソリューション・コードには、マーシャリング用およびアンマーシャリング用のシリアライザー・クラスが付属しているので、それらのクラスを使用することができます。「参考文献」に記載されているリンクを参照してください)。

メッセージングでのアプリケーション・コールバックには、スレッド・プールが必要になることにも注意してください。

## ステップ 5: Norbert クラスター・クライアントを初期化する

次は、Norbert クラスター・クライアントを初期化します。リスト 5 に記載する `MessagingClient.java` がクラスター・クライアントを構成し、そのクライアントをロード・バランシング・ストラテジーに従って初期化します。

## リスト 5. MessagingClient.java

```
public class MessagingClient
{
   …
   public static void init()
   {
      …
      NetworkClientConfig config = new NetworkClientConfig();

      // [a] need instance of local norbert based zookeeper cluster client
      config.setClusterClient(MessagingServer.getZooKeeperClusterClient());

      // [b] cluster client with round robin selection of message target
      nettyClient = new NettyNetworkClient(config,
                                new RoundRobinLoadBalancerFactory());
      …
   }
   ...
    …
   // [c] – if server id <0 – used round robin strategy to choose target
   // else send to identified target server
   public static Future<String> sendMessage(AppRequestMsg messg, int serverId)
         throws Exception
   {
```

```
      …
      // [d] load balance message to cluster- strategy registered with client
      if (serverId <= 0)
      {
         …
         return nettyClient.sendRequest(messg, serializer);
      }
      else
      {
         // [e] message to be sent to particular server node
         …
         if (destNode != null)
         {
            …
            return nettyClient.sendRequestToNode(messg, destNode, serializer);
         }
         …
      }
   }
   …
}
```

リスト 5 で注目する点は、ターゲット・サーバーの `server Id` が正の値ではない場合、構成されているロード・バランシング・ストラテジーに従ってアクティブなグループからサーバーが選択され、そのサーバーにメッセージが送信されることです。アプリケーションでは、おそらく追加サーバー属性を基に、それぞれに固有のメッセージ処理ストラテジーとその実装を構成することができます (サーバー・サブクラスターを識別してリクエストを転送できるマルチテナント・アプリケーションを考えてみてください。この場合、テナントごとにメッセージ処理ストラテジーとその実装を構成する必要があります。詳細については、「参考文献」を参照してください)。

## 状態のモニターとタスク分散

シミュレートしたアプリケーションには、以下に記載する 3 つのコンポーネントがさらに追加されます。以降のセクションでは、これらのコンポーネントについて説明します。

- クラスターの状態 (サーバー·メンバーシップ) をモニターするコンポーネント。
- Spring Integration フロー定義ファイル。フロー定義ファイルは、シミュレートしたアプリケーションのタスク・プールから中央のタスク・ディストリビューターまでの EIP ベースのメッセージ・フローを定義します。タスク・ディストリビューターは最終的に、使用可能なクラスター・メンバーのいずれかに各タスクをルーティングし、各タスクはそこで処理されます。
- タスク・ディストリビューター。クラスター・メンバーのいずれかへの最終的なタスク・ルーティングを実装します。

### クラスター状態 (トポロジー) リスナー

「クラスター状態リスナー」は、使用可能なノードの最新のリストをメッセージング・クライアントが確実に持てるようにします。さらに、リーダー・サーバー上で唯一のイベント・アダプター・インスタンス (ファイル・ポーラー) を起動します。ファイル・ポーラーは、ポーリング対象のファイルのリストをメッセージ・プロセッサー・コンポーネント (POJO) に渡します。このコンポーネントが、実際にタスクを配布するタスク・ディストリビューターです。タスク・ディストリビューター・インスタンスはクラスター内に 1 つしかないため、ファイルのリストを渡した後は、アプリケーション・レベルで同期する必要はありません。クラスター状態リスナーをリスト 6 に示します。

### リスト 6. ClusterStateListener.java

```
public class ClusterStateListener implements ClusterListener
{
   …
   public void handleClusterNodesChanged(Set<Node> currNodeSet)
   {
      …
      // [a] start event adapter if this server is leader
      if (Server.isLeader() && !Server.isFilePollerRunning())
      {
         Server.startFilePoller();
      }

   }
   …
}
```

## Spring Integration ベースのファイル・ポーラー

Spring Integration のフローは以下のとおりです (リスト 7 にコードを示してあります)。

- `messageInputChannel` という名前のメッセージ・チャネル (パイプ) を作成します。
- JVM システム・プロパティー (「input.dir プロパティー」) から読み取ったディレクトリーに対して 10 秒ごとにファイルのポーリングを実行するインバウンド・チャネル・アダプターを定義します。ポーリング対象のファイルが検出されると、メッセージ・チャネル `messageInputChannel` に渡されます。
- メッセージ・チャネルからメッセージを受け取るタスク・ディストリビューター (Java Bean) を構成します。この Java Bean のメソッド `processMessage` が呼び出されて、タスク分散機能が実行されます。

### リスト 7. Spring Integrationベースのフロー (FilePoller_spring.xml)

```
…
   <integration:channel id="messageInputChannel" />

    <int-file:inbound-channel-adapter channel="messageInputChannel"
        directory="file:#{ systemProperties['input.dir'] }"
        filename-pattern="*.*" prevent-duplicates="true" >

        <integration:poller id="poller" fixed-rate="10" />
    </int-file:inbound-channel-adapter>

    <integration:service-activator input-channel="messageInputChannel"
        method="processMessage" ref="taskDistributor" />

    <bean
        id="taskDistributor"
        class="ibm.developerworks.article.TaskDistributor" >
    </bean>
…
```

## タスク・ディストリビューター

タスク・ディストリビューターには、リクエストをクラスター・メンバーにルーティングするロジックが含まれています。ファイル・ポーラー・コンポーネントは、リーダー・サーバー上でのみアクティブにされます。アクティブにされたファイル・ポーラー・コンポーネントは、ポーリングで検出したファイル (この例の場合、シミュレートされた統合イベント) をタスク・ディスト

リビューターに渡します。すると、タスク・ディストリビューターが Norbert クライアント・モジュールを使用して、リクエスト (ポーリングで検出したファイルをメッセージとしてラップしたもの) をクラスター内のアクティブなサーバーにルーティングします。タスク・ディストリビューターをリスト 8 に示します。

### リスト 8. Spring Integration のフローで制御されたタスク・ディストリビューター (Java Bean)

```
{
   …
   // [a] invoked by spring integration context
   public void processMessage(Message<File> polledMessg)
   {
      File polledFile = polledMessg.getPayload();
      …

      try
      {
         logr.info("Received file as input:" + polledFile.getCanonicalPath());

         // prefix file name and a delimiter for the rest of the payload
         payload = polledFile.getName() + "|~" + Files.toString(polledFile, charset);

         …
         // create new message
         AppRequestMsg newMessg = new AppRequestMsg(payload);

         // [b]load balance the request to operating servers without
         // targeting any one in particular
         Future<String> retAck = MessagingClient.sendMessage(newMessg, -1);

         // block for acknowledgement - could have processed acknowledgement
         // asynchronously by repositing to a separate queue
         String ack = retAck.get();
         …
         logr.info("sent message and received acknowledgement:" + ack);
      …
   }
}
```

ファイル・ポーラーが処理対象のファイルを検出すると、「サービス・アクティベーター」メソッドが制御側の Spring Integration コンテキストによって呼び出されることに注意してください。また、ファイルの中身がシリアライズされて、新しいリクエスト・オブジェクトのペイロードを形成することにも注意が必要です。メッセージング・クライアントの `sendMessage()` メソッドが呼び出されますが、このメソッドは特定の宛先サーバーをターゲットとしていません。Norbert クライアント・モジュールがロード・バランシングを行い、リクエストがラップされたメッセージの処理をクラスター内で稼働中のサーバーの 1 つに任せます。

## シミュレートしたアプリケーションの実行

この記事のソース・コード (「参考文献」を参照) には、3 つのサーバーからなるクラスターのサンプル構成ファイルと併せて、「実行可能」JAR ファイル (ZooKeeper-Norbert-simulated-app.jar) が含まれています。

このアプリケーションをテストするには、単一のマシン上で 3 つすべてのサーバーをローカル起動するか、3 つのサーバーをネットワークに分散させます。アプリケーションを複数のマシンで

実行する場合は、入力イベント・フォルダーを共通ネットワークにマウントしてアクセス可能な状態にする必要があります。クラスター内のサーバーの数を増やすには、追加するサーバーごとに 2 つの構成ファイルを作成して、既存の構成ファイルを更新します。

テキスト・コンテンツを含めたファイルを目的の入力フォルダーにコピーして、イベント処理をトリガーしてください。一連のファイルは、それぞれに異なるサーバーによって処理されます。サーバーの信頼性をテストするには、サーバーのいずれか 1 つを停止させます (3 つのサーバーからなるクラスターのクォーラムの場合、複数のサーバーが同時に停止した状態になると、アプリケーションが機能しなくなります)。アプリケーションに含まれている log4j.properties ファイルのロギングは、デフォルトで、TRACE レベルで有効に設定されています。ただし、サーバー・トポロジーは稼働中のサーバーで更新されることに注意してください。リーダー・サーバーを停止すると新しいリーダーが選出され、その新しいリーダー・サーバー上でファイル・ポーリング・フローがアクティブにされるため、確実に継続的な処理が行われます。

サーバー・クラスター対応のアプリケーション開発に Apache ZooKeeper および Norbert を使用する方法について詳しく学ぶには、「参考文献」セクションを参照してください。

---

# 著者について

Mukul Gupta

Mukul はこれまで 15 年間、C++、Forte、Scala、JavaScript、Java 言語などのさまざまな言語と技術を使用して、主に大規模なエンタープライズ・アプリケーションを設計、開発してきました。

---

Paresh Paladiya

Paresh は、システム開発ライフサイクル全体に 12 年間取り組んできたシニア・テクノロジストです。彼は、オブジェクト指向の設計とアジャイル開発手法、SOA およびシステム統合、インターネットおよびクライアント/サーバー・アーキテクチャーを専門としています。