# Build server-cluster-aware Java applications

Apache ZooKeeper and LinkedIn Project Norbert ease server-group coordination in distributed enterprise Java applications

Mukul Gupta and Paresh Paladiya
Published on November 13, 2012 / *Updated: August 27, 2013*



Many enterprise applications today are delivered by a set of cooperative, distributed processes and servers. Server-clustering capability, for example, is available for web requests for almost all of the popular Java enterprise servers, which may also provide limited configuration options like server weights and configuration reloads.

Although most Java enterprise servers have built-in support for clustering, that support isn't as readily available at the application level for custom use cases. As software developers, how should we manage use cases that involve distributed task coordination or that entail supporting multi-tenant applications? (A *multi-tenant application* is one that requires instances to be isolated on subsets of the overall server cluster or group.) For these types of use cases, we must find a way to make group coordination capabilities available at the application software layer, preferably at a high level of abstraction.

In this article, we present a guide to incorporating group membership and management capabilities into distributed Java applications. We'll start with a simulated Spring Integration-based Java application, which we will build out with a server-cluster abstraction layer based on two open source projects: Apache ZooKeeper and LinkedIn's Project Norbert.

## Server-clustering overview

Server-cluster-aware applications typically require at least some of the following capabilities:

- **Group membership with state maintenance and querying capability**: Real-time group membership is required in order to distribute processing over an active set of servers. In order to manage group membership, an application must be able to establish a process/server group and track the status of all of the servers in that group. It must also be able to notify the active servers when a server goes down or is brought up. The application will route and load-balance service requests among only the active servers in the cluster, thereby helping to ensure highly available services.
- **A primary or leader process**: One process in the cluster assumes responsibility for the group-coordination function of maintaining state synchronization across the server cluster. The mechanism of selecting the leader process is a special case of a broader set of problems known as *distributed consensus*. (Two and three-phase commits are well-known distributed consensus problems.)
- **Task coordination and dynamic leader server election**: At the application level, the *leader server* is responsible for task coordination, which it does by distributing tasks among other (follower) servers in the cluster. Having a leader server eliminates the potential for contention among servers, which would otherwise require some form of mutual exclusion or locking to enable eligible tasks to run. (This would be the case, for example, if servers were polling for tasks from a common data store.) Dynamic leader election is what makes distributed processing reliable; if a leader server crashes, a new leader can be elected to continue processing application tasks.
- **Group communication**: An application in a cluster-aware application should be able to facilitate the efficient exchange of structured data and commands across the server cluster.
- **Distributed locks and shared data**: Distributed applications should be able to access features like distributed locks and shared data structures like queues and maps if required.

**About ZooKeeper and Project Norbert**

Apache ZooKeeper is an open source project (see Related topics) that provides server-group coordination capability to distributed applications. Developed by LinkedIn, Project Norbert exposes ZooKeeper's server-group management and request-routing capabilities at a higher-level of abstraction, making them more accessible to Java application developers.

## Example use case: Spring Integration

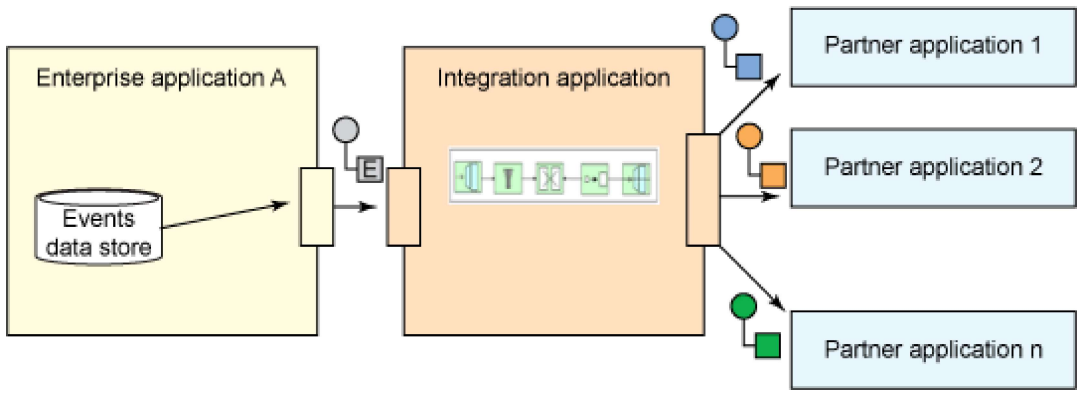
Our representative use case is an enterprise application integration (EAI) scenario that we will resolve with a simulated application based on Spring Integration. The application has the following characteristics and requirements:

1. A simulated source application produces integration-related events and messages as part of its regular transactional processing and stores them in a datastore.
2. Integration events and messages are processed by a distributed set of Java processes (a server cluster) that may run on the same server machine or be distributed across machines connected by a performant network. Server clustering is required for both scalability and high availability.

3. Each integration event is processed just once by any cluster member (that is, a given JVM). Output messages are communicated to partner applications over the intranet or Internet as applicable.

Figure 1 shows the integration event and message-processing flow outbound from the simulated source application.

Figure 1. Schematic of the Spring Integration-based example application




## Architecting a solution

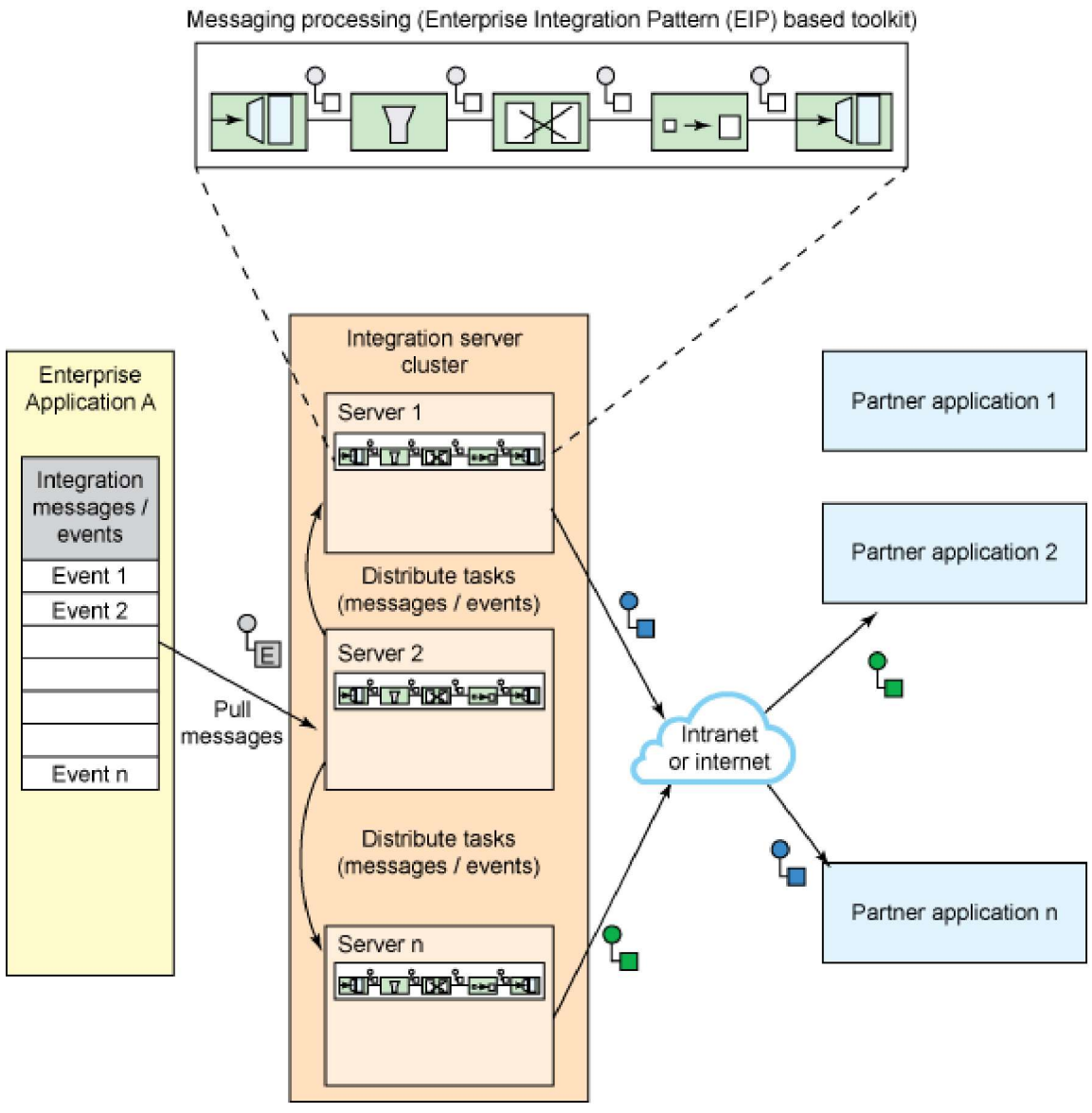
Our first step in developing a solution for this use case is to distribute the integration application to run over a server cluster. This should increase processing throughput and ensure both high availability and scalability. The failure of a single process will not stop the application from processing altogether.

Once distributed, the integration application will pull integration events from the application's datastore. A single server out of the server cluster will pull application events from the event store via a suitable application adapter and then distribute those events to the rest of the servers in the cluster for processing. This single server thus assumes the role of *leader server, or task coordinator*, responsible for distributing events (processing tasks) through the rest of the cluster.

Server cluster members that support the integration application become known at configuration time. Cluster membership information is dynamically distributed to all operating servers as new server members are started or when any server crashes or goes down. Likewise, the task-coordinator server is dynamically chosen; if the task-coordinator server crashes or becomes unavailable, an alternate leader server will be chosen cooperatively from among the remaining operating servers. Integration events may be processed by one of the many open source Java frameworks that support the Enterprise Integration Patterns (EIP) (see Related topics).

Figure 2 shows the use case solution schematic and components, which we further describe in the next section.

Figure 2. Use case solution schematic and server cluster components

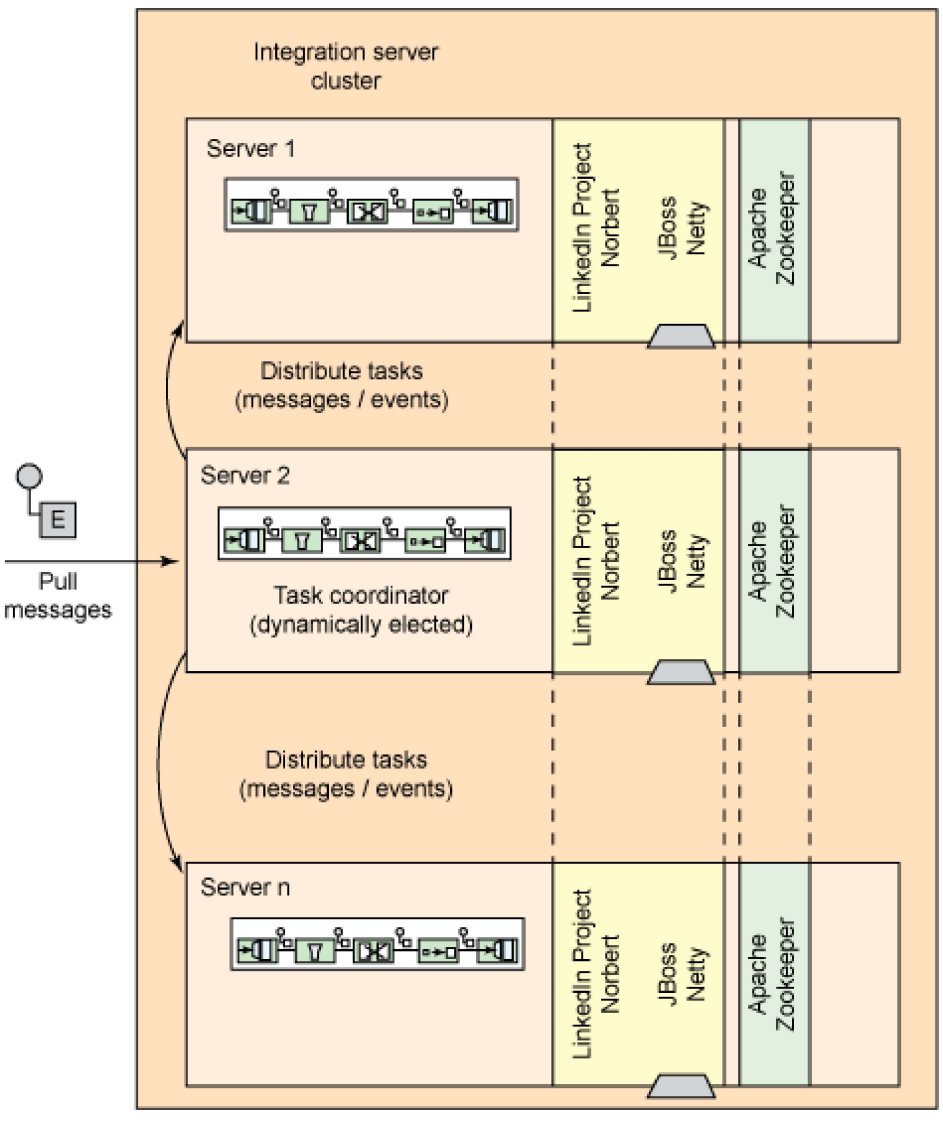



## The server cluster

Our integration application requires server-group-related features that are not available out-of-the-box on either Java Standard Edition (Java SE) or Java Enterprise Edition (Java EE). Examples of these include server clustering and dynamic server leader election.

Figure 3 shows the open source tools that we'll use to implement our EAI solution; namely Spring Integration for event processing and Apache ZooKeeper and LinkedIn Project Norbert to enable cluster awareness.

Figure 3. Technology mapping of the server cluster




## About the simulated application

The purpose of the simulated application is to demonstrate the use of Apache ZooKeeper and Project Norbert to resolve common challenges in developing Java-based server clusters. The application works as follows:

- The application event store is simulated by a shared folder that can be accessed by all servers within the integration server cluster.
- Files (and their data) that are reposited on this shared folder are used in order to simulate integration events. An external script may also be used to continually create files, thus simulating event creation.
- A Spring Integration-based file-poller component (the inbound event adapter) pulls events from the application event store, which is simulated by the shared file system folder. File data is then distributed to the rest of the server cluster members for processing.
- Event processing is simulated by prefixing the file data with brief header-type information such as `server id` and `timestamp`.
- Partner applications are simulated by a few other unique shared file system folders, one for each partner application.

You now have an overview of the example use case, our proposed solution architecture, and the simulated application. Now we're ready to introduce the two main components of our server cluster and task-distribution solution: Apache ZooKeeper and LinkedIn's Project Norbert.

# Apache ZooKeeper and Project Norbert

First developed by Yahoo Research, ZooKeeper was initially adopted by the Apache Software Foundation as a sub-project of Hadoop. Today it is a top-level project that provides distributed group coordination services. We'll use ZooKeeper to create the server cluster that hosts our simulated application. ZooKeeper will also implement the server-leader-election functionality that our application requires. (Leader election is essential to all of the other group-coordination functions that ZooKeeper offers.)

ZooKeeper servers enable server coordination via an in-memory, replicated, hierarchical file system-like data model of *znodes* (ZooKeeper data nodes). Like files, znodes can hold data, but like directories they can also hold child znodes.

There are two types of znode: *regular znodes* are explicitly created and deleted by client processes, whereas *ephemeral znodes* are automatically deleted when the originating client session goes away. When a regular or ephemeral znode is created with a sequential flag, a 10-digit monotonically increasing suffix is attached to the znode name.

More about ZooKeeper:

- ZooKeeper ensures that when servers are started, each of them is aware of the listener ports of the other servers in the group. *Listener ports* support the services that facilitate leader election, peer-to-peer communication, and client connections to servers.
- ZooKeeper uses a group-consensus algorithm to elect a leader, after which the other servers are known as followers. A minimum number of servers — a quorum — is needed for the server cluster to operate.
- Client processes have a defined set of operations available, which they use to orchestrate reads and updates of the data model based on znodes.
- All writes are routed through the leader server, which limits the scalability of write operations. The leader uses a broadcast protocol called ZooKeeper Atomic Broadcast (ZAB) to update follower servers. ZAB preserves the update order. The in-memory file system-like data model is thereby eventually synced on all the servers in a group or cluster. The data model is also periodically written to disk via persistent snapshots.
- Reads are vastly more scalable than writes. Followers respond to client process reads from this synced copy of the data model.
- znodes support a onetime callback mechanism for clients, called a "watcher." The watcher triggers a signal on the monitoring client process about changes to watched znodes.

## Group management with Project Norbert

LinkedIn's Project Norbert hooks into an Apache ZooKeeper-based cluster to provide applications with server-cluster-membership awareness. Norbert does this dynamically, at runtime. Norbert also wraps a JBoss Netty server and provides a corresponding client module to enable applications to exchange messages. Note that earlier versions of Norbert required using the Google Protocol Buffers object-serialization library for messages. The current version supports custom object serialization. (See Related topics to learn more.)

## Building a server cluster

**Spring Integration**

Spring Integration is an open source framework that addresses the challenges of EAI. It supports the Enterprise Integration Patterns via declarative components and is founded on the Spring programming model.

With all the components in place, we're ready to begin configuring our event-processing server cluster. The first step for the cluster is to establish server quorum, after which the newly elected leader server automatically starts its local file polling flow. File polling happens via Spring Integration, which simulates an inbound application event adapter. Polled files, simulating application events, are distributed to available servers using a round-robin task-distribution strategy.

Note that ZooKeeper defines a valid quorum as *the majority of server processes*. Thus a minimum cluster consists of three servers, with the quorum being established when at least two of the servers are active. Additionally, each server in our simulated application requires two configuration files: a properties file to be used by the driver logic that bootstraps the overall server JVM, and a separate properties file for the ZooKeeper-based server cluster (of which each server is a part).

### Step 1: Create a properties file

`Server.java` (see Related topics) is the controller and entry class that launches our distributed EAI application. The application's initial parameters are read from a properties file, as shown in Listing 1:

Listing 1. Server properties file

```
# 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
```

Note that a unique `server id` (integer value) is required for each of the servers in this minimum server cluster.

The input event directory is shared by all of the servers. The output event directory simulates a partner application and may optionally be shared by all of the servers. The ZooKeeper distribution provides a class for parsing configuration information for each of the server cluster's constituent servers or "quorum peers." Because our application reuses this class, it requires the ZooKeeper configuration in the same format.

Also note that the `messageServerPort` is the listener port for the Netty server (which is started and managed by the Norbert library).

### Step 2: Create a configuration file for the in-process ZooKeeper server

Listing 2. Configuration file for ZooKeeper (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
```

The parameters displayed in Listing 2 (as well as a few optional ones for which default values are used unless overridden) are described in the ZooKeeper documentation (see Related topics). Note that each ZooKeeper server uses three listener ports. The `clientPort` (2181 in the above configuration file) is used by client processes to connect to the server; the second listener port is used to enable peer-to-peer communications (value 2888 for server 1); and the third enables leader-election protocol (value 3888 for server 1). Each server knows the overall server topology of the cluster, so that `server1.cfg` also lists server 2 and server 3 and their peer-to-peer ports.

## Step 3: Initialize the ZooKeeper cluster on server start

The controller class `Server.java` launches a separate thread (`ZooKeeperServer.java`) that wraps the ZooKeeper-based cluster member, as shown in Listing 3:

Listing 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;
   }
```

## Step 4: Initialize the Norbert-based messaging server

After we've established server quorum, we can launch the Norbert-based Netty server, which supports fast intra-server messaging.

Listing 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
```

```
10
11      zkClusterClient = new ZooKeeperClusterClient("eai_sample_service", host, 5000);
12
13      zkClusterClient.awaitConnectionUninterruptibly();
14      …
15      // [b] nettyServerURL - is URL for local Netty server URL
16      nettyServerURL = String.format("%s:%d", InetAddress.getLocalHost().getHostName(),
17            Server.getNettyServerPort());
18      …
19
20      // [c]
21      …
22      zkClusterClient.addNode(nodeId, nettyServerURL);
23
24      // [d] add cluster listener to monitor state
25      zkClusterClient.addListener(new ClusterStateListener());
26
27      //  Norbert - Netty server config
28      NetworkServerConfig norbertServerConfig = new NetworkServerConfig();
29
30      // [e] group coordination via zookeeper cluster client
31      norbertServerConfig.setClusterClient(zkClusterClient);
32
33      // [f] threads required for processing requests
34      norbertServerConfig.setRequestThreadMaxPoolSize(20);
35
36      networkServer = new NettyNetworkServer(norbertServerConfig);
37
38      // [g] register message handler (identifies request and response types) and the
39      // corresponding object serializer for the request and response
40      networkServer.registerHandler(new AppMessageHandler(), new CommonSerializer());
41
42      // bind the server to the unique server id
43      // one to one association between zookeeper server and Norbert server
44      networkServer.bind(Server.getServerId());
45
46  }
```

Note that the Norbert-based messaging server includes a client to the ZooKeeper cluster. Configure this to connect to the local (in process) ZooKeeper server, then create a client for the ZooKeeper server. The session timeout affects how quickly changes in cluster topology are communicated back to the application. This effectively creates a small time window within which the recorded state of the cluster topology will be out of sync with the actual state of the cluster topology, as new servers are started or existing ones crash. Applications need to buffer messages and/or implement retry logic for message-send failures during this time period.

`MessagingServer.java` (Listing 4) does the following:

- Configures the endpoint (URL) for the Netty server.
- Associates the local `node Id` or `server Id` with the configured Netty server.
- Associates an instance of a cluster state listener, which we'll describe shortly. Norbert will use this to push cluster topology changes back to the application.
- Assigns the ZooKeeper cluster client to the server configuration instance being populated.
- Associates a unique message-handler class for a request/response pair. A serializer class is also needed for marshaling and unmarshaling request and response objects. (You may access the respective classes included with the solution code, available on GitHub; see Related topics for a link.)

Also note that application callback on messaging requires a thread pool.

## Step 5: Initialize the Norbert cluster client

Next, we initialize the Norbert cluster client. `MessagingClient.java`, shown in Listing 5, configures the cluster client and initializes it with a load-balancing strategy:

Listing 5. MessagingClient.java

```
1   public class MessagingClient
2   {
3      …
4      public static void init()
5      {
6         …
7         NetworkClientConfig config = new NetworkClientConfig();
8
9         // [a] need instance of local norbert based zookeeper cluster client
10        config.setClusterClient(MessagingServer.getZooKeeperClusterClient());
11
12        // [b] cluster client with round robin selection of message target
13        nettyClient = new NettyNetworkClient(config,
14                                    new RoundRobinLoadBalancerFactory());
15        …
16     }
17     ...
```

```
18      …
19     // [c] - if server id <0 - used round robin strategy to choose target
20     // else send to identified target server
21     public static Future<String> sendMessage(AppRequestMsg messg, int serverId)
22           throws Exception
23     {
24        …
25        // [d] load balance message to cluster- strategy registered with client
26        if (serverId <= 0)
27        {
28           …
29           return nettyClient.sendRequest(messg, serializer);
30        }
31        else
32        {
33           // [e] message to be sent to particular server node
34           …
35           if (destNode != null)
36           {
37              …
38              return nettyClient.sendRequestToNode(messg, destNode, serializer);
39           }
40           …
41        }
42     }
43     …
44  }
```

Note in Listing 5 that if the target server is not identified by a positive `server Id` value, the message is sent by selecting a server from among the active group based on the load-balancing strategy configured. Applications can configure their own message-handling strategies and implementations, perhaps based on additional server attributes. (Consider multi-tenant applications in which requests can be forwarded to identified server sub-clusters, one for each tenant; see Related topics for further discussion.)

# State monitoring and task distribution

There are three more components to the simulated application, which we describe in the following sections:

- A component to monitor the state (server membership) of the cluster.
- A Spring Integration flow-definition file. The flow-definition file defines the EIP-based flow of messages from the simulated application's task pool to the central task distributor. The task distributor will eventually route each task for processing to one of the available cluster members.
- A task distributor, which implements the final task routing to one of the cluster members.

## The cluster-state (topology) listener

The *cluster state listener* ensures that the messaging client has the updated list of available nodes. It also starts the sole event-adapter instance (the file poller) on the leader server. The file poller hands off the polled files to the message-processor component (a POJO), which is the actual task distributor. Because there is only one task-distributor instance within the cluster, no further application-level synchronization is required. Listing 6 shows the cluster state listener:

Listing 6. ClusterStateListener.java

```
1  public class ClusterStateListener implements ClusterListener
2  {
3     …
4     public void handleClusterNodesChanged(Set<Node> currNodeSet)
5     {
6        …
7        // [a] start event adapter if this server is leader
8        if (Server.isLeader() && !Server.isFilePollerRunning())
9        {
10          Server.startFilePoller();
11       }
12
13    }
14    …
15 }
```

## The Spring Integration-based file poller

The Spring Integration flow does the following (as demonstrated in Listing 7):

- Creates a message channel or pipe called `messageInputChannel`.
- Defines an inbound channel adapter that polls for files every 10 seconds from a directory that is read from the JVM system properties (that is, *property input.dir*). Any files that are polled for and located are passed down to the message channel, `messageInputChannel`.
- Configures the task distributor, a Java bean, to receive messages from the message channel. Its method, `processMessage`, is invoked to run the task-distribution function.

Listing 7. Spring Integration-based flow — 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>
…
```

## The task distributor

The task distributor contains the logic that routes requests across cluster members. The file-poller component is activated only on the leader server and passes polled files (in this case simulated integration events) to the task distributor. The task distributor uses the Norbert client module to route requests (which are polled files wrapped as messages) to the active servers in the cluster. The task distributor is shown in Listing 8:

Listing 8. Spring Integration flow-controlled task-distributor (a 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);
      …
   }
}
```

Note that the *service activator* method is called by the controlling Spring Integration context after the file poller locates a file for processing. Also note that the file's contents are serialized and form the payload of a new request object. The messaging client's `sendMessage()` method is invoked, but without targeting a particular destination server. The Norbert client module then load-balances the resulting message to one of the operating servers in the cluster.

## Run the simulated application

A "runnable" JAR file (*ZooKeeper-Norbert-simulated-app.jar*) along with sample configuration files for a three-server cluster are included with the source code for this article (see Related topics).

You may test the application by starting all three servers locally on a single machine or distributing them across your network. A common network mounted/accessible input events folder will be required in order to run the application on more than one machine. You can increase the number of servers in the cluster by creating corresponding configuration files, two per each additional server, and updating the existing configuration files.

Trigger event processing by copying files, with text content, into the designated input folder. Successive files are handled by different servers. Test the reliability of the service by stopping one of the servers. (Note, however, that quorum for a three-server cluster requires that only one server be down at any time for the application to function.) By default, the included *log4j.properties* file has logging enabled at the TRACE level; notice that the server topology will be updated with the servers that are running. If you bring down the leader server, then a new leader will be elected and the file polling flow will be activated on that server, thus ensuring continuous processing.

See the Related topics section to learn more about using Apache ZooKeeper and Project Norbert for server-cluster-aware application development.

## Downloadable resources

PDF of this content

## Related topics

- Learn more about Apache ZooKeeper: Visit the ZooKeeper homepage and documentation wiki.
- Next, check out these ZooKeeper presentation slides (Scott Leberknight, Near Infinity, April 2012) with accompanying source code.
- Learn more about LinkedIn's Project Norbert: Visit Norbert's homepage on GitHub and read about it on the LinkedIn Blog.
- Multi-tenant application development with Eclipse RT and Apache Gyrex is discussed in the tutorial presentation "Dynamic Server Applications with Eclipse RT" (Gunnar Wagenknecht, Andreas Mihm, Jochen Hiller; Eclipse Con 2012) (PDF format).
- "ZooKeeper fundamentals, deployment, and applications" (Mark Grover, developerWorks, August 2013): Explore the fundamentals of ZooKeeper, then learn how to set up and deploy a ZooKeeper cluster in a simulated miniature distributed environment. The author also examines the use of ZooKeeper in popular projects.
- Also see: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (Gregor Hohpe, Bobby Woolf; Addison-Wesley Professional, October 2003).
- "Distributed data processing with Hadoop, Part 1: Getting started" (M. Tim Jones, developerWorks, May 2010): ZooKeeper was initially adopted into Apache as a sub-project of Hadoop, a distributed data processing framework that is explored in depth in this four-part series.
- Download Apache ZooKeeper.
- Download Project Norbert.
- Download the source code for this article and use it to set up and run your own simulated Spring Integration server cluster.