The Corus Guide

www.sapia-oss.org

Table of Contents

REVISION HISTORY

Author	*Description*	*Date*	*Version*
Y.D	Added tagging, remote property update, execution configurations, process restart from the CLI,	2009/10/19	1.3.5

Introduction

Corus is an infrastructure that allows centrally controlling application processes in distributed environments. In a short list, here are Corus' features:

- Centralized, remote, replicated application deployment/undeployment across multiple hosts.
- Centralized, remote, replicated execution/termination of application processes across multiple hosts.
- Monitoring of running processes to detect crashes (and restart crashed processes automatically).
- Possibility for processes to publish status information at the application level, through the Corus infrastructure. Status information can then be monitored centrally.
- Replicated, JNDI-compliant naming service: based on Sapia's Ubik distributed computing framework (guaranteeing fail-over and load-balancing), the naming service can be used by deployed Java applications.
- Platform agnostic: potentially allows ANY type of applications to be deployed -Java, Python, C++ (currently only Java support offered).
- Allows centralized distribution of «standard» Java applications (Java classes with a «main» method), and remote control of corresponding JVMs.
- Allows multiple Java applications per-JVM, through the use of Magnet (http://www.sapia-oss.org/projects/magnet).
- Integration with the Java Service Wrapper (http://wrapper.tanukisoftware.org/doc/english/introduction.html) to ensure high-availabilityof Corus instances (and startup at OS boot time).
- HTTP/XML hook allowing to integrate Corus has part of a monitoring infrastructure.
- Free with an Apache business-friendly license.

In short, what Corus gives you is full, centralized control of distributed applications and application processes. For Java developpers, this means getting management features that you typically find in commercial application servers for free.

This guide delves into:

- Corus Architecture.
- Corus installation.

- Developing with Corus.
- Corus administration.

Building Blocks

This section gives a high-level view of the Corus architecture.

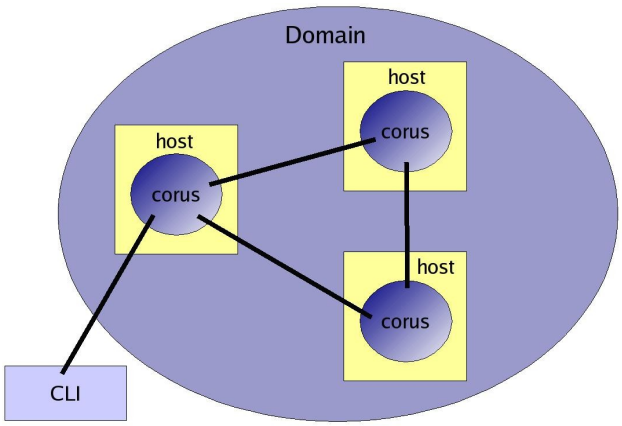
Infrastructure

The Corus infrastructure comes in three main pieces:

- A lightweight Java server that is installed on a host in order to allow remote deployment and remote execution of processes on that host.
- A Java command-line interface (CLI) that gives centralized control over multiple Corus servers.
- A Java monitoring tool that helps users/administrators see what is happening in a remote Corus server at runtime.

Topology

A typical Corus topology is illustrated below:



As can be seen, a single Corus server is typically installed on a given host. Corus servers are grouped by domains (or clusters). A domain of Corus servers can be centrally managed through the command-line interface. Each Corus server monitors/controls processes on the host on which it is installed.

Corus servers discover each other through multicast. The following describes how Corus servers are grouped in domains:

- At startup, Corus servers are given a domain name through the command-line, or through their configuration file.
- When a new Corus server appears in the network, it broadcasts an event stating the domain it is part of.
- Existing Corus servers will trap the event. If their domain name corresponds to the domain name that is part of the received event, they then in turn broadcast an event stating their existence.
- The newly started Corus server will in turn trap the events coming from existing servers. It will add these servers (or, rather, their corresponding addresses) in its internal sibling list.

A good rule of thumb is to group Corus servers by domains of 10 instances, to keep domains manageable.

The Corus Server

As was explained, a Corus server is installed on a given host to remotely execute and terminate processes on that host, as well as to deploy/undeploy distributions from which processes are eventually instantiated. At the core, that is all there is to it.

In fact, at a high-level, a Corus server does no more than a human being: typically, when wanting to use an application, we acquire it (very often, we download a zip or setup.exe file); then we install it; and then we start an instance of it (we «execute» it). When we're finished, we shut it down.

Of course, Corus (i.e.: a Corus server) will not download a zip file or an executable on its own. In this case, this part is replaced by an administrator deploying the required distribution into Corus, and then starting processes from it. The big advantage that Corus gives us is pretty obvious: typically, executing an application means invoking some executable on the machine on which the corresponding distribution has been installed. With Corus, local execution is handled by a server; the latter will execute processes upon request by «us», the users. Such requests are made with Corus' command-line interface (see next section).

The remote execution scheme becomes all the more powerful when multiple Corus servers collaborate in a domain: multiple processes on multiple machines can be controlled centrally, from the command-line interface.

In addition, Corus' functionalities do not stop at deployment/undeployment of application distributions, and at process execution/termination. Corus has full control over running processes. Through that control, Corus can offer the following features:

- Executed processes are monitored; those that are deemed crashed or in an unstable state are automatically terminated (and optionally restarted) by the Corus server that started them.
- When executing processes, Corus acquires their «original» process identifier (or PID) - the one that is assigned by the OS. This identifier is used to eventually agressively kill processes that do not obey the shutdown requests from their Corus server. It can also be used in problem-solving by system administrators.
- Client applications (running within Corus-executed processes) can publish status information to their corresponding Corus server. That status information can then be perused by administrators (using the command-line interface – see next section).

The Command-Line Interface

The command-line interface is used to:

- Deploy application distributions into Corus servers that are part of a given domain - when targeted at more than one Corus server, a deployment is replicated (the distribution is simultaneously deployed to all Corus servers).
- Start processes from the given distributions. Process execution can be replicated also: processes corresponding to given deployed distributions can be started on multiple hosts simultaneously.
- Undeploy applications from a single Corus server, or from multiple Corus servers in a domain.
- Stop running processes at a single Corus server, or at multiple Corus servers in a domain.
- Obtain status information from processes in a domain.
- View what distributions have been deployed.
- View what processes are running.

To ease its use, the CLI emulates Unix commands (`ls` lists deployed distributions, `ps` and `kill` respectively list and stop running processes, etc.).

The CLI in addition supports pattern matching operations. The following examples illustrate typical use cases:

Command	Description
`deploy dist/*.zip`	Deploys all distribution archives ending with the .zip extension, under the dist directory.
`exec -d myapp -v 1.* -n echoServer -p test`	Starts the echoServer process corresponding to all 1.xx versions of the myapp distribution under the test profile
`exec -d *  -v * -n *  -p test -cluster`	Starts the processes corresponding to all versions of all distributions under the test profile, on all Corus hosts in the cluster.
`kill -d myapp -v 1.* -n echoServer -p test`	Kill all echoServer processes corresponding to all 1.xx versions of the myapp distribution under the test profile.
`kill -d myapp -v 1.* -n *  -p test`	Kills all processes corresponding to all 1.xx versions of the myapp distribution under the test profile.
`kill -d * -v * -n *  -p test -cluster`	Kills all processes under the test profile, on all Corus hosts in the cluster.

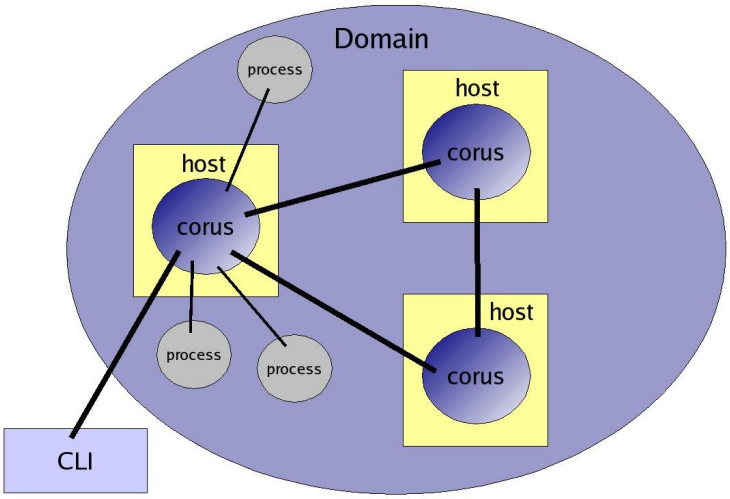
Help on the available commands can be obtained through the `man` command in the CLI. Typing `man` will display all available commands; typing `man <command_name>` will display help on the specified command.

Processes

Process execution occurs upon request of Corus administrators. Concretely: the `exec` command is typed at the CLI (with required command-line arguments), and a corresponding command object is sent to the Corus server to which the user is connected. If performed in clustered mode (if the `-cluster` switch has been entered at the command-line), the command is replicated to other Corus servers.

Upon receiving the command, the Corus server will start the specified process. From then on, processes executed by Corus servers are required to poll their respective Corus server, according to the protocol explained by the ***Corus Interoperability Specification***().

All communications between Corus-controlled processes and their server indeed follow that specification, which details a SOAP/HTTP-based protocol (where processes are in fact clients). The following diagram illustrates a Corus server and the processes it controls:



The protocol specifies a series of commands, corresponding to the life-cycle of processes started by a Corus server. That life-cycle can be summed up as follows:

- Processes are started following an `exec` command.
- After startup, processes poll their Corus server at a predefined interval to signal that they are up and running. At a predefined interval also, processes send status information to their server (application-code within the process can take part in status generation).
- When polling, processes eventually get back their list of pending events (the server keeps an object representation of each process, and within these objects, a queue of pending commands targeted at the «real» process).

- Processes are eventually killed using the `kill` command (not the one of the OS, but the one emulated by the CLI). The object corresponding to the kill command is sent to the desired Corus servers. Internally, the servers introspect the `kill` command to figure out what processes must be stopped; the objects representing the processes that must be killed have a «kill» event queued up in them. At the next poll, that signal is «recuperated» (sent as part of a SOAP response) to polling processes.
- Processes that are shutting down following a «kill» event must confirm their shutdown to their Corus server. As a matter of precaution, processes that fail doing so are eventually killed using an «OS» kill by their Corus server.
- Processes that do not poll their Corus server within the predefined interval are deemed crashed or unstable, and thus automatically killed (and optionally restarted, depending configuration information provided at deployment).

The fact that interoperability between Corus servers and their corresponding processes is implemented using SOAP over HTTP means that even if Corus is implemented in Java, potentially any type of application processes can be controlled with it (C++, Python, Perl, etc.).

Currently though, the only implementation of the ***Corus Interoperability Specification*** is in Java.

Naming and Remoting

A Corus server embeds a JNDI naming service that can be accessed remotely by Java applications that wish to bind/look up services. The naming service is based on Sapia's Ubik (http://www.sapia-oss.org/projects/ubik) distributed computing framework: Ubik is thus used by the Corus infrastructure as its remoting backbone (in fact, the Corus JNDI provider is based on Ubik's).

The JNDI module that is part of a Corus server offers the following features (directly inherited from Ubik):

- Multiple services can be bound under the same name (which enables the next two features)
- Load-balancing (according to two strategies: statefull/sticky, and stateless).
- Fail-over at the naming service (upon lookup) and at the stubs.
- Client-side discovery (client applications automatically discover JNDI server instances; stubs discover corresponding new servers appearing on the network).
- Replicated JNDI tree (service stubs bound to the naming service are replicated among all Corus servers in a domain).

The naming service is the glue that makes Java applications distributed with the Corus infrastructure fully scalable[1], without sacrificing manageability: multiple service instances can be centrally controlled and executed on multiple hosts (and even have multiple JVMs per host). These services bind themselves (or, rather, their corresponding stub) to the Corus JNDI, under the same name. Client applications lookup the services according to their given name; the JNDI implementation returns the appropriate stub to the client (in a round-robin fashion) - for more information pertaining to Corus' distributed computing framework, see the Ubik web site.

From the programmer's point of view, binding services to Corus' JNDI is similar to binding any object to a JNDI provider. The only difference is that the objects that are bound are automatically “remoted”: a stub for the object is created and sent to the JNDI server that resides within Corus. More concretely, here are the steps involved:

- Create an interface for the service (contrary to Sun's RMI, your interface does not have to extend `java.rmi.Remote`, and your methods need not throwing `RemoteExceptions`).
- Create an implementation of the interface (which will be bound to the Corus JNDI provider)
- Create the client of the service.
- There are examples of the above steps in Corus' source tree, under the `org.sapia.corus.naming.example` package. The snippet below, taken from these examples, illustrates how to bind a service:

```
try{
  TimeServer svr = new TimeServer();
  Properties props = new Properties();
  props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    RemoteInitialContextFactory.class.getName());
  props.setProperty(
    RemoteInitialContextFactory.UBIK_DOMAIN_NAME,
    "default");
  props.setProperty(Context.PROVIDER_URL,
    "ubik://localhost:33000");
  InitialContext ctx = new InitialContext(props);
  ctx.bind("timeService", svr);
  System.out.println("Time server bound...");
  while(true){
    Thread.sleep(100000);
  }
}catch(Throwable t){
```

1 Of course other distribution communication mechanisms, such as JMS, may be used.

```
    t.printStackTrace();
  }
```

And now the client part:

```
  try {
    Properties props = new Properties();
    props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
      RemoteInitialContextFactory.class.getName());
    props.setProperty(
      RemoteInitialContextFactory.UBIK_DOMAIN_NAME,
      "default");
    props.setProperty(Context.PROVIDER_URL,
      "ubik://localhost:33000");
    InitialContext ctx = new InitialContext(props);
    TimeService ts = (TimeService)ctx.lookup("timeService");
    while(true){
      System.out.println("Got time: " + ts.getTime());
      Thread.sleep(2000);
    }
  } catch (Throwable t) {
    t.printStackTrace();
  }
```

Note that the domain name is used has part of client-side discovery: if no Corus server is found on the given host and port, the JNDI implementation uses multicast to discover any Corus server that is part of the given domain and connect to it.

> Since Corus' JNDI is inherited from Ubik, we will spare ourselves from further details and encourage you to browse Ubik web site. Working with Ubik RMI is every similar to working with Jini, but a lot simpler.

Installation

Installing Corus is straightforward:

Download the Corus distribution from SourceForge (http://www.sf.net/projects/sapia) – the distribution comes as a .gzip or .zip file, depending on your target platform (sapia_corus-x.x.x-windows.zip or sapia_corus-1.x.x-linux.tar.gz).

- On the host where Corus will be installed, create a directory where the distribution will be "unzipped".

> Make sure that the full path to the directory you have chose does not contain spaces.

- On Unix/Linux, create a user (by convention, it is named `corus`) that will own the above-created directory.
- Create the `CORUS_HOME` environment variable on the Corus host; the variable should correspond to the full path of the directory that has been created in the previous step (under Linux/Unix, you can also define that variable in a file of the `/etc/profile.d` directory to automatically define the variable for every user).
- Add the `bin` directory under `CORUS_HOME` to your `PATH` environment variable.

That's it.

> Corus has been designed to preferably run on Linux/Unix machines.

Testing the Server

You are ready to start a Corus server. To test your installation, go to the command-line of your OS and type:

- On Linux/Unix: `sh corus.sh`
- On Windows: `corus`

Typing the above with no arguments should display the Corus server help. Read it for further information. Typically, to start a Corus server, you provide

the name of the domain “under” which the server should start, and the port on which it should listen (by default, the port is 33000). Hence:

```
sh corus.sh -d mercury -p 33100
```

Will start the server under the “mercury” domain, on port 33100.

Installing as a Service

Unix/Linux

- Copy the file `$CORUS_HOME/bin/corus.init.d` to the target `/etc/init.d/corus`.
- Edit the file `/etc/init.d/corus` and set the value of the `CORUS_HOME` variable that it contains. This definition is required because the corus script will be executed from an empty environment.
- Register the corus startup script with the `chkconfig` utility using the command `chkconfig -add corus`. The registration with the various run levels with make the Corus server to execute at startup.
- To manually start and stop the Corus server, use the command `service corus {start|status|stop}` as the root user.

Windows

- To install the Corus server as a windows service, executes the script `InstallCorusService.bat` provided in the distribution at the location `%CORUS_HOME%\bin\win32`.
- To uninstall the Corus windows service, execute the script `UninstallCorusService.bat` that is also located in the directory `%CORUS_HOME%\bin\win32`.

Testing the Client

To connect to the above server using the command-line interface, type:

- On Linux/Unix: `sh coruscli.sh -h localhost -p 33100`
- On Windows: `coruscli -h localhost -p 33100`

As you might have guessed, the arguments passed at the command-line

specify the host of the server we wish to connect to, as well as the port of that server (for help, type the command without any arguments).

Once you are in the CLI, you can see the list of available commands by typing:

```
man
```

Yes, you are right: just like Unix. You should see the list of available commands then. To read help about one of them, type:

```
man <name_of_command>
```

And off you go: help about the specified command is displayed.

To exist the CLI, type:

```
exit
```

> Reading the help of every command, as explained above, should give you a pretty clear idea of what you can concretely do with Corus. It is very much recommended doing that before going further.

Testing the Monitor

The Corus monitor is a nifty utility that displays logging output received from a Corus server, at runtime, remotely. It is great to introspect what the server is doing while you're deploying, executing processes, etc.

To start the Corus monitor, type (of course, a Corus server must be running):

- On Linux/Unix: `sh corusmon.sh -h localhost -p 33100`
- On Windows: `corusmon -h localhost -p 33100`

It can be a little while before you see something, but you will, eventually. To exit the monitor, just type `CTRL-C`.

Packaging Applications

Overview

The deployment unit in Corus is called a “distribution”. There are currently two ways to configure a distribution in Corus so that it can be deployed.

Distributions

With Corus, you deploy J2SE applications (meaning: Java classes with a `main()` method) packaged in .jar files. By convention, these .jar files are in fact named with a .car extension, but this is strictly a convention used to help distinguish Corus archives from other types of archives.

Corus does not force a programming model upon the programmer: learning and using a bloated API (such as EJB) is out of the question. With Corus, you deploy standard Java apps, period.

> Corus complements the lightweight-container approach very neatly: you can embed your lightweight-container in a `main()` method and deploy your application within Corus; you can use Corus' JNDI implementation to publish your “lightweight” services on the network, in a scalable and robust manner.

The .car file that you deploy within Corus (dubbed a “distribution” in Corus' jargon) is only mandated to provide a Corus descriptor under the `META-INF` directory (under the root of the archive). An example of that file is given below:

```
<distribution name="demo" version="1.0">
  <process name="echoServer"
           maxKillRetry="3"
           shutdownTimeout="30000" invoke="true">
    <port name="test" />
    <java mainClass="org.sapia.corus.examples.EchoServer"
           profile="test"
```

```
            vmType="-server">
        <xoption name="ms" value="16M" />
      </magnet>
    </process>
</distribution>
```

Before delving into the details of the configuration format, we will discuss it at large:

- In Corus, a distribution is the deployment unit. As was mentioned previously, Corus allows only managing Java applications for now, and thus a distribution consists of a set of Java process definitions.
- Process definitions are in fact blue prints for Java virtual machines: these Java virtual machines are started up by the Corus server, upon request by the user.
- The Java virtual machines started by Corus may optionally be configured with Magnet (see http://www.sapia-oss.org/projects/magnet). Magnet is simply a tool that is used to start multiple Java applications (Java classes with a main method) within the same VM; it allows configuring the classpath of each application (and much more) through a convenient XML format. Thus, what happens really is that Corus starts a Magnet process, which in turn invokes the `main()` method of each of its configured Java applications. Each application in turn has its own classloader, independent from the classloader of other applications (see Magnet's documentation for more information).
- Without the Java applications "knowing" it, Magnet starts a lightweight client that implements the *Corus Interoperability Specification*. That client polls the Corus server at a regular interval to let Corus know that the process in which it "lives " is still up and running. This is where Corus' process management functionality kicks in: a Corus server is aware of which processes it has started and will monitor them, making sure that they are polling at their predefined interval.

Deploying with Magnet: Step-by-Step

Rather then delving further into abstraction, we will go through implementing an application that is meant to be deployed into Corus (the configuration format will be explained in the next section). Implementing such applications is not much different then implementing them for "normal" use (except for the packaging part). The steps involved are as follows:

- Implement a J2SE application.
- Write the Magnet file for the application.
- Write the Corus descriptor for the application.
- Package the application in a .jar file, with the Corus descriptor

(named `corus.xml`) under the `META-INF` directory.

- Deploy the distribution (the .jar file) into Corus.
- Execute processes that are defined in the distribution.

To better illustrate the above steps, we provide the appropriate concrete examples:

1) Implement a J2SE application

The applications consists of a naive echo server implementation. As can be seen, it does not make use of any special class and does not follow any predefined development model in order to be deployed into Corus; it is a plain Java server started from a `main()` method:

```
package org.sapia.corus.examples;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class EchoServer extends Thread{

  private ServerSocket _server;
  private List _clients = Collections.synchronizedList(
    new ArrayList());

  /** Creates a new instance of EchoServer */
  public EchoServer(ServerSocket server) {
    _server = server;
  }

  public void close() throws IOException{
    _server.close();
  }

  public void run(){
```

```
  while(true){
    try{
      System.out.println("EchoServer::accept...");
      Socket client = _server.accept();
      System.out.println("EchoServer::client connection");
      ClientThread ct = new ClientThread(client, _clients);
      _clients.add(ct);
      ct.start();
    }catch(Exception e){
      e.printStackTrace();
      break;
    }
  }
  System.out.println("EchoServer::exiting...");
}

public static void main(String[] args){
  try{
    int port = Integer.parseInt(args[0]);
    EchoServer server = new EchoServer(
      new ServerSocket(port));
    server.start();
    Runtime.getRuntime().addShutdownHook(
      new Shutdown(server));
  }catch(Exception e){
    e.printStackTrace();
  }

}

////////////// INNER CLASSES ////////////

public static class ClientThread extends Thread{

  Socket _client;
  private List _queue;
```

```
public ClientThread(Socket client, List queue){
  _client = client;
  _queue = queue;
}

public void run(){

  BufferedReader reader;
  PrintWriter pw;
  try{
    reader = new BufferedReader(new
      InputStreamReader(_client.getInputStream()));
    pw = new PrintWriter(_client.getOutputStream(), true);
  }catch(IOException e){
    System.out.println("Client::terminated");
    _queue.remove(this);
    return;
  }
  while(true){
    try{
      String line;
      while((line = reader.readLine()) != null){
        System.out.println("Client::ECHO: " + line);
        if(System.getProperty(line) != null){
         pw.println("ECHO: " + System.getProperty(line));
        }
        else{
          pw.println("ECHO: " + line);
        }
        pw.flush();
      }
    }catch(IOException e){
      e.printStackTrace();
      try{
        _client.close();
      }catch(IOException e2){}
      break;
```

```
        }
      }
      System.out.println("Client::terminated");
      _queue.remove(this);
    }
  }

  public static class Shutdown extends Thread{

    private EchoServer _server;

    public Shutdown(EchoServer server){
      _server = server;
    }

    public void run(){
      try{
        _server.close();
      }catch(Exception e){}
    }
  }
}
```

2) Write the Magnet file

Magnet has been originally written to provide a multi-platform way of describing runtime parameters for Java applications that are typically started with os-specific startup scripts (such as `.sh` or `.bat` scripts, for example). In the case of Java applications, runtime parameters generally consist of system properties, the name of the Java application class, command-line arguments, and classpath information. Magnet's powerful XML format allows specifying all of that in a convenient and portable way (refer to Magnet's web site for more information). Here is the Magnet file for our echo server:

```
<magnet xmlns="http://schemas.sapia-oss.org/magnet/core/"
        name="EchoServer" description="A basic echo server">
  <launcher type="java" name="echoServer"
            mainClass="org.sapia.corus.examples.EchoServer"
            default="test" isDaemon="false" waitTime="2000"
            args="5656">
```

```
    <profile name="test">
      <classpath>
        <path directory="lib">
          <include pattern="*.jar" />
        </path>
      </classpath>
    </profile>
  </launcher>
</magnet>
```

In the above example, file resources (namely, the libraries that are part of the classpath) are resolved relatively to the "current directory", (available in Java under the `user.dir` system property). In terms of directory structure, the application is therefore organized in the following way:

`<user.dir>/echoServerMagnet.xml`

`<user.dir>/lib`

The `echoServerMagnet.xml` file contains the above Magnet configuration. Of course required libraries (if any) would appear under the `lib` directory. At development time, the `user.dir` variable corresponds to the "working" directory of your IDE.

One convenient way to work with Corus is to test the application prior to deploying it into Corus. Making sure that everything works properly prior to deployment allows sparing the dreadful deploy-test-redeploy cycle that is so common with EJB applications and consumes a lot of development time. In this case, we would invoke Magnet from the command-line (after compiling and generating a .jar for our echo server under the lib directory):

```
sh magnet.sh -magnetfile echoServer.xml -p test
```

The first command-line option consists of the name of our Magnet configuration file (if not specified, Magnet expects a `magnet.xml` file under the current directory). The second option consists of the name of the profile under which the Magnet VM should be started (for more information on the concept of profile, have a look at the Magnet web site).

If everything works fine, we are ready to write our Corus descriptor (there is a client for the server as part of Corus but we won't show it here; have a look in the package to see the source if you feel like it).

> The great strength of Corus in terms of application development is the absence of a rigid programming model that forbids starting applications from a simple command-line; it does not force deployment into a bloated container prior to be able to use and test applications.

3) Write the Corus Descriptor

The Corus descriptor is expected under the `META-INF` directory of the the archive that will be generated and deployed. In the case of this example, the

`META-INF` directory is created under the current directory (since the distribution will be generated from that directory). The content of the `corus.xml` file for the echo server is provided below:

```
<distribution name=”demo” version=”1.0”>
  <process name="echoServer"
           maxKillRetry="3"
           shutdownTimeout="30000"
           invoke="true">
    <port name="test" />
    <magnet magnetFile="echoServerMagnet.xml" profile="test">
      <vmType>-server</vmType>
      <xoption name="ms" value="16M" />
    </magnet>
  </process>
</distribution>
```

One thing to note is that the `corus.xml` file contains a `magnet` element. Since Corus' architecture was thought to eventually allow different types of processes to be started (for example, Python or Perl applications), the `magnet` element is a side- effect of that intent (eventually, a `python` element could be introduced): it is meant to indicate that a Magnet VM should be invoked; more concretely: a Magnet bridge instantiated within Corus is invoked and triggers the execution of the Magnet command-line, consistently with the parameters that are configured in the `corus.xml` file.

As can be noted, the `magnetFile` attribute indicates which Magnet configuration file should be used; the file's location should be given as a path that is relative to the root of the eventual Corus distribution (i.e.: the root of the .jar file). This is because as part of the deployment procedure, Magnet distributions are extracted under a predefined directory under Corus; the directory where a given distribution was extracted is the directory relatively to which Magnet configuration files of that distribution are resolved.

Another noticeable detail are the `name` and `version` attributes of the `distribution` element: every distribution deployed into Corus is uniquely identified by the value of its name and version. This also means that you can have multiple distributions with the same name, but that in this case the versions must differ.

> The Corus descriptor will be explained in more details further below.

4) Package the Application in a .jar File

Packaging a Corus distribution only requires putting in a .jar file the required resources: application libraries, Magnet file(s), Corus descriptor. In

the context of your example, we could use Ant to package everything that is under the root of the working directory. This would in effect mirror the structure on the file system, and that is one of the main goals: to eliminate disparities between usage in a local, standalone mode, and deployment on a multi-VM, distributed mode. The application that you implement and use locally on your workstation is the exact same one that will be deployed into Corus.

Thus, our application archive would have the following structure:

```
echoServerMagnet.xml
lib/
META-INF/corus.xml
```

5) Deploy the Distribution

Since an archive properly packaged as a Corus-deployable unit may contain more than one application (given as multiple process definitions or as multiple applications in the same Magnet configuration or both), that archive is named a "distribution", not an application anymore. That is such even in the case of our example, where only one application is involved.

Deploying an application into Corus involves calling the following command through the Corus command-line:

```
deploy <distribution_path>
```

Where `<distribution_path>` is the path to the Corus distribution that is to be deployed (if a relative, the path is resolved from the directory in which you have invoked the Corus command-line interface).

> See the *Installation* section further below for instructions on how to use the command-line interface.

6) Executing a process

Once a distribution has been successfully deployed, processes described as part of the distribution can be started. In the case of our example, we would invoke the `exec` command from the Corus client console:

```
exec -d demo -v 1.0 -n echoServer -p test
```

The above command will start a Java virtual machine "containing" an instance of our echo server (more precisely: a Magnet VM is invoked, with the configuration corresponding to the process that we want to execute).

In the above case, we are indicating to Corus that we want to start a VM corresponding to the demo distribution, version 1.0. We indicate the name of the process configuration (`-n`) since many such configurations could be part of a Corus descriptor. The last option tells Corus under which profile the process will be started (the notion of profile is fully explained in the *Advanced Issues* section, later on).

Note that we could also have passed the `-cluster` option to the command:

```
exec -d demo -v 1.0 -n echoServer -p test -cluster
```

This would actually start the process on all Corus servers in the domain that have the corresponding distributions.

Furthermore, we could have started more than one process, by specifying the `-i` option, indicating to Corus how many instances of the processes are to be started:

```
exec -d demo -v 1.0 -n echoServer -p test -i 3
```

or

```
exec -d demo -v 1.0 -n echoServer -p test -i 3 -cluster
```

In the latter case, we are requesting the startup of 3 echo servers at all Corus servers in the domain (note that this would result in port conflicts, since our echo servers on the same host all listen to the same port).

It is important to understand that processes are executed by a Corus server on the host machine of that Corus server; meaning: a process (a JVM in the case of our example) is started by a Corus server on its own host for each invocation of the `exec` command it receives.

Deploying using standard Java : Step-by-Step

If you find using Magnet unsuitable, you may use a simpler method (in this case, you will not benefit from Magnet's powerful multi-application and classpath management features).

In such a case, you must create a Corus archive that will have the following structure:

```
classes/

lib/

META-INF/corus.xml
```

Upon startup, the JVM corresponding to your application will be assigned the “classes” directory and all the archives under the “lib” directory as part of its classpath.

> The content of the “lib” directory (and subdirectories) must consist of archives with either the .jar or .zip extension – and supporting the zip format. The archives will be “inserted” in the classpath in alphabetical order. Corus will include archives in subdirectories, but will only sort based on the names of the files (excluding the path). Note also that the content of the “classes” directory will be inserted into the classpath first.

The content of the corus.xml file will in this case consist of the following:

```
<distribution name=”demo” version=”1.0”>
  <process name="echoServer"
           maxKillRetry="3"
           shutdownTimeout="30000"
           invoke="true">
```

```
    <port name="test" />
    <java mainClass="org.sapia.corus.examples.EchoServer"
          profile="test">
      <vmType>-server</vmType>
      <xoption name="ms" value="16M" />
    </java>
  </process>
</distribution>
```

Advanced Issues

Corus Server Working Directory

You can start multiple Corus servers on the same hosts if you please, as part of the same domain, or as part of different ones. In any case, each server instance will have its own working directory under the Corus home directory. The working directory of a given server instance is identified by the following pattern:

<domain>_<port>

For example, a Corus server instance listening on port 33000, as part of domain `dev`, would have `dev_33000` as its working directory.

Corus Distribution Directories

As was mentioned previously, when a distribution is deployed into Corus, it is extracted under the server's `deploy` directory. More precisely, Corus creates the following structure under the deployment directory:

```
distribution_name
            /version_number
                        /common
                        /processes
```

The actual distribution archive is extracted under the `common` directory. When executing a process of that distribution, Corus internally assigns an identifier to that process; it then creates a directory under the processes directory that is given the process identifier as a name. For example, if the process identifier is 1234567, then the `1234567` is created under the `processes` directory.

As is described in the *Corus System Properties* section further below, processes are passed the above information (their `common` and `process` directories) as the following system properties, respectively: `user.dir` and `corus.process.dir`.

The Corus Descriptor

Here is a full-fledged Corus descriptor (all elements are formally explained in the table further below) :

```
<distribution name="demo"
              version="1.0" tags=”someTag”>
  <process name="echoServer"
           maxKillRetry="3"
           shutdownTimeout="30000"
           deleteOnKill=”false”
           invoke="true"
           pollInterval=”15”
           statusInterval=”45”
          tags=”someOtherTag”>
    <dependency dist="testDist"
                version="1.0"
                process="testApp"
                profile="prod" />
    <port name="test" />
    <magnet magnetFile="echoServerMagnet.xml"
            profile="test"
            javaCmd="java"
           vmType="-server">
    <!-- Alternate standard Java configuration:
         simpler, but less powerful than Magnet -->
    <!-- java mainClass="org.sapia.corus.examples.EchoServer"
            profile="test"
            javaCmd="java"
           vmType="-server" -->
      <xoption  name="ms" value="16M" />
    </magnet>
  </process>
</distribution>
```

Name	***Description***	***Attributes***		
		Name	*Mandatory*	*Description*
distribution	The root element of all Corus descriptors	name	Yes	The name of the distribution.
		version	Yes	The version of the distribution.
		tags	Yes	A comma-delimited list of tags used to determine if the configured processes should be started (based on the tags of the current Corus server) – see the *Tagging* section.
process (1-*)	Holds process description information.	name	Yes	The name that identifies the process configuration.

Name	*Description*	*Attributes*		
		maxKillRetry	No	The number of times Corus should attempt to kill processes corresponding to this process configuration that are deemed stalled or "down" (defaults to 3).
		shutdownTimeout	No	The number of milliseconds that is given to processes to confirm their clean shutdown (defaults to 30000).
		deleteOnKill	No	Indicates if processes corresponding to this process configuration should have their process directory deleted after shutdown (defaults to false).
		invoke	No	Indicates if processes corresponding to this distribution will be started automatically when invoking `exec` for the distribution without indicating which process configuration to create processes from (if true, the value of this attribute indicates that processes corresponding to this process configuration must be invoked explicitly by passing the `-n` option to the `exec` command) – defaults to false.
		pollInterval	No	The interval (in seconds) at which processes corresponding to this configuration are expected to poll their Corus server (defaults to 10).
		statusInterval	No	The interval (in seconds) at which processes corresponding to this configuration are expected to provided their runtime status to the Corus server (defaults to 30).
		tags	No	A comma-delimited list of tags used to determine if the process should be started (based on the tags of the current Corus server) – see the *Tagging* section.
port (0-*)	Used to indicate a network port number that should be passed to the insances of the process.	name	Yes	The name of a port range, as configured in the Corus server (see the *Port Management* section futher on for more information).

Name	*Description*	*Attributes*		
magnet (0-1)	Encapsulates information required by the underlying Magnet launcher when starting up JVMs.	magnetFile	Yes	The path to the Magnet configuration file, relatively to the root of the Corus distribution archive.
		profile	Yes	The name of the profile under which processes are to be started.
		javaHome	No	Allows specifying usage of a different JVM by specifiying the home of the JDK or JRE installation directory that is desired (defaults to the same Java home as the Corus server by which processes are executed).
		javaCmd	No	Allows specifying the name of the Java executable that is to be invoked when starting JVMs (defaults to `java`).
		vmType	No	Indicates the type of VM (-client or -server for Java Hotspot) that is to be started.
java (0-1)	Encapsulates information required to launch a standalone Java application in its own JVM.	mainClass	Yes	The name of the application class (class with a “main” method) to invoke upon JVM startup.
		profile	Yes	The name of the profile under which processes are to be started.
		javaHome	No	Allows specifying usage of a different JVM by specifiying the home of the JDK or JRE installation directory that is desired (defaults to the same Java home as the Corus server by which processes are executed).
		javaCmd	No	Allows specifying the name of the Java executable that is to be invoked when starting JVMs (defaults to `java`).
		vmType	No	Indicates the type of VM (-client or -server for Java Hotspot) that is to be started.

Name	*Description*	*Attributes*		
dependency (0-*)	Used to indicate that a given process depends on another	distribution or dist	No	The name of the distribution to which the other process belongs. If not specified, the current distribution will be assumed.
		process	Yes	The name of the process on which the current process depends.
		version	No	The version of the distribution to which the other process belongs. If not specified, the version of the current distribution will be assumed.
		profile	No	If not specified, the profile of the parent **magnet** or **java** element is used.
option (0-*)	Corresponds to a VM option – such as `-cp` or `-jar` (see the help of the `java` command for more information).	name	Yes	The name of the option.
		value	Yes	The value of the option.
property (0-*)	Corresponds to a VM system property that will be passed to processes using the form: `-Dname=value`.	name	Yes	The name of the system property.
		value	Yes	The value of the system property.

Profiles

Corus supports the concept of “profile”. A profile is simply a character string that identifies the “type” of execution of a process. To be more precise, imagine that you have a distribution (in Corus' sense) that contains multiple process configurations corresponding to applications that are deployed in different environments, or used under different conditions. For example, you could deploy a distribution to a Corus server in your development environment, and deploy that same distribution in QA or pre-prod. By the same token, you could connect to some server simulator in a development environment, but to the real one in pre-prod or QA. A common problem is also the database, which could have different addresses across environments, and even be of a different brand (HSQLDB, Postgres) in these different environments.

The notion of profile is a simple way to work around the configuration problems that arise when working in different environments or using applications under different conditions. Based on the profile (passed around as a string, remember...) you could use different configuration files.

Therefore, Corus makes no assumption with regards to the profile; it just passes it to executed processes as a system property (the `corus.process.profile` system property). In addition, the Magnet starter (used by Corus to start Magnet processes) passes the profile to the VM through the `magnet.profile.name` system property (this is because Magnet also supports the notion of profile and uses that system property to identify the current profile – having a look at the Magnet web site will enlighten you).

So from your application, you only need "interpreting" that system property (i.e.: implement code that acts based on the value of that system property).

Process Properties

When executing a process, Corus passes to it process properties (in fact, VM system properties as supported in Java). Theses properties consist of all the ones specified as part of a) the process configuration in the Corus descriptor; b) the corus_process.properties file under CORUS_HOME/config; c) the properties stored in the server's configuration module, which can be administered through the command line – see the *Corus Properties* section further below. In addition, Corus will pass the following properties to new processes:

Name	***Description***	***Value***
`corus.process.id`	The unique identifier of the process that was started.	A Corus process identifier.
`corus.server.host`	The address of the Corus server that started the process.	An IP address.
`corus.server.port`	The port of the Corus server that started the process.	A port.
`corus.server.domain`	The name of the domain of the Corus server that started the process.	A domain name.
`corus.distribution.name`	The name of the distribution to which the process corresponds.	An absolute path to a directory.
`corus.process.dir`	The directory of the process.	The directory of the process.
`corus.process.poll.interval`	The interval at which the process is expected to poll its Corus server.	A time interval in seconds.
`corus.process.status.interval`	The interval at which the process is expected to send its status to its Corus server.	A time interval in seconds.

Name	*Description*	*Value*
`corus.process.profile`	The name of the profile under which the process was started.	The name of a profile.
`user.dir`	The “common” directory of all processes of the distribution of which the process is part.	An absolute path to a directory.

Process Dependencies

It may occur that some processes depend on other processes, and that these processes themselves depend on other ones, and so on. The Corus descriptor supports declaring such dependencies, through `dependency` elements.

When such dependencies are detected, Corus will process them so that processes are started in the appropriate order. This will startup multiple processes consecutively, in such a case the `corus.process.start-interval` property should be set properly in order to not bottleneck the CPU while applications complete their initialization phase.

The excerpt below (corresponding to a Corus descriptor) shows a dependencies can be declared on a per-process basis:

```
<distribution name=”demo” version=”1.0”>
  <process name="echoServer"
           maxKillRetry="3"
           shutdownTimeout="30000"
           invoke="true">
    <java mainClass="org.sapia.corus.examples.EchoServer"
          profile="test">
      <vmType>-server</vmType>
      <xoption name="ms" value="16M" />
        <dependency dist=”demo” version=”1.0”
                process=”configurator” />
    </java>
  </process>
</distribution>
```

Dependencies (processes on which other processes depend) are executed first, in the order in which they are declared: if process A depends on process B, and process B depends on process C, the execution order will be as such: C, B, A.

Execution Configurations

It may become tedious to start multiple processes manually. The process dependency feature may help work around this hassle, but what if some processes have no other processes that depend on them ? In such a case, if not started explicitly, they will never execute.

Corus allows defining so-called “execution configuration”. These configurations consist of XML files that are uploaded (through the `deploy` command) to a Corus server (or to multiple servers, in cluster mode).

An example process configuration is given below:

```
<exec name="test" startOnBoot="true">
  <process dist="web"
           version="1.0" name="httpServer" profile="prod" />
</exec>
```

The `exec` element takes the following attributes:

- **name (mandatory)**: an arbitrary name used to refer to the configuration later on.
- **startOnBoot (defaults to false)**: indicates if the configured processes are to be started when the Corus server itself boots up.

In addition, the `exec` element contains one to many `process` elements, each indicating which processes should be started. Each process element takes the following attributes, all mandatory:

- **dist**: the distribution to which the process belongs.
- **version**: the distribution's version.
- **name**: the name of the process.
- **profile**: the profile under which to start the process.

Execution configurations can be deployed, listed, undeployed, and “executed”, all through the command line. The following shows example commands – see the command line help of the appropriate command for more information.

1) To deploy

```
deploy -e myTestConfig.xml
```

2) To list the currently deployed execution configurations

```
ls -e
```

3) To undeploy

```
undeploy -e test
```

4) To execute

```
exec -e test
```

The processes corresponding to an execution configuration are treated the same way as if they had been started manually: process dependencies are resolved, and processed are started in the appropriate order. Thus, if all dependencies are declared appropriately, there is no need to specify processes on which other process depends as part of execution configurations: these will be started through the normal dependency resolution mechanisms.

Distributed Applications

As was explained in a previous section, Corus supports distributed application development by embedding a Ubik JNDI server. Therefore, your applications can use the Ubik distributed computing with Corus in a transparent manner; the only requirement is that you use Corus' JNDI provider on the application's side (and put the Corus library – `sapia_corus.jar` – in your classpath).

Of course, your are not mandated to use Ubik (and the Corus JNDI provider). You could start Java RMI servers, or as a matter of fact any type of server with Corus. Your could for example elect to use ActiveMQ (http://www.activemq.org/), and start ActiveMQ servers with Corus on multiple hosts, thereby providing an asynchronous, scalable and robust messaging fabric.

The real strength of Corus is that it allows deploying and controlling processes on multiple host, in a centralized manner. Corus has a grid-like feel to it, and allows you to do things that were possible only by paying expensive application server licenses.

Corus Properties

A Corus server takes into account two configuration files (actually, Java property files), both in its `config` directory. It also support remotely overriding the properties in these files: in such cases, the properties are kept in Corus' internal database. These files are:

- `corus.properties`: holds the server's configuration parameters
- `corus_process.properties`: holds properties that are passed to each started process.

Both of these files must respect the Java properties format. The `corus_process.properties` file holds arbitratry properties that are passed to executed processes (through the command-line) as Java system properties.

This means a command-line option respecting the format below is created for every property in the file:

`-D<property_name>=<property_value>`

For its part, as stated, the file named `corus.properties`, contains the server's configuration properties. These properties are described below (the description tells if the current property can be remotely updated – see further below for the remote update feature):

Property	***Description***	***Mandatory***	***Remote Update***
`corus.server.domain`	The name of this property corresponds to the name of the domain to which the Corus server belongs. If the domain is specified at the startup script of the Corus server (through the `-d` option), this property is not taken into account.	No (if not specified in the file, must be given as the `-d` option at the Corus server startup script).	No
`corus.server.port`	The port on which the Corus server should listen. If the port is specified at the startup script of the Corus server (through the `-p` option), this property is not taken into account.	No (defaults to 33000)	
`corus.server.tmp.dir`	The directory where distributions are deployed, pending their extraction under the `deploy` directory.	No (defaults to `<server_dir>/tmp`)	Yes
`corus.server.deploy.dir`	The directory where distributions are extracted and kept.	No (defaults to `<server_dir>/deploy`)	Yes
`corus.server.db.dir`	The directory where Corus' internal database stores its files.	No (defaults to `<server_dir>/db`)	Yes
`corus.process.timeout`	Delay(in seconds) after which processes that have not polled their Corus server are considered "timed out".	Yes	Yes
`corus.process.check-interval`	Interval(in seconds) at which the Corus server checks for timed out processes.	Yes	Yes
`corus.process.kill-interval`	Interval(in seconds) at which the Corus server attempts killing a crashed process.	Yes	Yes
`corus.process.start-interval`	Amount of time that Corus will wait for between process startups, when multiple processes are started at once by end-users.	No (defaults to 15)	Yes
`corus.process.restart-interval`	Amount of time (in seconds) a process must have been running for before it crashed and in order for an automatic restart to be authorized.	Yes	Yes

Property	*Description*	*Mandatory*	*Remote Update*
`corus.server.multicast.address` `corus.server.multicast.port`	Multicast address and port used for discovery, respectively. **Note**: the values of these properties are by default made to correspond with the default values used by Ubik (the distributed computing framework on which Corus is based).	Yes	Yes

`corus.server.security.hostPattern.allow`
`corus.server.security.hostPattern.deny`
	Each property is expected to hold a comma-delimited list of patterns of IP addresses from which clients are either allowed or denied connection, respectively.
	Here are valid patterns:
	`192.168.*.*`
	`192.*.*.*`
	`192.**`
	`192.**.*`
		No	Yes

`corus.server.syslog.protocol`	The protocol to use to connect to the Syslog daemon - value may be udp, tcp. **Note**: for Syslog integration to be activated, all properties under corus.servder.syslog.* must be set.	No	No
`corus.server.syslog.host`	The host of the Syslog daemon.	No	No
`corus.server.syslog.port`	The port of the Syslog daemon.	No	No

Remote Property Update

Corus supports remotely updating server or process properties. In such cases, it is not the actual property files that are modified: rather, Corus adds/removes the properties to/from its internal database; if specified, the properties will override the corresponding ones in the file.

The updates are performed using the Corus command line interface. The `conf` command allows adding, deleting, and listing either server or process properties. The command also supports updating so-called “tags” (explained in the next section). The syntax goes as follows:

1) To add a property, or a list of properties:

```
conf add -s svr|proc -p name1=value1[,name2=value2[,nameN=valueN[...]]
```

Where:

- **s** specifies the scope (indicating if the property is a server or process property)

- **p** consists of a comma-delimited list of name-value pairs corresponding to actual properties (there must be no space in the list for it to be parsed properly).

2) To list the properties

```
conf ls -s svr|proc
```

Where:

- **s** specifies the scope (indicating if the property is a server or process property)

3) To remove a property

```
conf del -s svr|proc -p some.property
conf del -s svr|proc -p some.*
```

Where:

- **s** specifies the scope (indicating if the property is a server or process property)
- **p** specifies the property to remove (may include wildcards corresponding to a property name pattern – in which case all matching properties will be removed

> For the properties with the "server" scope, the Corus instance must be restarted for the new values to be taken into account.

Tagging

In might be desirable to start given processes on specific hosts, and not on others. For example, imagine a process that scans a database table periodically, performs an action based on the data that is retrieved, and then deletes that data. Multiple such processes acting in parallel will pose a concurrency issue: the action will be performed redundantly. Thus, in such a case, one would want to make sure that this process is executed on a single host.

To support this, Corus has a so-called "tagging" feature: a tag is an arbitrary string that is used to determine if a process can be executed by a given corus instance. Concretely, it works has follows:

1) The Corus descriptor may be enriched with tags: both the distribution and process elements of the descriptor support a tags attribute: the value of the attribute takes a comma-delimited list of tokens, each corresponding to an actual tag.
2) A Corus server itself may be attributed with given tags: the conf command of the command-line interface may be used to add, list, and remove tags to/from a Corus server.
3) Prior to starting a process, the Corus server will determine if the tags

of that process match the ones it has configured: if the process and its distribution have no tags, the process is started; if all the tags assigned to a process and its distribution are contained in the set of tags of the server, the process is started; if all the tags assign to a process and its distribution are not contained in the set of tags of the server, the process is NOT started.

The example descriptor below shows how to configure tags: they can be specified as a comma-separated list on the **distribution** element:

```
<distribution name=”id-generator” version=”1.0”
tags=”singleton”>
  <process name="server"
           maxKillRetry="3"
           shutdownTimeout="30000"
           invoke="true">
    <java mainClass="org.myapp.IDGenerator"
          profile="test">
      <vmType>-server</vmType>
      <xoption name="ms" value="16M" />
    </java>
  </process>
</distribution>
```

The can also be specified on a per-process basis:

```
<distribution name=”id-generator” version=”1.0”>
  <process name="server"
           maxKillRetry="3"
           shutdownTimeout="30000"
           invoke="true" tags=”singleton”>
    <java mainClass="org.myapp.IDGenerator"
          profile="test">
      <vmType>-server</vmType>
      <xoption name="ms" value="16M" />
    </java>
  </process>
</distribution>
```

At runtime, the Corus server will determine the tags of a process by merging the tags at the process level with the ones at the distribution level: it is thus the union of these tags that is used for validating against the server's own

tags, according to the previously defined algorithm.

Remote Tag Update

The Corus command-line can be used to add, remove and list the tags of a corus server. The following provides examples (see the documentation of the `conf` command – by typing `man conf`) for more details.

1) Adding tags

```
conf add -t someTag,someOtherTag
```

2) Listing tags

```
conf ls -t
```

3) Removing tags

```
conf del -t someTag
conf del -t *
```

Port Management

Corus has a port management feature allowing the specification of so-called "port ranges". Network ports belonging to these ranges are leased to started processes and "recuperated" upon process termination – in order to be leased to other processes.

This feature was implemented as a workaround that can pose a configuration challenge when dealing with distributed applications that must run on a well-known port (a port known by the client, not a dynamic one that is abstracted by a naming service).

Since ports on a given host cannot be shared, applications using static ports must have them explicitly configured, which quickly becomes burdensome when dealing with multiple running instances of applications performing the same service.

Hence the Corus port management feature, which works as follows:

- The administrator creates port ranges (with a minimum and a maximum port) that are given a unique name.
- As part of the corus.xml file, if a process requires a port, then a corresponding `port` XML element should appear for that process' configuration. The element takes a `name` attribute whose value must correspond to the name of a configured port range.
- Upon process start-up, the Corus server remove an available port from the range that matches the port specified as part of the process configuration (Corus keeps a an internal list of leased and available ports for every configured port range).

- The acquire port is "passed" to the process as a system property (through the command-line that starts the process). The system property format is given below:

  ```
  -Dcorus.process.port.<range_name>=<port>
  ```

- That property can then be recuperated by the started process from application code.

The port manager built within Corus can be administered through the Corus command-line interface (type `man ports` at the command-line for more information).

The configuration snippet below shows how a port is configured; of course multiple `port` elements can be configured for each `process` element.

```
<distribution name="demo" version="1.0">
  <process name="echoServer"
           maxKillRetry="3"
           shutdownTimeout="30000"
           invoke="true">
    <port name="test" />
    <magnet magnetFile="echoServerMagnet.xml" profile="test">
      <vmType>-server</vmType>
      <xoption name="ms" value="16M" />
    </magnet>
  </process>
</distribution>
```

Syslog Integration

It is possible to redirect the Corus server's internal task manager output to a Syslog daemon (the Syslog4j library is used to that end – see http://syslog4j.org/).

To activate integration with Syslog, the following properties must be confured in the CORUS_HOME/config/corus.properties file (see the Corus Properties section):

- corus.server.syslog.protocol
- corus.server.syslog.host
- corus.server.syslog.port

Here is an example configuration:

```
corus.server.syslog.protocol=udp
corus.server.syslog.host=localhost
corus.server.syslog.port=5555
```

HTTP Extensions

Corus provides a mechanism dubbed “HTTP extensions” that allows various internal services to provide remote access over HTTP. The extensions are accessed through HTTP, at predefined URIs (each extension as an exclusive context path). The URL format to have access to a HTTP extension is as follows:

```
http://<corus_server_host>:<corus_server_port>/<extension_context_path>/...
```

Corus currenlty supports many convenient extensions, of which certain provide XML output, and are meant first and foremost for integration into monitoring infrastructures. The Corus home page is available at the following URL:

```
http://<corus_server_host>:<corus_server_port>/
```

This displays a HTML page describing the various extensions, and their respective URIs. These extensions explained in the next sub-sections.

File System

The file system extension is bound under the **files** context path. This extension simply serves the files under $CORUS_HOME over HTTP (meaning that the Corus home directory serves as the document root).

The extension is accessed as follows:

```
http://<corus_server_host>:<corus_server_port>/files/
```

Any additional data after the context path (excluding the query string) is interpreted as the path to a file or directory under the document root.

> Note that the trailing slash character (**/**) at the end of the URI is mandatory if for proper access to directories.

JMX

The JMX extension (available under the **jmx** context path) provides status information gathered from the plattorm MBeans of the Corus VM. The resulting XML output is provided as follows:

```
<vmStatus>
  <attribute name="runtime.startTime" value="1180649263641"/>
```

```
  <attribute name="runtime.startDate" value="Thu May 31 18:07:43 EDT 2007"/>
  <attribute name="runtime.upTime" value="7961"/>
</vmStatus>
```

Of course more data than illustrated above is provided. The format can easily be parsed by monitoring scripts. The goal of the JMX extension is indeed to allow remote monitoring by system administrators.

The JMX extension is accessed as follows:

```
http://<corus_server_host>:<corus_server_port>/jmx
```

Deployer

The Deployer is the internal module that manages the distributions deployed in Corus. It offers an extension that displays the currenlty deployed distributions (such as when using the `ls` command in the CLI). It can be accessed through the following:

```
http://<corus_server_host>:<corus_server_port>/deployer/ls
```

In addition, query string parameters can be used to filter the result (the parameters correspond to the options available for the `ls` command in the CLI), as the examples below illustrate:

```
http://<corus_server_host>:<corus_server_port>/deployer/ls?d=demo
http://<corus_server_host>:<corus_server_port>/deployer/ls?d=demo&v=1.0
http://<corus_server_host>:<corus_server_port>/deployer/ls?d=demo&v=1.*
```

As showed in the last URL, pattern matching is supported.

Processor

The Processor is the internal module that manages the processes that have been started through Corus. It provides an extension that allows viewing these processes and their respective status, mirroring the functionality of the `ps` and `status` command in the CLI. Two URIs are supported by the deployer extension, one displaying running processes, the other displaying their status. These URIs are the following, respectively:

```
http://<corus_server_host>:<corus_server_port>/processor/ps
http://<corus_server_host>:<corus_server_port>/processor/status
```

Both of these URIs support query string parameters corresponding to the options available through the `ps` and `status` commands in the CLI.

```
http://<corus_server_host>:<corus_server_port>/processor/ps?d=demo&v=1.0&p=prod
http://<corus_server_host>:<corus_server_port>/processor/ps?d=demo&v=1.*&p=prod
http://<corus_server_host>:<corus_server_port>/processor/status?i=1178989398
```

As can be observed, pattern matching is supported.

Java Processes

As was mentioned earlier, Corus allows starting any type of process (provided it interacts with Corus according to the Corus interoperability specification – http://www.sapia-oss.org/projects/corus/corusInterop.pdf). As far as Java processes are concerned, there are a few things to know...

Corus Interoperability Specification in Java

The Corus interop spec implementation in Java is another Sapia project. It is part of the Sapia CVS repository at SourceForge. The implementation consists of a small .jar file named `sapia_corus_iop.jar`. That library consists of a small XML/HTTP client (as required by the spec).

When executing Java VMs from Corus, that library is "placed" in Magnet's classpath. The library is made "invisible" to your applications since it is not placed at the level of the system classloader, but at the level of a classloader that is a child of the system classloader.

The Java interop implementation detects that it has been started by a Corus server by looking up the system properties; if it finds a value for the `corus.process.id` property, then it determines that it should poll the corresponding Corus server. In order to do so, it needs the host/port on which the Corus server is listening, which it acquires through the `corus.server.host` and `corus.server.port` system properties. Using the values specified through these properties, the Corus interop instance connects to the Corus server and starts the polling process.

Troubleshooting

In order to facilitate troubleshooting, the Java interop client creates the following files under the process directory:

- `stdout.txt`
- `stderr.txt`

These files hold the output resulting from the redirection of the `java.lang.System.out` and `java.lang.System.err` output streams.

In addition, when a Corus server starts a process, it logs that process' initial output to the given process' directory, in the following file:

- `process.out`

In the case of Java processes, this allows tracking errors that have occurred before the Magnet runtime could be loaded.

Conclusion

Corus is simply an infrastructure that allows starting processes (and for the moment, distributed Java virtual machines) on multiple hosts. It offers a low-cost, non-intrusive way to distribute applications on a large scale.