University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics
VU University Amsterdam
Faculty of Sciences

Julian Krzemiński
Student id. no. 209476 (UW), 1735705 (VU)

Integration of Microsoft Compute Cluster Server with the ProActive Scheduler

Master's Thesis
in COMPUTER SCIENCE
in the field of DISTRIBUTED SYSTEMS

Supervisors:

Fabrice Huet
OASIS Project, INRIA
University of Nice-Sophia Antipolis

and

Janina Mincer-Daszkiewicz
Institute of Informatics,
University of Warsaw

and

Thilo Kielmann and Kees van Reeuwijk
Dept. of Computer Science,
Vrije University Amsterdam

August 2008

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiazującymi przepisami.
Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.
Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Abstract

My role was to integrate the Microsoft Compute Cluster Server with the ProActive Scheduler. The ProActive Scheduler manages heterogeneous and largely distributed resources. The resources might be composed of a collection of clusters, P2P networks, etc. The integration enables to use Microsoft CCS as a part of the resources managed by the Scheduler in order to execute computational-intensive applications on the cluster. This is accomplished by long-term reservation of CCS compute nodes and short-term job scheduling inside these reservations.

Once the resources are under the supervision of the Scheduler users can specify and submit jobs for execution. A job is usually composed of a collection of tasks. Using the ProActive Scheduler we can specify tasks which are native applications, Java applications or ProActive applications. For all tasks I implemented a walltime mechanism which is a maximum allowed execution time. The walltime can be subsequently used in sophisticated scheduling polices (like backfilling) inside the scheduler. Another refinement to the job submission process is an implementation of a Forked Java Task, which is a Java task executed in a dedicated JVM (as opposed to simple Java tasks all executed in the same JVM). This gives more flexibility as to the execution environment of the task.

Keywords

ProActive Scheduler, deployment on clusters, Microsoft Compute Cluster Server, task walltime, forked java task

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Category and subject descriptor according to ACM Computing Classification System:
C.2.4 [`Computer-Communication Networks`]:
Distributed Systems - *Distributed applications*

Contents

List of Figures

Chapter 1

Introduction

Engineers and researchers have access to resources, which provide more and more computational capabilities. This creates an opportunity for solving new categories of problems, as well as enhancing solutions to already existing in the industry or research domain. To accomplish the above, we should be able to take full advantage of the resources, and this may pose a big challenge. In general, the resources can have a heterogeneous and largely distributed nature. A key issue here is to use a system which enables searching for resources, allocation, and in the end deployment of the application on these resources. An example of such system is a meta-scheduler. A meta-scheduler manages cluster resources, allocates single hosts and collections of hosts in P2P networks. It accepts users jobs for execution and optimizes execution by scheduling policies.

Interesting information about meta-schedulers can be found in [DGF95, HJC99].

1.1. INRIA

INRIA is the French National Institute for Research in Computer Science and Control - ranked 12th in the top 1000 R&D Centres in the world ([RAN]). I spent the spring semester 2007/08 at INRIA (OASIS team) in Sophia-Antipolis where I was writing this thesis. At that time I was also participating in the joint master programme between University of Warsaw and Vrije University in Amsterdam. The software I have implemented is now a part of the ProActive library which belongs to the OASIS team.

1.2. ProActive Scheduler

The ProActive Scheduler is an example of a meta-scheduler. It aims at providing heterogeneous resources to the users in a transparent way. It is composed of two components: one responsible for job management and one responsible for resource management. The job management component is in charge of scheduling, executing, monitoring, stopping and collecting results of computational jobs. The resource management component is primarily responsible for the acquisition and monitoring of nodes.

The ProActive Scheduler uses the ProActive library. Therefore, it uses the notion of ProActive nodes, active objects, wait-by-necessity mechanism, deployment framework and other features included in the library. The library description (Chapter 2) contains fundamental concepts essential to fully understand the contribution part.

1.3. Contribution of this Thesis

The contribution of this thesis involves work on both the resource and job management components. For the resource management part I integrated Microsoft Compute Cluster Server with the ProActive Scheduler (Chapter 4). It enables to use CCS as a part of a grid managed by the Scheduler. For the job management part I added task walltime mechanism and a new type of a task: Forked Java Task (Chapter 5). Task walltime is maximum allowed execution time of a task. Forked Java Task is a Java task executed in a dedicated JVM on the allocated node. In the end of my internship at INRIA all my implementations were incorporated into the trunk versions of the source code.

1.3.1. Resource management

The process of resource management includes two steps. First off, the resources are acquired. The acquisition process looks differently for every cluster or P2P network. The second step is monitoring of resources, and this process is the same for each resource.

My role involved adding support for a new cluster environment Microsoft Compute Cluster Server (CCS). Microsoft CCS is software for high-performance cluster computing. It manages a cluster of servers that includes a single head node and one or more compute nodes. The head node controls and mediates access to the cluster resources and is the single point of management, deployment, and job scheduling for the compute cluster.

The ProActive Scheduler has already been integrated with a few cluster environments. The Scheduler for the resource acquisition uses the deployment framework from the ProActive library. Therefore, a new module which performs CCS acquisition was integrated into the deployment framework. The acquisition process of CCS in the ProActive Scheduler resembles the acquisition process of foreign resources in Condor (described further on in *Related Work* section).

Chapter 2 briefly describes how to use the deployment framework. Chapter 3 presents a general architecture of Microsoft Compute Cluster Server and describes a cluster located at INRIA which I used in my thesis. In chapter 4 there is a thorough description of the integration process of CCS into the deployment framework. Once the integration into the deployment framework is complete it can be used by the ProActive Scheduler. Section 5.2 includes details how the ProActive Scheduler controls CCS resources.

The work on the CCS was sponsored and carried out in the collaboration with Microsoft. Upon successful completion of the integration the Microsoft renewed the collaboration with the OASIS team. The details of the cooperation and benefits of it are presented in section 4.5.

1.3.2. Job management

Once the resources are allocated, users can submit jobs for execution. A job is a collection of tasks. The ProActive Scheduler enables submission of different types of tasks like native applications, Java applications or ProActive applications.

My first goal was to add a task walltime mechanism, which is a maximum execution time of a task. In case a task does not finish before its walltime it is terminated by the Scheduler. The walltime can be used in sophisticated scheduling polices (like backfilling) inside the Scheduler.

The implementation of the Task Walltime is discussed in section 5.4.

Another area of my work on ProActive Scheduler involved adding more flexibility as to the execution environment of Java tasks. A Java task is simply a class name (with a fixed method) to execute. On each allocated machine there is a JVM running which will execute all Java tasks which are designated to this resource. However, some Java tasks might require execution on a JVM from a specific provider (IBM, HP, Oracle, ...), or might require additional options like increased memory size. Therefore, my second contribution was an implementation of Java tasks executed in dedicated virtual machines: Forked Java Tasks. Then, a new JVM is launched (on the allocated machine) and the execution takes place in that JVM.

The implementation of Java tasks executed in dedicated JVMs is discussed in section 5.3.

1.4. Related Work

1.4.1. Condor

Condor project ([TTL05]) started in 1988 at the University of Wisconsin-Madison and since then the system has been developed by tens of programmers. It is one of the most renowned high-throughput computing systems. One of its users is NASA Advanced Supercomputing facilities where Condor is used to distribute large computations across multiple computers. The process of allocation of machines resembles the process which I used in my implementation of CCS integration with the ProActive Scheduler (described in chapter 4).

Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

Condor can be used to manage a cluster of dedicated compute nodes. In addition, unique mechanisms enable Condor to effectively harness wasted CPU power from otherwise idle desktop workstations. For instance, Condor can be configured to only use desktop machines where the keyboard and mouse are idle. Should Condor detect that a machine is no longer available (such as a key press detected), in many circumstances Condor is able to transparently produce a checkpoint and migrate a job to a different machine which would otherwise be idle. Condor does not require a shared file system across machines - if no shared file system is available, Condor can transfer the job's data files on behalf of the user, or Condor may be able to transparently redirect all the job's I/O requests back to the submit machine. As a result, Condor can be used to seamlessly combine all of an organization's computational power into one resource.

Condor can take advantage of resources managed by foreign systems such as clusters. This is accomplished by a technique called *gliding-in*. The technique is illustrated in the figure 1.1. In Step 1 we submit a job composed of N Condor Servers to the job scheduler of the cluster (located on the head node). In Step 2 the servers begin executing and form a Condor pool. Finally, users may submit normal jobs which are then matched to and executed on remote resources.

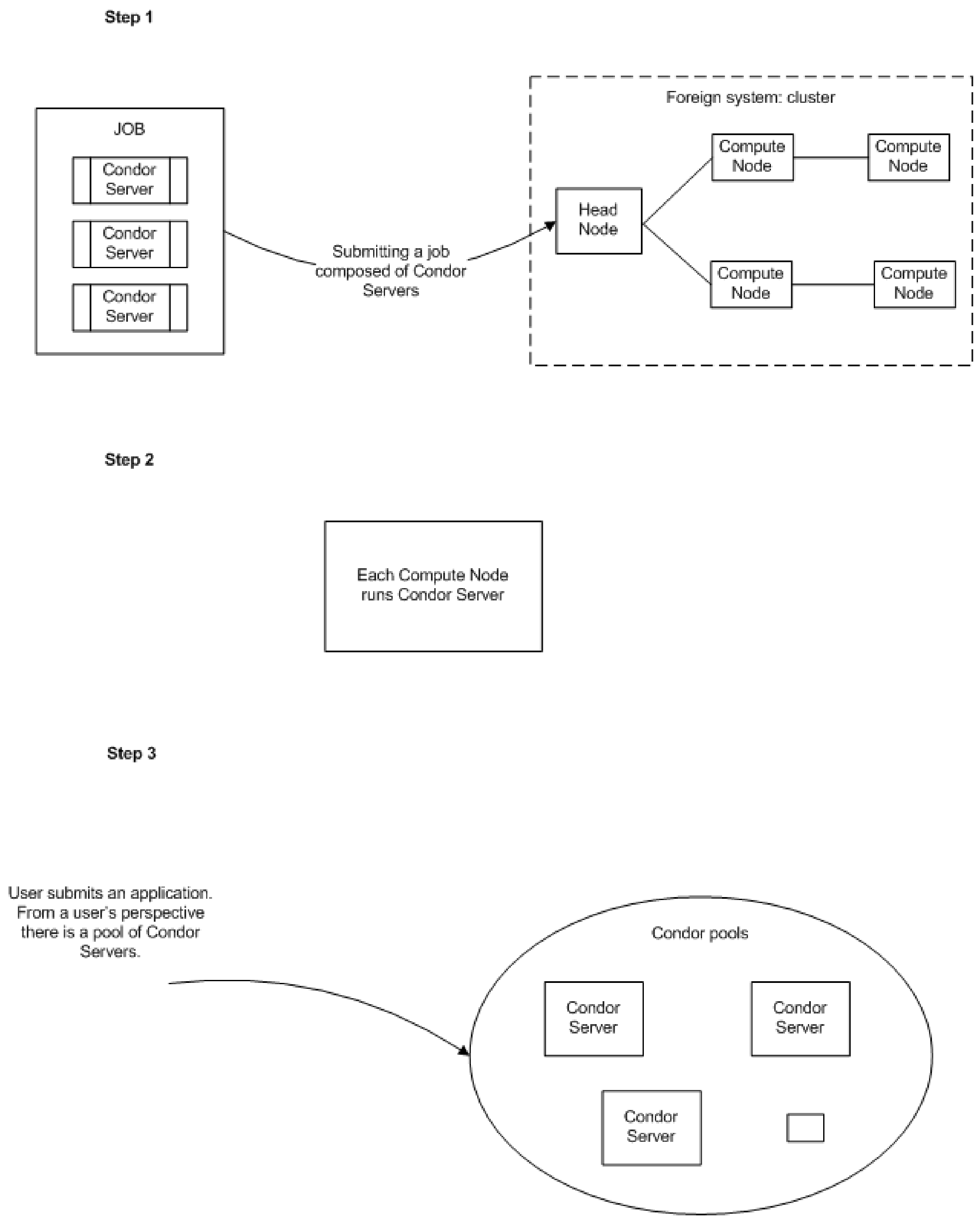


Figure 1.1: Gliding-in in Condor

Chapter 2

The ProActive library

2.1. Background

This chapter describes the library which I used in my master thesis. ProActive is an open source Java library aiming at the simplification of the programming of multithreaded, parallel, and distributed applications for Grids, multicores and clusters.

The basic concepts of the library are: *active objects, wait-by-necessity mechanism, future objects, ProActive Nodes* and *deployment framework*. This chapter contains a brief discussion of these concepts. Thorough introductions of the library can be found in [CQ03, CH05].

2.2. Programming model

2.2.1. Model of execution

The basic concept of the library is an active object. Compared to standard Java objects we gain:

- location transparency - standard Java objects are created in the current JVM, active objects can be created on any of the allocated machines. This is accomplished by using ProActive Nodes described in section 2.2.2.
- asynchronous communication - when calling a method on a standard Java object, caller waits until the result of a method is returned. In case of a method invocation on an active object, the caller can immediately return to its work and the result will be returned once it is computed.

By combining these two features we achieve distributed, asynchronous communication in Java.

An active object resembles the object composition technique in the Object-Oriented programming. The technique combines simple objects into more complex ones. It enables to intercept the invocations to the simple object and then provide some additional functionality. An active object contains a standard Java object and redefines all public methods provided by that standard object. A method call on an active object is not immediately invoked on the internal standard object. Instead, an object representation of a method call is put into a queue with pending method calls. Each active object has an additional internal thread which retrieves a method call (object) representation from the top of the queue, executes it on the standard Java object and returns a result to the caller.

An active object also uses inheritance. It extends a class of a standard Java object which is included inside this active object. Thus, using polymorphism, from a user's perspective standard Java objects and active objects look the same. However, the semantics of the method calls is different:

- When parameters of a method call on an active object are references to standard objects, they are always passed by deep-copy. Active objects, on the other hand, are always passed by reference. Symmetrically, this also applies to objects returned from methods called on active objects.
- When a method is called on an active object, it returns immediately. A future object, which is a placeholder for the result of the methods invocation, is returned. From the point of view of the caller, no difference can be made between the future object and the object that would have been returned if the same call had been issued onto a standard object. Then, the calling thread can continue executing its code just like if the call had been effectively performed. The role of the future object is to block this thread if it invokes a method on the future object and the result has not yet been set (i.e. the thread on which the call was received has not yet performed the call and placed the result into the future object). This type of inter-object synchronization policy is known as wait-by-necessity.

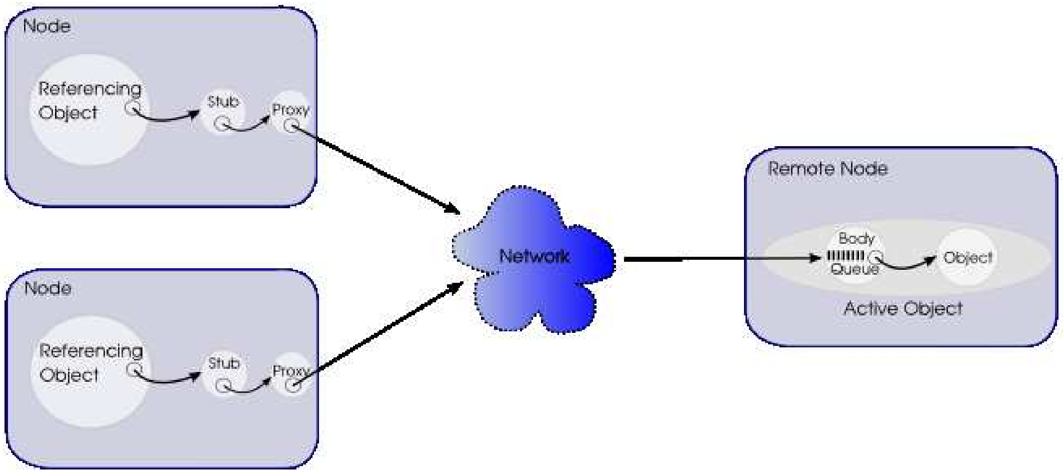


Figure 2.1: Model execution in ProActive

2.2.2. ProActive Nodes

Active objects are created on the ProActive Nodes. A ProActive Node is simply a Java application running on the allocated machine. To allocate a machine means to start at least one ProActive Node on that machine. Then, we can communicate with the Node itself and no longer use the machine address. Thus, the Node provides an abstraction of the location of the machine.

2.2.3. Programming Active Objects

Given a sequential Java program, it takes only minor modifications from the programmer to turn it into a multithreaded application which uses ProActive library.

Active Object Creation

ProActive actually only requires instantiation code to be modified in order to transform a standard object into an active one. All the code that had previously been written for the standard version of the same object remains perfectly valid. Besides the standard constructor parameters for the object, the creation of an active object requires at least a ProActive Node to create the object on. Depending on special semantics requirements, additional parameters may be passed. Here is a sample of code with several techniques for turning a standard instance of class A into an active, possibly remote, one.

A standard Java object created through such a statement:

```
A a = new A ("foo", 7);
```

becomes an active object by:

- using *ProActive.newActive* library function:

  ```
  Object[] params = {"foo", new Integer (7)};
  A a = (A) ProActive.newActive ("A", params, myNode);
  ```

- or using *ProActive.turnActive* library function:

  ```
  A a = new A ("foo", 7) ;
  a = (A) ProActive.turnActive (a, myNode);
  ```

This piece of code creates an instance of class A on node myNode. The mapping mechanism between nodes and actual virtual machines, processors and network hosts is described in section 2.3.

2.2.4. Communication

The communication between objects is carried out via method invocation.

- First off, we can distinguish unidirectional communication. Then, one object (caller) calls a method on an active (possibly remote) object (callee), which does not return anything.

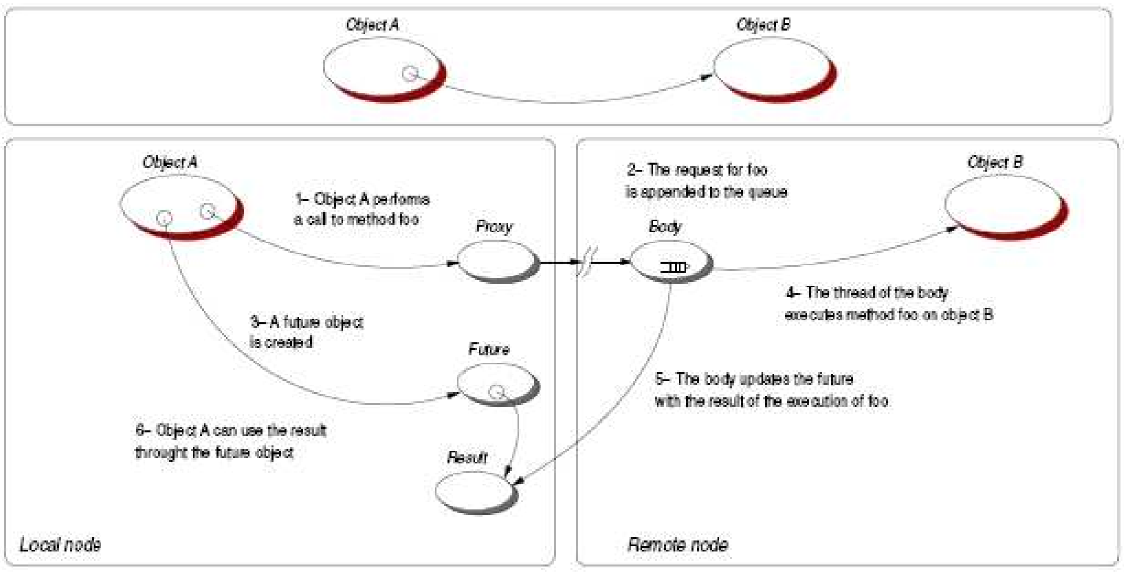
  ```
  public class A {
          public void foo (...) { ... }
  }
  ```

  The caller may pass data (passive or active objects) as method parameters, but receives nothing in return. Thus it is referred to as unidirectional communication.

- Secondly, we can introduce bidirectional communication by the invocation of a method with non-void return type:

```
public class A {
        public V bar (...) { ... }
}
```

Now the caller can pass parameters, and have a reply forwarded to him as a return object of the method.



Object A
Object B
Object A
1- Object A performs a call to method foo
Proxy
2- The request for foo is appended to the queue
Body
Object B
4- The thread of the body executes method foo on object B
3- A future object is created
Future
5- The body updates the future with the result of the execution of foo
6- Object A can use the result throught the future object
Result
Local node
Remote node

Figure 2.2: Bidirectional communication with futures

The user can retrieve a final value from a future object using the mechanism wait-by-necessity.

Consider the following code:

```
A a = (A) ProActive.newActive("A", params, node);
V v = a.compute();
```

As previously explained v is a future object, so it might be updated with a real value immediately, or at some point in the future.

Consider a new instruction after the above block:

```
v.getValue();
```

Here the wait-by-necessity mechanism will be used to ensure maximum efficiency of the asynchronous communication.

The programmer can also explicitly suspend further execution until the future is updated with a real value. To achieve this, the following method must be called:

```
ProActive.waitFor(v);
```

More information on asynchronous communication is presented in [CHS03, BBRZ05].

Synchronous communication

There are a few ways we can perform, or rather enforce, synchronous communication. Below, I present one of them:

Let the method return a simple Java type (int,boolean,...). The future object cannot be returned because primitive types are not instances of standard classes.

```
public class A {
       public int gee() {...}
}
```

2.3. Deployment Framework

Deployment framework is a part of the ProActive library. It enables applications to be deployed in diverse environments: local JVM, P2P network, or a cluster without any alterations to the application code. A good description of the framework is in [BCM+02, BBC+06].

A ProActive application comes with a set of XML deployment description files. When the deployment environment changes, the application remains untouched, only the deployment description files must be adjusted to a new environment.

2.3.1. Deployment restrictions

In order to deploy an application in different environments without changing the source code, we must follow a few principles.

First off, we need to remove from the application code any references to physical locations like machine names, creation protocols, registry lookup protocols. To launch JVMs used by an application we may use different protocols (rsh, ssh, Globus, prun). Also, when discovering services created by application, we might have to use protocols like RMIregistry, Jini, Globus MDS etc. Therefore the creation, registration and discovery of services must be carried out outside the application.

The second principle is to describe the application with the use of abstract entities. Suppose we have an application with a master process and a set of workers processes. The master process manages a task queue and the workers retrieve tasks from the queue, execute them, and return the results.

The code for the master and worker processes must be independent of physical locations of processes and must be independent of the number of workers. Hence, we have two abstract entities used by the application: a master and a worker. The master entity will be mapped onto one physical location, whereas the worker entity will be mapped onto many locations. An abstract entity in ProActive is called a *Virtual Node* (VN). It can be characterized in the following way:

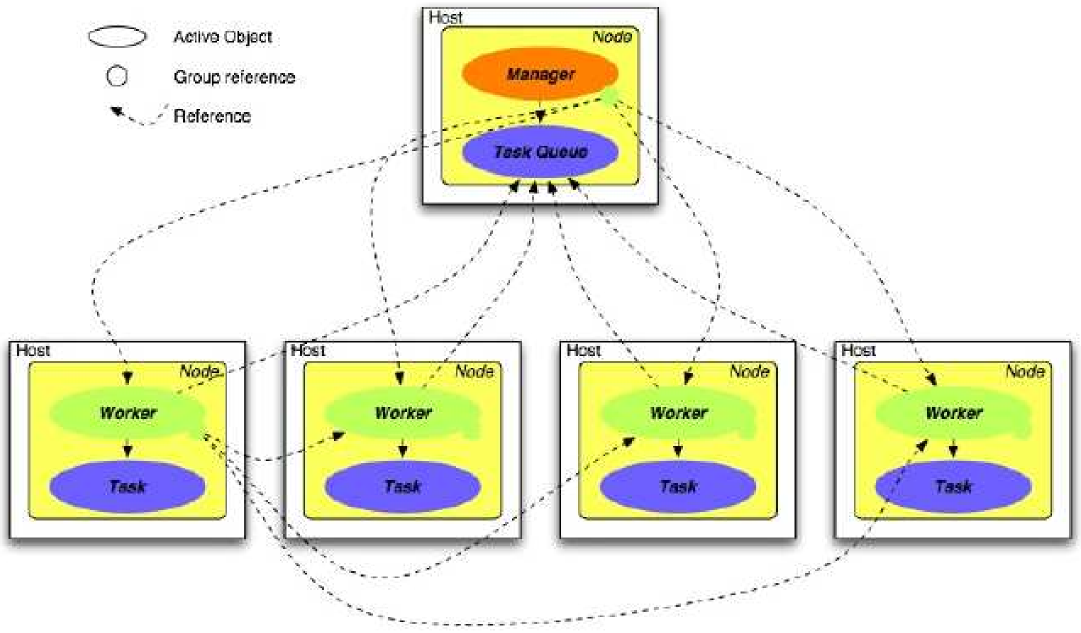


Figure 2.3: Master-Worker Paradigm

- The VN is identified by name (for example ”master” or ”worker”)
- The VN is used in the application code
- The VN in configured in the deployment framework and may be mapped onto one or more physical machines

Active Objects are created on ProActive Nodes, not on Virtual Nodes. Virtual Node is a concept of a distributed program (like a master VN) while a ProActive Node is actually a deployment concept (like an allocated node). There is a correspondence between Virtual Nodes and ProActive Nodes: the function created by the deployment, the mapping. This mapping is specified in an XML descriptor. By definition, the following operations can be configured in a deployment descriptor:

- the mapping of VNs to Nodes and Nodes to JVMs
- the way to create or acquire JVMs
- the way to register or lookup VNs

2.3.2. Master-Worker example

Now, within the source code, the programmer can manage the creation of Active Objects without relying on machine names and protocols. In the below example, the master virtual node and workers virtual node are retrieved, and then a master object, and a vector of worker objects are created basing on the mapping defined in descriptorFile :

```
ProActiveDescriptor pad = ProActive.getProactiveDescriptor( descriptorFile );
VirtualNode masterVN = pad.getVirtualNode("Master");
VirtualNode workersVN =pad.getVirtualNode("Workers");
```

```
Vector<Worker> workers = new Vector<Worker>();
for (Node node : workersVN.getNodes()) {
        Worker w = (Worker) ProActive.newActive(
                            Worker.class.getName(), new Object[] {},node);
        workers.add(w);
}
Master master = (Master) ProActive.newActive(
                         Master.class.getName(),new Object[] {workers},
                         masterVN.getNode());
master.performComputations(workers);
```

Having implemented the above application, we can create a deployment file adjusted to the environment that we have. Let's assume that the master process will run on a current JVM, and there will be two worker objects defined on Node1, Node2, which are subsequently mapped onto a local machine, and Amstel machine respectively.

First off, we define two abstract components Master Virtual Node and Workers Virtual Node:

```
<componentDefinition>
      <virtualNodesDefinition>
            <!-- we declare below a (single) master component -->
            <virtualNode name="MasterVN" property="unique_singleAO"/>
            <!-- a declaration of a (multiple) workers component -->
            <virtualNode name="WorkersVN" property="multiple" />
      </virtualNodesDefinition>
</componentDefinition>
```

Then, two mappings are specified. The MasterVN is mapped directly onto the current JVM, and the WorkersVN is mapped onto Node1 and Node2.

```
<mapping>
      <map virtualNode="MasterVN">
            <jvmSet>
                  <currentJVM protocol="rmi"/>
            </jvmSet>
      </map>
      <map virtualNode="WorkersVN">
            <jvmSet>
                  <vmName value="Node1"/>
                  <vmName value="Node2"/>
            </jvmSet>
      </map>
</mapping>
```

Next thing is to describe via which processes (elements processReference) the Node1 and Node2 will be acquired:

```
<jvms>
      <jvm name="Node1">
            <!-- The first JVM is created locally on the same machine-->
            <creation>
```

```
                <processReference refid="localJVM"/>
            </creation>
        </jvm>
        <jvm name="Node2">
            <!-- The second JVM is created on an amstel machine-->
            <creation>
                <processReference refid="amstelJVM"/>
            </creation>
        </jvm>
</jvms>
```

Finally, there are descriptions of processReference elements which describe the actual acquisitions of single hosts. The localJVM processReference simply points to the Java class to execute, while amstelJVM uses an RSH process to log in and invoke a localJVM process on that machine.

```
<processes>
        <processDefinition id="localJVM">
            <jvmProcess
                class="org.objectweb.proactive.core.process.JVMNodeProcess" />
        </processDefinition>

        <processDefinition id="amstelJVM">
            <rshProcess
                class="org.objectweb.proactive.core.process.rsh.RSHProcess"
                hostname="amstel.inria.fr">
                <processReference refid="localJVM" />
            </rshProcess>
        </processDefinition>
</processes>
```

Being provided with this information the ProActive library will take care of acquisition of the requested resources.

Chapter 3

Microsoft Compute Cluster Server architecture

Microsoft Compute Cluster Server is a cluster of servers that includes a single head node and one or more compute nodes. The head node controls and mediates all access to the cluster resources and is the single point of management, deployment, and job scheduling for the compute cluster.

Any computer configured to provide computational resources as part of the compute cluster is called a compute node. Compute nodes allow users to run computational jobs. These nodes must run a supported operating system, but they do not require the same operating system or even the same hardware configuration. Optimally, compute nodes include a similar software configuration to simplify deployment, administration, and resource management.

3.1. General Architecture

This chapter presents a general architecture of the CCS. I describe respectively: the hardware and software requirements for the nodes in the cluster, possible network topologies, types of jobs supported by CCS, architecture of the Job Scheduler, and run-time job and task management.

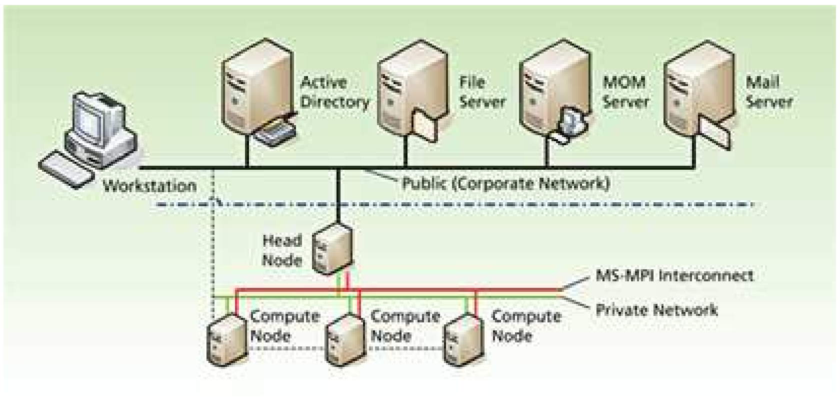


Figure 3.1: Typical Windows Compute Cluster Server network

3.1.1. Hardware and Software Requirements

Every node in the cluster must meet specific hardware and software requirements.

Hardware requirements

The minimum hardware requirements for the nodes in the cluster are listed below:

- CPU - 64-bit computer with Intel Pentium or Xeon family processors with Extended Memory 64 Technology (EM64T) architecture; AMD Opteron or Athlon family processors; other compatible processor(s)
- RAM - 512 megabytes (MB) minimum
- Disk space for setup - 4 GB
- Disk volumes - Head node: two volumes (a system volume and a data volume), Compute node: a single system volume
- Network adapter - Each node requires at least one network adapter (See section 3.1.2 for different network topologies)

Software requirements

The head and compute nodes for Windows Compute Cluster Server can be any of the following operating systems:

- Microsoft Windows Server 2003, Compute Cluster Edition
- Microsoft Windows Server 2003, Standard x64 Edition
- Microsoft Windows Server 2003, Enterprise x64 Edition
- The x64-based versions of Microsoft Windows Server 2003 R2

3.1.2. Network topology

Selecting the appropriate topology depends on the goals for the compute cluster. Windows Compute Cluster Server supports five different network topologies. I will describe in detail two of them, and briefly characterize the rest.

Scenario 1: Two network adapters on the head node, one network adapter on each compute node

In this scenario, the head node provides connection sharing between the compute nodes and public network. The public network adapter on the head node is registered in DNS on the public network, and the private network adapter controls all communication to the cluster. The private network is used for the management and deployment of all compute nodes; it can also be used for high- speed MS MPI computational traffic. This is illustrated in figure 3.2.

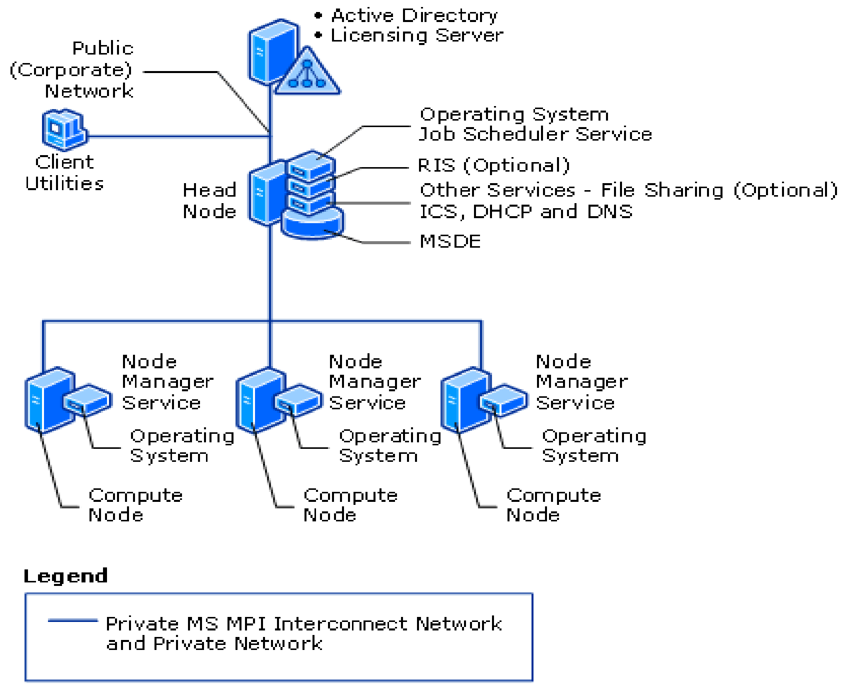


Figure 3.2: Network topology: scenario 1

Scenario 2: Three network adapters on the head node, two on each compute node

This configuration is similar to scenario 1 with one key difference: because each compute node has two network adapters, they have a connection to the private, dedicated cluster network and a connection to a secondary private network running the MS MPI high-speed protocol. The head node provides connection sharing between the compute nodes and the public network. This topology is illustrated in figure 3.3.

Other scenarios

Other scenarios involve topologies where compute nodes are connected to the public network. Apart from the connection to the public network, compute nodes might have a dedicated network for MPI traffic, or simply an additional dedicated private network for cluster management. There might be also a simplified topology, where every node (compute and head) is connected only to the public network, and all communication is carried out in this network.

3.1.3. Creation and submission of jobs

A job is a resource containing one or more computing tasks to be run in parallel. To create a job, first you specify job properties, which are a set of options that determine such elements as job priority, run time limit, number of processors required, specific nodes requested, and whether nodes will be reserved exclusively for the job. Then you add the tasks that the job will execute. In addition to the command line to be executed, each task contains properties that are similar to the job properties. The task's properties also include input, output, and error file streams, as well as a list of any other tasks which this task depends on.
CCS allows users to create different kinds of jobs by combining options for jobs and tasks. The types of jobs that we can create are described briefly below.

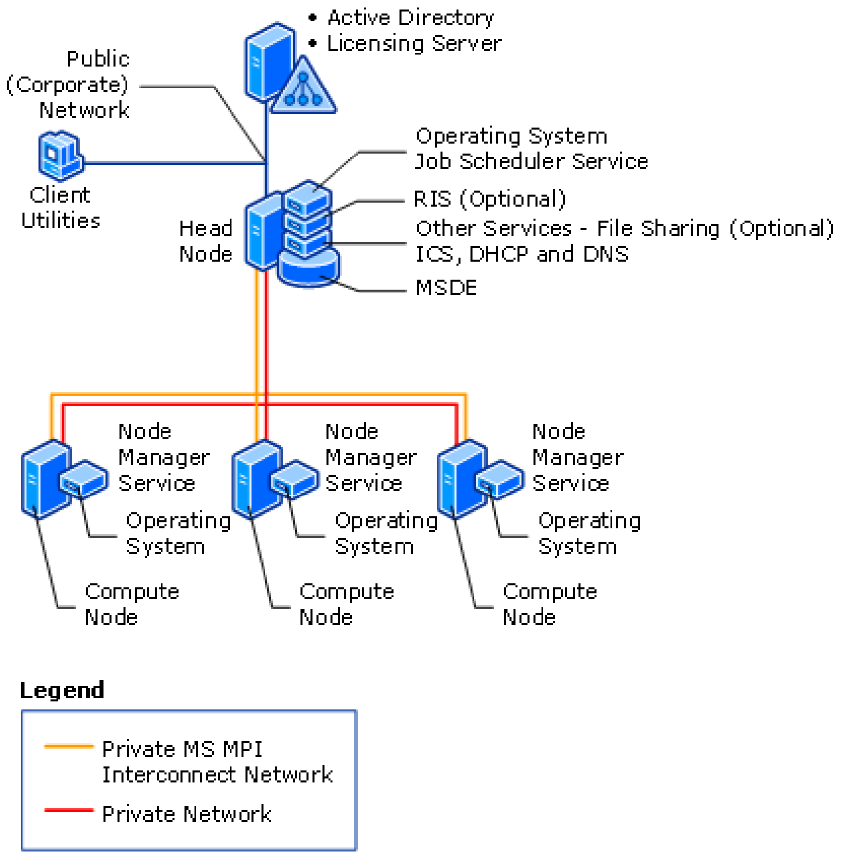


Figure 3.3: Network topology: scenario 2

Job with parallel tasks

A good example of parallel tasks is an MPI application, where a single executable is running concurrently on multiple processors, with communication between processes.

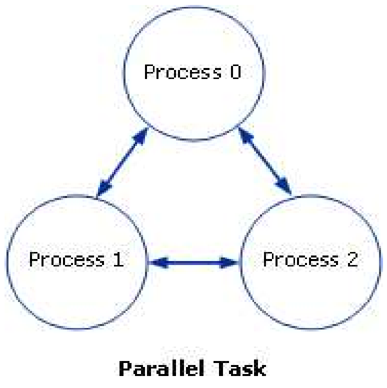


Figure 3.4: Parallel Task

The process numbers 0, 1 and 2 represent the process rank and have greater or less significance depending on the processing paradigm. Very often process 0 handles the input/output and determines what other processes are running. In one paradigm, process 0 also acts as a master process; the others are worker processes that communicate with the master process but not with each other. In another paradigm, the relationship among the processes is symmetrical.

Parametric sweep

A parametric sweep is the concurrent running of multiple instances of the same program with input supplied by an input file and output directed to output file. There is no communication or interdependency among the tasks.

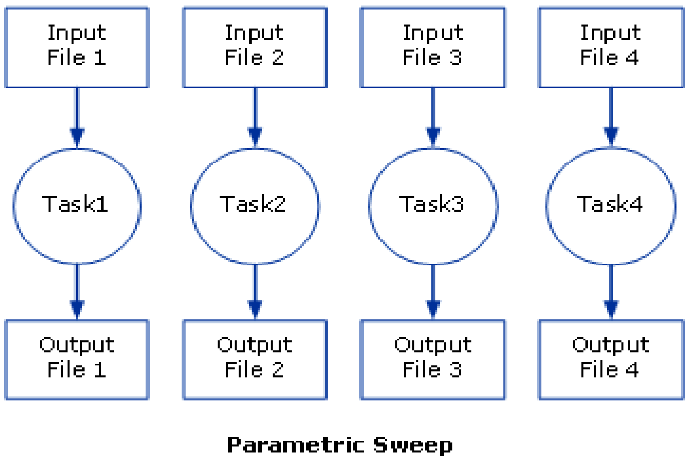


Figure 3.5: Parametric Sweep

Task flow

A task flow job is a job where a set of tasks is executed in a prescribed order, usually because one task depends on the result of another task. A task flow job is illustrated in the 3.6 figure. Note that only Tasks 2 and 3 are performed in parallel, because neither is dependent on the other.

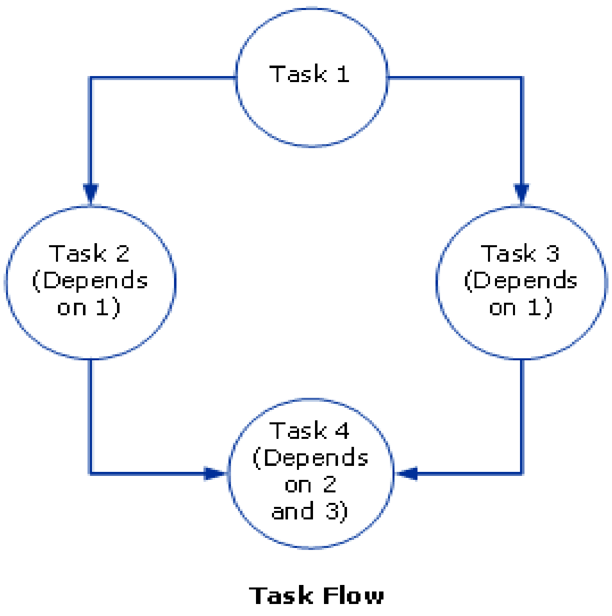


Figure 3.6: Task Flow

3.1.4. Job Scheduler

Job scheduler is a service responsible for managing resources and jobs. When a job is submitted, it places the job in the queue with pending jobs and once the job is on top of the queue, it allocates resources and deploys the job. Jobs are ordered in the queue according to a set of rules called scheduling policies. Resource allocation is based on resource sorting.

Deploying a job means not only dispatching the job tasks to the allocated nodes, but also starting monitoring the nodes, and updating the job status in case of the end of the job or failure.

Job scheduler implements the following scheduling policies:

Priority-based, first-come, first-served scheduling

Priority-based, first-come, first-served (FCFS) scheduling is a combination of FCFS and priority- based scheduling. Using priority-based FCFS scheduling, the scheduler places a job into a higher or lower priority group depending on the job's priority setting, but always places that job at the end of the queue in that priority group because it is the last submitted job.

Backfilling

Backfilling policy aims at taking advantage of unused resources. The Scheduler takes the job from top of the queue and searches for resources to execute that job. If there are not enough resources for the execution, the schedulers puts this job as a future reservation and searches inside the queue for a smaller job that might fit into the currently available resources. What is crucial is that the smaller job must not collide with the future execution of the job from top of the queue. Therefore, it must terminate before the first job is expected to commence.

Non-Exclusive scheduling

By default, a job has exclusive use of the nodes reserved by it. This can produce idle reserved processors on a node. By turning off the exclusive property, the user allows the job to share its unused processors with other jobs that have also been set as nonexclusive. Therefore, nonexclusivity is agreement among participating jobs, allowing each to take advantage of the other's unused processors.

More information on Job Scheduler can be found in [SCH].

3.1.5. Run-time job and task management

Along with job scheduling, Job Scheduler is also responsible for status management of jobs, tasks and nodes.

Job and task status

The following list summarizes the job status flags:

- Not Submitted - Job has been created but has not been submitted.
- Queued - Job has been submitted and is waiting to be activated.
- Running - Job is running.
- Finished - Job has completed successfully.
- Failed - Job has failed to complete.
- Canceled - Job has been canceled.

The task status flags are the same as the job status flags.

Node status

A node can be in the following states:

- Pending - Node has been added to cluster but not yet approved by administrator.
- Ready - Node has been added to cluster and approved by administrator.
- Paused - When a node is in the Paused state, jobs running on the node continue to run but no new jobs from users without administrative rights will be started.
- Unreachable - The system cannot establish a connection with node.

3.2. Hardware configuration of the cluster at INRIA

The cluster at INRIA is composed of 8 nodes and onboard administrator. Each node contains:

- 2 processors Xeon QuadCore E5320
- 8GB RAM
- 2 hard disks SAS 72GB hot plug
- 2 NC373i Gigabit network adapters

There is a private network in the cluster for job management and MPI traffic. The access to the public network is only possible from the head node, thus the compute nodes are not visible from the outside. The topology used is presented in section 3.1.2: *Scenario 1.*

3.3. Software installed on the cluster

The software for cluster and job management is installed on the head node of the cluster. The software includes:

- Compute Cluster Administrator
- Compute Cluster Job Management

Compute Cluster Administrator allows configuration and monitoring of each compute node. Each node can be paused or set ready and the remote desktop connection can be established with the node. The software provides functions to add a node to a cluster or remove it.

Compute Cluster Job Management is a GUI application for adding, cancelling, modifying and monitoring jobs. The other way to create, submit and monitor jobs is via command-line interface. This is a set of commands which relies on user's interaction. The details of how to use command-line interface are presented in [CLI].

Chapter 4

Integration with Microsoft Compute Cluster Server

ProActive library provides developers with mechanisms to distribute their application across many hosts. Developers firstly focus on the algorithmic nature of the application, and then decide how to map the distributed parts of the application onto the computational resources. This mapping is described in XML deployment files. Once the application is ready to be deployed it is the responsibility of the ProActive library to acquire requested resources for application execution. This chapter thoroughly presents the integration of Microsoft Compute Cluster Server into the deployment framework. This integration enables CCS to be used as a resource managed by the ProActive Scheduler, thus serve as a part of a Grid.

Section 4.1 presents the general concept of adding support for a new environment to the deployment framework.

Section 4.2 includes a detailed description of the implementation of CCS integration.

Section 4.3 presents an example of an application running on the Microsoft CCS taking advantage of its computational capabilities.

Section 4.4 contains an evaluation part of the integration.

Section 4.5 presents the real goal of this integration. The integration was sponsored by Microsoft and upon successful completion of this integration Microsoft renewed collaboration with the OASIS team.

Section 4.6 is about deployment process on DAS-2 and DAS-3 distributed computers located at Vrije University in Amsterdam. In this section I compare the resource acquisition processes on DAS and CCS.

The latest version of the ProActive library which contains my contribution can be downloaded from the main ProActive website ([DPR]).

4.1. Deployment Architecture

In chapter 2 there is a description of the deployment framework from an application programmer's perspective. It includes information how to prepare an application to be easily deployable in diverse environments, introduces a concept of a Virtual Node and its relation to a ProActive Node and finally it presents a full example of an application with corresponding XML deployment files.

In this section I would like to focus on the deployment framework from the library programmer's perspective. Namely, what it takes to add support for a new environment in which the applications can be executed.

Deployment framework already supports many environments and configurations, therefore adding support for a new environment involves an implementation of a module which must adhere to the already built structure of the framework. First off, the XML files describe the deployment environment of an application. An application might be executed on a grid (a collection of resources: CCS, DAS, P2P network, etc.). Then, the XML files include deployment parameters for all environments which compose the grid.

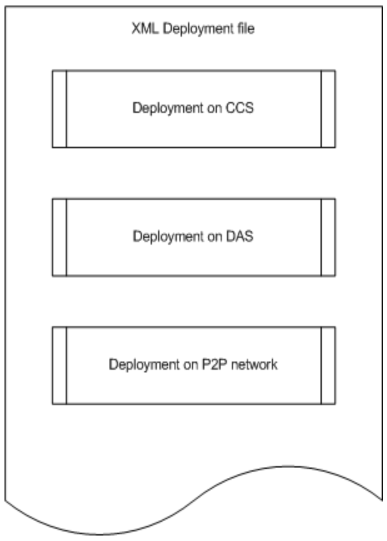


Figure 4.1: XML Deployment file

New environment requires new deployment configuration. For each configuration a module must be implemented in the deployment framework. The module will handle the allocation of resources accordingly to the given configuration. In the case of clusters the deployment configuration usually includes the number of CPUs to allocate, the time of the allocation, output files, etc. For each new deployment environment (like CCS) the XML Schema and XML Parser must be implemented. Schema will check correctness of the deployment configuration and parser will read it into the memory.

The next step in adding support for a new deployment environment is implementing a command builder object. This object must build a command (based on the given parameters) which will be executed on the head node of the cluster. The command must handle the allocation of the cluster resources and the submission of an application. Usually it is impossible to carry out the allocation and the submission just in one command therefore very often the command simply invokes a script. For instance, in the case of deployment on PBS the script will ask for the requested number of resources, receive a list of allocated nodes and execute (via ssh or rsh) an application on each node. The description of the script on CCS is presented in the next section.

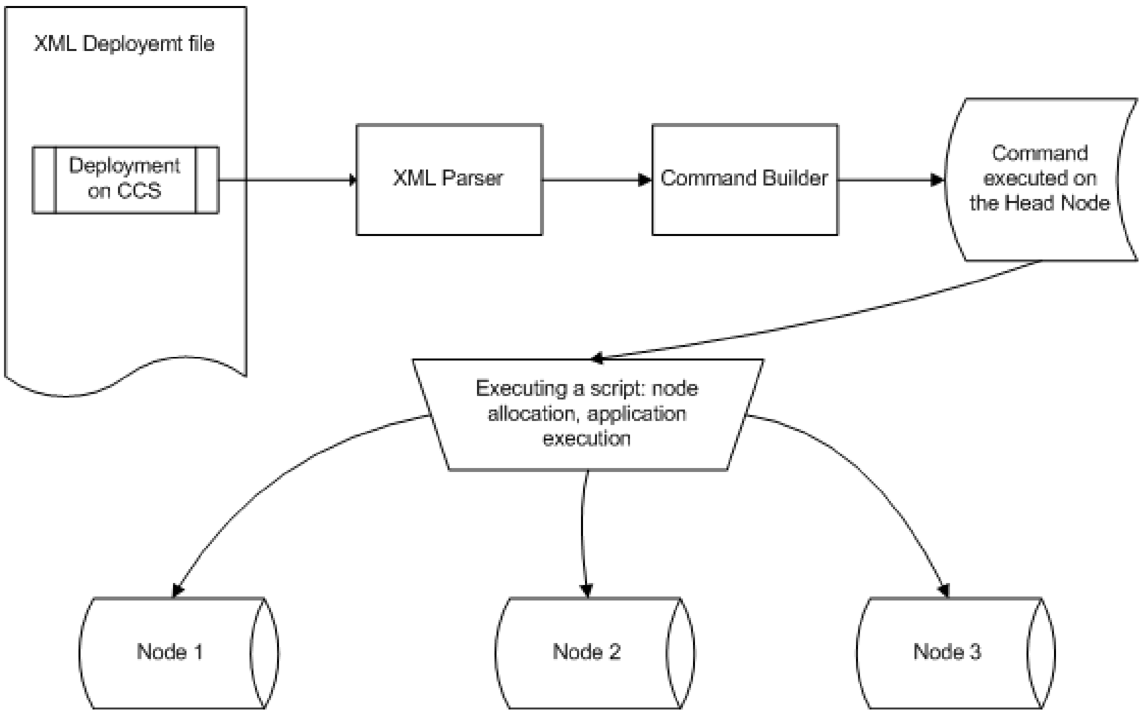

XML Deployemt file
Deployment on CCS
XML Parser
Command Builder
Command executed on the Head Node
Executing a script: node allocation, application execution
Node 1
Node 2
Node 3

Figure 4.2: Deployment process

4.2. Implementation

The XML part (deployment files, XML Parser and XML Schema) is presented in the section *Specification of acquisition* and the allocation and execution process in the *Acquisition of resources* section.

4.2.1. Specification of acquisition

First step in the acquisition of CCS resources is writing XML deployment files. In chapter 2 about the ProActive library there is a complete example of XML deployment files. Here, I will only focus on the part of deployment files which are related to CCS.

In chapter 2 the Master-Worker paradigm is presented. Suppose, we want to deploy the worker processes on the Microsoft Compute Cluster Server. To do that we need to specify the Worker Virtual Node in a deployment file. The definition of a Virtual Node will contain capacity and a Node Provider. The relation between a Virtual Node and a Node Provider will be used as a mapping of the abstract entities in the application (Virtual Nodes) onto the physical machines (ProActive Nodes). The capacity attribute says what is the maximum number of ProActive Nodes which can be retrieved from a Virtual Node.

```
<virtualNode id="Workers" capacity="32">
```

```
        <nodeProvider refid="workers" />
</virtualNode>
<resources>
        <nodeProvider id="workers">
               <file path="WorkersDeploymentCCS.xml" />
        </nodeProvider>
</resources>
```

The Node Provider contains a path to WorkersDeploymentCCS.xml file. This file describes the CCS resources. In the beginning we define the *resources* element. This element informs that we want to allocate a collection of nodes (element *group*) identified by a name *ccs*. Each host in this group is identified by a name *node*.

```
<resources>
       <group refid="ccs">
              <host refid="node"/>
       </group>
</resources>
```

Subsequently, there is a description of a host identified by a name *node* (every compute node in the CCS):

```
<host id="node" os="windows" >
        <homeDirectory base="root" relpath="${user.dir}" />
</host>
```

Finally, the *ccs* group element is presented. It includes a number of CPUs to allocate, maximum allocation time of compute nodes (runtime) and output files. The schema for this element is presented further on.

```
<ccsGroup id="ccs">
       <resources cpus="16" runtime="01:01:01"/> <!-- runtime=days:hours:min -->
       <stdout>\\HeadNode\work\out</stdout>
       <stderr>\\HeadNode\work\err</stderr>
</ccsGroup>
```

XML Schema

The main element is ccsGroup element of type ccsProcessType :

```
<xs:element name="ccsGroup" type="ccsProcessType" ... />
```

The ccsGroupType is defined as follows. It contains 3 subelements: resources, stdout (redirection of standard output to a file), stderr (redirection of standard error to a file).

```
<xs:complexType name="ccsProcessType">
  <xs:complexContent>
      <xs:extension base="abstractGroupSchedulerElementType">
        <xs:sequence>
          <xs:element name="resources" type="ccsResourcesType" ... />
          <xs:element name="stdout" type="pathElementGeneralType" ... />
          <xs:element name="stderr" type="pathElementGeneralType" ... />
        </xs:sequence>
      </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The ccsResourcesType is composed of 2 attributes. The cpus attribute defines how many CPUs the application requires to be allocated on the cluster. The allocation is exclusive, meaning that if there are unallocated CPUs on the node they will not be used by other jobs. The runtime attribute is the maximum execution time of an application. Application which does not finish before the end of its runtime, will be terminated by the scheduler.

```
<xs:complexType name="ccsResourcesType">
        <xs:complexContent>
        <xs:restriction base="xs:anyType">
                <xs:attribute name="cpus" type="PosintOrVariableType"/>
                <xs:attribute name="runtime" type="TimeType" use="optional"/>
        </xs:restriction>
        </xs:complexContent>
</xs:complexType>
```

4.2.2. Acquisition of resources

The parameters from the XML deployment files are passed to a command builder object which will handle allocation of the resources and tasks execution. As presented before, the structure of deployment framework enforces that this object builds a command that will run on a head node of the cluster. In order to allocate the resources on the CCS the communication with the CCS Job Scheduler must take place and different approaches might be applied to carry out this communication.

There is a possibility of using Command Line Interface (CLI) to create a job, add tasks to the job, and submit the job. The disadvantage is that it requires interaction from a user and even if it could be bypassed there are more convenient ways to carry this out. The CCS is provided with an object-oriented API to communicate with the Job Scheduler. The API is accessible in C# and VBScript. I implemented the allocation of resources and submission of jobs in VBScript as it is easier to maintain and does not require a compiler.

Using VBScript on Windows

VBScript (short for Visual Basic Scripting Edition) is an Active Scripting language developed by Microsoft ([SCR]). Active Scripting is the technology used in Windows to implement component-based scripting support. Additionally, when installing Compute Cluster Server, the COM scripting library is installed. This library provides an object model that represents the major elements of a compute cluster: clusters, nodes, jobs and tasks. Each of these major objects includes specific for itself properties and methods that enable configuration changes.

Command Builder: GroupCCS

The GroupCCS object receives parameters from XML deployment files and builds a command that will be subsequently invoked on the head node of the cluster. The command is an invocation of VBScript and is composed of the following parts: an interpreter of Visual Basic (cscript.exe), script name (ccs.vbs) and a collection of options. An example is presented below:

```
csript.exe ccs.vbs /tasks:8 /application:"java StartRuntime"
/classpath:"\\HeadNode\lib\start.jar" /runtime:10:10
/stdout:"\\HeadNode\out\stdout" /stderr:"\\HeadNode\err\stderr"
```

Note that we can pass parameters to the Visual Basic script with corresponding names (tasks:8, application:"java StartRuntime", ...), and then in the script retrieve the parameters using those names. Therefore, we do not care about the order and whether some of the parameters are optional.

ccs.vbs: communication with Job Scheduler

The *ccs.vbs* script is in charge of the creation of a job and its tasks, and a submission of the job to the CCS Job Scheduler. Once the job is submitted for execution, on each allocated compute node a ProActive StartRuntime service will be running which enables creation of a ProActive Node. The interior of the *ccs.vbs* script is as follows.

Creation of compute cluster object, and setting a connection with the cluster:

```
Set objComputeCluster = CreateObject("Microsoft.ComputeCluster.Cluster")
objComputeCluster.Connect(strClusterName)
```

Creation of a new job, setting the requested number of CPUs to allocate, and runtime (maximum execution time of the task):

```
Set objJob = objComputeCluster.CreateJob
objJob.MinimumNumberOfProcessors = cpus
objJob.MaximumNumberOfProcessors = cpus
objJob.Runtime=runtime
```

Creation of tasks, setting parameters, and adding a task to the job:

```
For i = 1 To tasksNumber
      Set objTask = objComputeCluster.CreateTask
      objTask.CommandLine = commandLine
      objTask.SetEnvironmentVariable "CLASSPATH", classPath
      objTask.MinimumNumberOfProcessors = 1
      objTask.MaximumNumberOfProcessors = 1
      objTask.Stdout=strStdout & i & ".txt"
      objTask.Stderr=strStderr & i & ".txt"
      objComputeCluster.AddTask(objJob.ID, (objTask))
Next
```

Finally, submission of the task to the Job Scheduler:

```
objComputeCluster.SubmitJob jobID username password
```

The job will be executed on the CCS, with one task per CPU. Each task will write its output to the files located in the shared folder, which is accessible from a head node.

4.3. Application example

Creating a Blender animation

Blender ([BLE]) is an integrated application that enables the creation of a broad range of 2D and 3D content. Blender provides a broad spectrum of modeling, texturing, lighting, animation and video post-processing functionality in one package. Blender is one of the most popular Open Source 3D graphics application in the world.

Blender on Microsoft CCS

We created a job which was composed of N identical tasks, each invoked with different parameters and one additional task invoked after all previous tasks are complete. Each task (from a set of N identical tasks) was responsible for the generating of a set of picture frames based on the input model file. The goal of generating picture frames was to produce an animation. The task invoked in the end was in charge of the collection of the results (frames) from other tasks, merging the results, and creating a final animation.

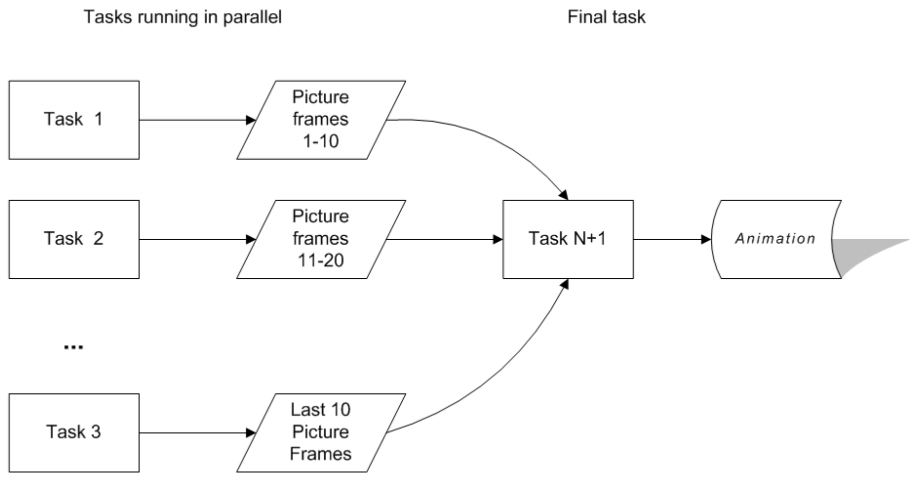


Figure 4.3: Blender on Microsoft CCS

4.4. Evaluation

The evaluation of the CCS integration into the deployment framework was judged by a few criteria. These are:

- Portability - whether the solution is portable within Windows machines serving as a part of a cluster
- Flexibility - whether the solution applied is configurable and whether it imposes any constraints with respect to the deployment process
- Extensibility - whether the solution can be easily extended to support new deployment options or change of the deployment process
- Performance - whether the solution is scalable with respect to the allocation of increasing number of compute nodes

4.4.1. Portability

The solution I have applied involves Java and VBScript. Also, I use the Microsoft API to communicate with the CCS Job Scheduler. VBScript is installed by default in every desktop release of the Windows Operating System since Windows 98. The API to communicate with the CCS Job Scheduler comes with the software which is installed on the head node. The job management and job scheduling would not be possible without installation of this software. Thus, I consider the VBScript and API portable within Windows machines which can serve as a part of a cluster. Last thing to consider is Java which is not installed by default on Windows machines. When the number of compute nodes is small (for instance 8 nodes), the manual installation of Java on each compute node is simple and relatively quick. When the number of compute nodes in the cluster is substantial (for instance 1000 nodes) another approach can be applied. The image of the operating system (with accompanying software) can be prepared and installed (perhaps remotely) on each compute node. The image should already include Java. This way, Java will be installed together with the Operating System, so the installation process is simplified and perhaps automatized.

4.4.2. Flexibility

The communication with the Job Scheduler is carried out by using the CCS API in the VBScript. As presented in the implementation section, it requires creation of a job object, setting job options, adding tasks to the job, and finally submitting the job. Although it is flexible with respect to the specification of options and combining different API functions, but it does not allow to submit jobs in a PBS-like manner. The PBS-like submission process looks as follows. We ask the scheduler for a collection of resources (without specifying the application which will run on those resources). The scheduler provides us with a list of allocated nodes (by a name or IP address). By ourselves, we log in on every machine and start the application (either via ssh or rsh). This gives us more control over the deployment process. For instance, if the allocated node has multiple CPUs (each double or quad core) we can decide to start more than one application or if our application is already multithreaded then we can deliberately start only one application on one node. Such deployment process is not possible with my solution, but this limitation is mostly imposed by the API itself, and not specifically by my approach. The API enforces to ask for a collection of resources and submit application at once.

Another limitation is a network topology. Very often network topologies in clusters are organized as follows. The head node is in the public network and compute nodes in the private network (sometimes with a dedicated high-speed network like Myrinet for internal communication). This means that the network connection with any compute node can be established only from a head node. In our case, this poses a big limitation, as we want to allocate the compute nodes (via CCS Job Scheduler) and then use them directly as our own resources. To accomplish that, two approaches might be applied but none of them is perfect. First one is simply enforcing a network topology where compute nodes are in the public network, thus the communication with each of them is possible from the outside. This might not be a good solution in terms of the network security or in terms of using too many public IP addresses for the single cluster. Another approach, currently applied, is running the part of the ProActive Scheduler on the head node. The part is specifically responsible for the resource management, and as it runs on the head node it has access to the compute nodes. This approach is also not perfect as it involves starting the resource management process on the head node. The perfect solution would involve significant changes in the ProActive Scheduler.

4.4.3. Extensibility

My solution is well integrated into the ProActive deployment framework. A user specifies deployment options in the XML files. The XML Parser (together with XML Schema) checks for errors and passes the options to the command builder. The command builder builds a command which is run on the head node. The command itself is the invocation of the VB-Script which handles communication with the CCS Job Scheduler.

There might be a need in the future to extend the set of available options. It requires modifications in every step of deployment process presented above. However, every of that modification is clear and well-defined (i.e. there is only one way to introduce this modification). Such modification involves:

- extending XML Schema to support new option
- adding support for the new option in the XML Parser
- translating the option into the option understandable by the Job Scheduler (this is done by the command builder) - sometimes it will require additional handling as we want to provide users with the simplest and unequivocal interface which might not directly correspond to the Job Scheduler interface
- modifying the VBScript to pass this option to the Job Scheduler when submitting a job

Another need that might arise is a change of a job submission process. Here, the extensibility depends on our goals. As long as we still use a script (not necessarily the VBScript) which runs on the head node and submits a job, my solution is still easily extensible. For instance, if we want to submit jobs in a PBS-like manner, it will require only the modifications in the script. However, if we want to change the idea that stands behind the deployment process, then it could be hardly (if at all) possible. For instance, if we want to specify deployment options in the XML files and then use a Web Service to communicate with the Job Scheduler, it would require substantial changes. These changes would be enforced not by my solution itself, but by the architecture of the deployment framework.

4.4.4. Performance

The integration with CCS enables to use its resources as a part of a grid managed by the ProActive Scheduler. The fundamental idea is to use long-term reservations of the resources and short-term jobs scheduling on the allocated resources. The performance of a job is mainly dependant upon two factors:

- the job itself - how well the application is written - includes many factors like algorithms used or communication overhead
- scheduling - whether the application's computational or communicational needs are well adjusted to the resources capabilities. The mapping between a job and resources is created by a scheduler.

My integration cannot improve or deteriorate any of those factors. The integration enables long-term reservation of resources and does not have an impact on the scheduling or executing part. The details of the reservation process by the ProActive Scheduler are in the section 5.2. Here, I will only briefly describe this process:

- reading reservation options from an XML file
- communicating with the CCS Job Scheduler by submitting a job with N ProActive Nodes as tasks
- waiting until each ProActive Node declares its readiness to execute user jobs

Once the reservation process is complete, a dedicated program (ProActive Node) is running on each allocated node and it will handle in the future executions of user tasks.

The performance of my solution can be measured by the time to place a reservation. As described in section 3.2 the cluster I used was composed of 8 nodes where each node has 2 processors (each quad core). Summing up, there were 64 processing units at my disposal.

I measured reservation time for 4, 8, 16, 32 and 48 processing units. Thus, there were respectively 4, 8, 16, 32 and 48 ProActive Nodes running on the cluster. I was unable to carry out the tests for the maximum (64) number of processing units because of the unavailability of the machines at that time. What is important is that I performed those tests when enough resources were available, otherwise we would measure how long the reservation spent in a queue with pending reservations. This was not my goal.

The results of the test are as follows. The X-axis says how many processing units were requested. The Y-axis says how much time in seconds the whole reservation process took. The time includes all three steps described above (reading XML, communication, declaring readiness).

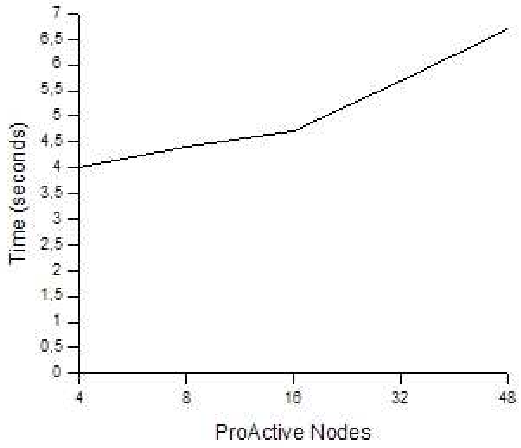


Figure 4.4: Performance of reservation process

First step (reading XML file) always takes the same amount of time as it only reads options into the memory (does not matter whether we request to start 4, 8 or 16 ProActive Nodes). The second step is the communication with the CCS Job Scheduler. Here, also the step takes always the same amount of time as we only pass options to the Scheduler. Next step is allocation of resources (we assume there are enough resources available at that moment). This is the first factor that contributes to the performance - it takes less time to allocate 4 nodes than 32 nodes. Then, on each allocated compute node a certain number of ProActive Nodes is started. ProActive Node is simply a Java application, thus starting one Java application takes less time than starting 2 (or 4,8) Java applications. Each compute nodes has 8 cores, thus it can host up to 8 ProActive Nodes. Basically, the more applications we schedule on one compute node, the more time it takes to launch those applications. In case of starting 48 ProActive Nodes, every node is used in the maximum way (the cluster has 48 cores). Thus the allocation time is slightly higher.

Performance assessment

The ProActive Scheduler is a high-throughput computing (HTC) software. HTC aims at the reservation of resources over long periods of time to execute computational tasks. As opposed to high-performance computing (HPC), HTC needs to allocate the resources for months rather than hours or days. The performance of high-performance computing systems is measured in FLOPS (FLoating point Operations Per Second), whereas the performance of high-throughput computing systems is measured in completed jobs per month. The resemblance that the HPC and HTC bear is that they both need resources with substantial computational capabilities.

HTC systems are usually used to submit jobs with parallel tasks, where the system may schedule the tasks on the "near" resources to decrease the communication overhead. As the jobs may take weeks or months to complete, HTC must provide additional features such as job monitoring, restarting tasks in case of a machine failure (or even better - checkpointing tasks - to restart from a certain stage), managing dependencies between tasks etc.

Taking into consideration the described character of the ProActive Scheduler, the time of the initial allocation of the resources may seem negligible (within reason). Having in the perspective jobs which take weeks or months to complete, the allocation process which takes 7 seconds (in case of allocating 48 cores from the CCS) should be totally acceptable.

4.5. Collaboration with Microsoft

The integration of the Microsoft Compute Cluster Server into the ProActive Deployment Framework was carried out in collaboration with Microsoft which provided scholarship to sponsor my work. During the spring semester the Microsoft experts from the office in Paris visited us twice to assess my progress. Firstly, they came in mid-March when the integration was at the initial stage. At that time I was searching for the best solution to implement the integration. I was investigating different approaches with respect to the communication with the CCS Job Scheduler. I considered using approaches such as Command-Line Interface or passing job description as an XML file. Finally (as described in this chapter), I have decided on using the Visual Basic Script API to communicate with the CCS Scheduler. The first visit of the experts was the perfect occasion to discuss my initial observations, problems I have encountered and suggestions for future implementation. During the implementation phase I was in contact with the Microsoft technical support which I could ask about issues that were not or were vaguely presented in the documentation.

The second visit of the Microsoft experts took place in the late May when I was ready to present the integration. The experts were curious about the approach I have applied, and the solution which uses VB Script API was completely acceptable by them. Upon this acceptance was dependant future collaboration between Microsoft and the OASIS team. To support my solution I helped to port a few internal applications to start using Microsoft Cluster, so we could present to the experts running computation-intensive tasks on the cluster. The Blender application presented in this chapter is one of them. Upon successful presentation of my solution, the Microsoft renewed the collaboration and offered 1000 CCS licences at 1 Euro each. My solution will be in the near future deployed on 1000 machines (on the Grid 5000 grid computing infrastructure) and will enable scientists to schedule time-consuming or computation-intensive applications on Microsoft CCS.

4.6. Comparison of the integration with DAS2 and DAS3

The main goal for adding support for DAS clusters (located at Vrije University) was to become familiar with the architecture of the deployment framework. I implemented support for DAS2 and DAS3 prior to the integration with the CCS as it seemed to be an easier task and in fact it was. Here, I would like to briefly present the main differences of the allocation processes on CCS and DAS clusters.

Integrating DAS clusters into the ProActive deployment framework was considerably simplified by a prun program (provided on the head nodes of DAS2 and DAS3 clusters). It reserves a requested number of CPUs (or nodes) and executes a parallel application on them. As explained before, during the allocation on clusters the deployment framework builds a command which is subsequently passed for the execution on the head node of the cluster. In the case of CCS the command contains an invocation of a VBScript with parameters. The script then uses the cluster dedicated API to communicate with the CCS Job Scheduler in order to submit a job. In the case of DAS clusters it is enough to translate deployment options into prun options and build a command that will directly invoke prun. Prun will take care of the allocation of the nodes and starting an application on each allocated machine. In fact prun is built on top of Portable Batch System [PBS]. Prun will, behind the scenes, ask PBS for the specific number of nodes and will receive a list of nodes in response (given by names or IP addresses). Subsequently it will use Remote Shell (rsh) to execute a submitted application on each allocated node.

Chapter 5

ProActive Scheduler

ProActive Scheduler aims at providing heterogeneous resources to the users in a transparent way. The Scheduler can manage simultaneously nodes from clusters, P2P networks, grids and other diverse environments if only they provide a way to run a Java application. A scheduler which aims at the above is often called in the literature a meta-scheduler ([Sch02]).

This chapter presents my contributions which involved a job management part of the Scheduler. In the first section the existing architecture of the Scheduler is presented crucial to understand my contributions. In the next section there is a description how the Scheduler controls the nodes on the Microsoft Compute Cluster Server. Last two sections present my contributions: Forked Java Task and Task Walltime.

One of the users of the ProActive Scheduler is Amadeus IT - provider of air travel reservation systems for airlines like Air France, Iberia and Lufthansa. The latest release of the ProActive Scheduler can be downloaded from ([DSC]). In June 2008 my implementations were incorporated into the trunk version of the Scheduler.

5.1. Architecture

The ProActive Scheduler is composed of two major components: a component responsible for job management (scheduling, executing) and a component responsible for the management of resources (acquisition, monitoring). The job management component is specifically in charge of accepting user jobs for execution, managing a queue with pending jobs, sorting jobs in the queue according to a scheduling policy, scheduling jobs on the allocated resources and returning results of the jobs to the users. The resource management component allocates and monitors nodes which can be obtained either from hosts, clusters, or P2P networks. When allocating a new machine which will serve as a scheduler resource for task execution, an instance of a ProActive Node is created on that machine (we assume there is a direct network connection with an allocated host). Once the ProActive Node is up and running it can host active objects. To execute a user task on an allocated machine a task-dedicated active object is created on the ProActive Node. The active object has full information how to execute the task. As we can schedule different types of tasks (with different parameters), a new active object is created each time. When a task is complete, the result is sent back to the user, and the active object is no longer used. Of course, the ProActive Node is still up and running awaiting subsequent task submissions.

Types of tasks

A task is the smallest execution unit. A collection of tasks is called a job. Each task in the job might be of different type:

- a Native Task - operating system native application
- a Java Task - Java class executed either in the current JVM or a dedicated JVM
- a ProActive Task - a task representing a ProActive application

A job and each type of a task has its Java class equivalent in the ProActive Scheduler source code. A job object contains a collection of task objects and attributes describing a job like a name, description, priority and many others. A task object may contain for instance an execution path (in case of a native task), or a Java class (in case of executing a Java class), parameters passed to an application, task description.

Job and task objects are created instantly when a job is submitted. The XML file with the job (and its tasks) description is sent to the Scheduler, which delegates it to the JobFactory object. JobFactory parses the XML file and creates an object representation of a job.

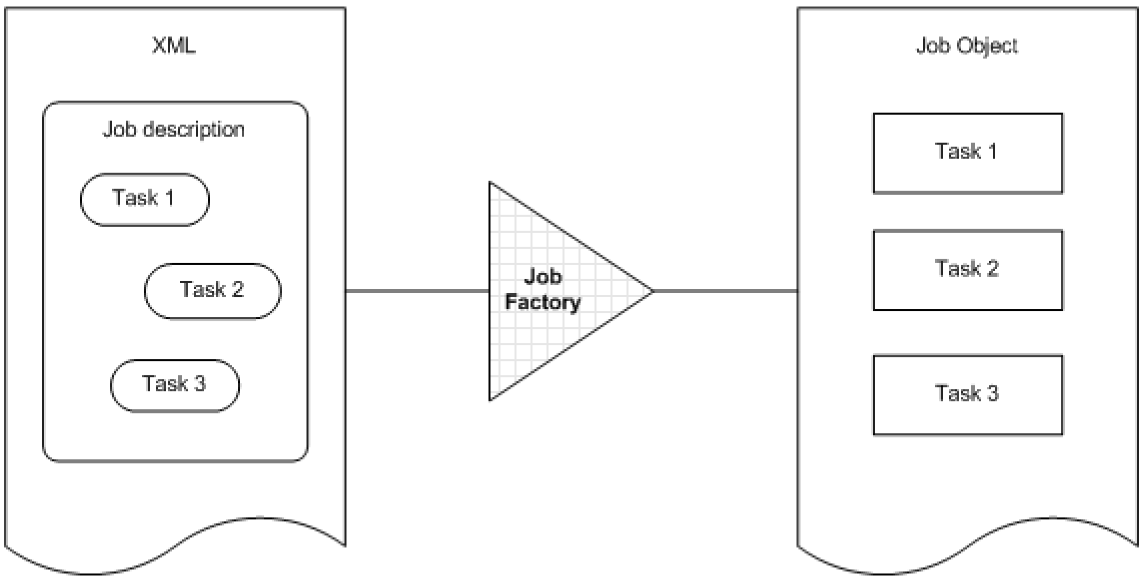


Figure 5.1: Job Factory in the Scheduler

Internal representation of tasks

The task object itself does not have knowledge about the way of its execution. It is only the description of a task from a user's perspective. Therefore, an internal representation of a task is created which has information how to execute it. This internal representation is a class hierarchy which resembles a Task class hierarchy; so there is an Internal Native Task, Internal Java Task and Internal ProActive Task.
The internal task is capable of creating two further objects, which in fact will manage the execution:

- a Task Launcher - an abstract class for different types of tasks, there is a Task Launcher of a Native Task (Native Task Launcher), of a Java Task (Java Task Launcher), of a Java Task in a dedicated JVM (Forked Java Task Launcher) and of a ProActive Task (ProActive Task Launcher)
- an Executable - an abstract class for different types of executables

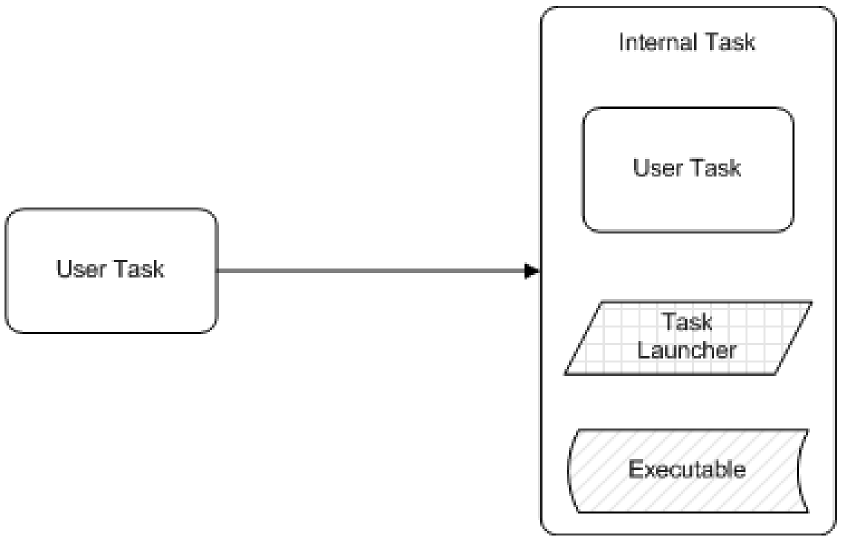


Figure 5.2: Creation of an internal task

Task Launcher will prepare an execution environment for an Executable. This preparation might involve for example invoking pre-execution scripts or setting up redirection of a standard output or error of a native task. When the preparation phase is complete, Task Launcher will prompt an Executable to start executing the task. Once the task is finished the result will be returned to the Task Launcher and then subsequently to the Scheduler. Task Launcher is also in charge of performing any cleaning operations that the task might require. Complementary to each type of a task there is:

- a Native Executable - executable object for native applications; this executable will create a new process which will run a task
- a Java Executable - executable of a Java task
- a Forked Java Executable - executable of a Java task in a dedicated JVM; see details in section 5.3
- a Proactive Executable - its execution is defined by a Java class extending the class org.objectweb.proactive.extensions.scheduler.common.task.executable.ProActiveExecutable, which defines the deployment and the execution of a ProActive application.

5.2. Integration with Microsoft Compute Cluster Server

The goal of the integration of Microsoft CCS into the ProActive deployment framework was to be able to use this cluster as a part of resources managed by the ProActive Scheduler.

Then the CCS not only provides its computational capabilities but also provides an availability for deployment of Windows native applications. As explained in the introductory part, the ProActive Scheduler enforces allocation of compute nodes in advance. Therefore, it communicates with the CCS Job Scheduler only once, in the very beginning, and submits a job which in fact will enable the use of CCS nodes as its own resources. This job has usually no runtime limit, and is composed of N identical tasks (N might be equal for example to the number of compute nodes, or the overall number of CPUs in the cluster). Each task runs a service which enables to create a ProActive Node on that node. Once the ProActive Node is up and running it can host active objects (TaskLauncher and Executable) responsible for user tasks execution. The reservation process resembles the gliding-in mechanism in Condor described in section *Related Work*.

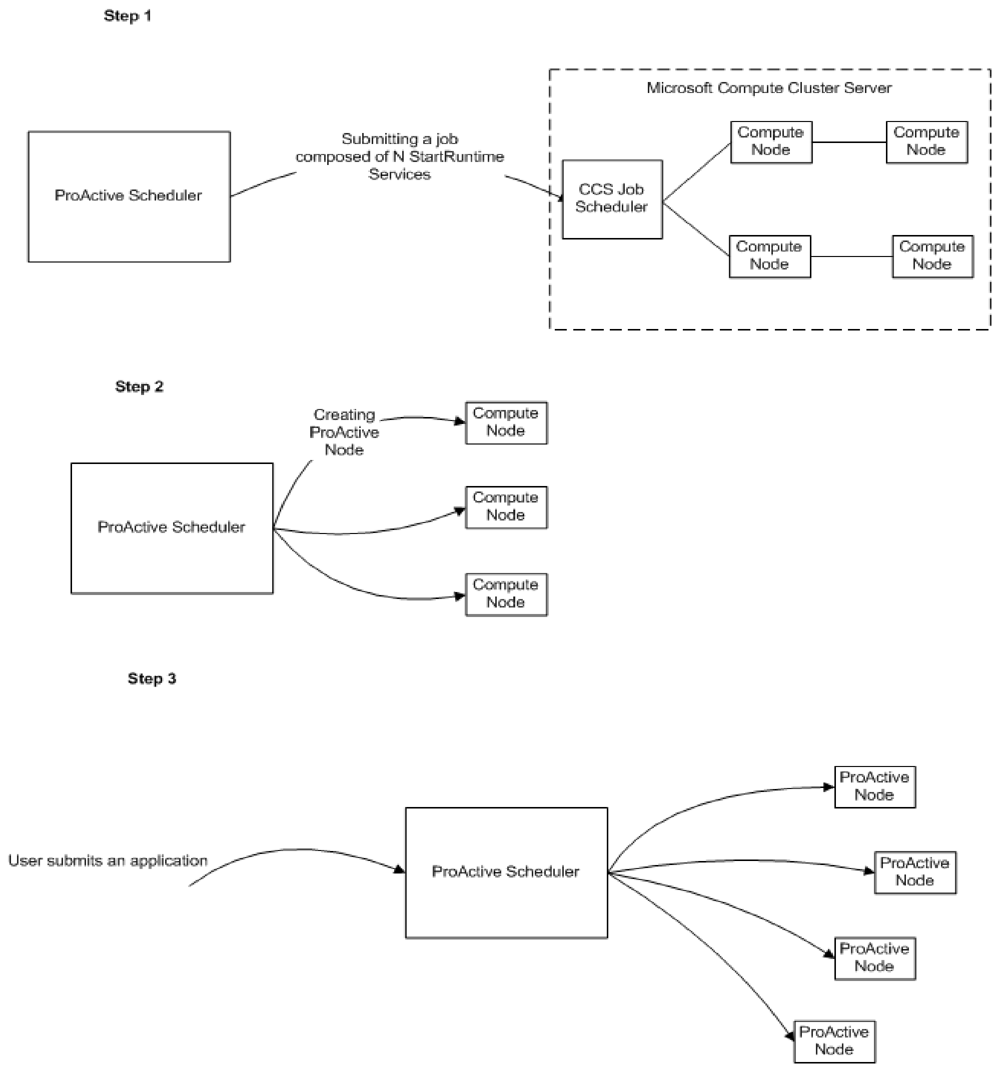


Figure 5.3: Integration with Microsoft CCS

In case the job scheduling is about to finish, the ProActive Scheduler will shut down its resources, including terminating ProActive Nodes on the CCS. Then the CCS Job Scheduler will detect that its job is complete and will release the resources in the cluster for future use.

5.3. Forked Java Task

A user submits a job which is composed of a collection of tasks. Each task might of different type (native, java, proactive) and have different properties. My role was to implement a new type of a task Forked Java Task.

The purpose of implementing a new type of a task Forked Java Task was to gain more flexibility with respect to the execution environment of a task. Previously, all Java tasks where executed in the same JVM and all were dependant upon the same Java options. Now, the new JVM is started with an inherited classpath and (possibly) redefined java home path and Java options. It allows to use a JVM from a different provider and specify options to be passed to JVM (like memory usage).

A Forked Java Task actually contains a usual Java Task but the execution of a task takes a different course. Java Task is simply a class name with parameters. An example of a description of such task is below. It computes an approximation of PI:

```
<task name="Computation">
        <description> Compute Pi </description>
        <javaExecutable class="org.proactive.extensions.scheduler.examples.Pi">
                <parameters>
                       <parameter name="steps" value="2000"/>
                       <parameter name="iterations" value="10000"/>
                </parameters>
        </javaExecutable>
</task>
```

Given the above Java Task we can specify a different course for execution of the task by adding a fork environment. This environment enforces that the task will be executed in a newly created, dedicated JVM (hereinafter referred to as Java child runtime). Using this environment we can specify additional options like java home and Java options. The example is as follows:

```
<javaExecutable class="org.proactive.extensions.scheduler.examples.Pi">
       <forkEnvironment javaHome="/auto/user/jkrzemin/opt/jdk1.6.0_04/jre/"
                        javaOptions="-Xmx128m" />
       <parameters>
       ......
</javaExecutable>
```

When the job is submitted for execution it is placed in a queue with pending jobs and waits for its turn to execute. Once it is on top of the queue, the job's tasks are executed on the allocated resources. When it comes to a task execution two objects play a major role: Task Launcher and Executable. As described in the (previous) Architecture section, Task Launcher performs preparations, then calls an execute method on an Executable, and afterwards does the cleaning operations and returns a result. In pseudocode it looks as follows:

```
public TaskResult doTask (Executable executable) {

      // step 1: preparations
```

```
        performPreExecutionScripts();

        // step 2: execution
        TaskResult result = executable.execute();

        // step 3: cleaning
        finalizeTask();

        // step 4: return a result
        return result
}
```

When executing a simple Java Task everything takes place in a JVM which is running constantly on the node managed by the Scheduler. Task Launcher performs its preparations and then Executable performs Java function in this JVM. This changes with a Forked Java Task, where a task is no longer executed in the same virtual machine. However, we still want to maintain the same behavior as executing a simple Java Task: a Java Task Launcher and an Executable run in the same JVM and Task Launcher prompts an Executable to start the execution. To accomplish the above subsequent steps are applied:

- Forked Java Task Launcher and Forked Java Executable are created in the current (parent) JVM
- Java Task Launcher is an active object created in the forked (child) JVM - as every active object it is accessible remotely by a Proxy object
- Java Executable is created in the parent JVM and passed to the Java Task Launcher. As described in chapter 2 about ProActive library, standard Java objects are passed by deep-copy

Forked Java Task Launcher sets up a new execution environment - it creates a new JVM process and then creates a ProActive Node on that JVM. Creating a new JVM process means running a ProActive StartRuntime routine. This routine enables creation of ProActive Nodes, but also we can ask it to register itself in the JVM by which it was created (parent, current JVM). This registration lets us make sure that the StartRuntime routine is running properly. After setting up a new execution environment an active object Java Task Launcher is created on the newly created ProActive Node. Subsequently, a Forked Java Executable is created locally and gets a Java Executable object (the only object that can carry out the task). When Forked Java Executable is prompted to start the execution it delegates it to the active object Java Task Launcher (on a child JVM). During delegation a Java Executable is passed to the Java Task Launcher and execution takes place in the child JVM. Thus, we achieve the desired goal: Java Task Launcher and Java Executable reside in the same JVM and execute the task. Also, in the current JVM there is a Forked Java Task Launcher and Forked Java Executable so from the Scheduler perspective the architecture of processes remains untouched.
The pseudocode with comments is below:

```
public TaskResult doTask (JavaExecutable executable) {

        // step 1: create new JVM process with ProActive StartRuntime service
        Process process = createJVMProcess("java StartRuntime");
```

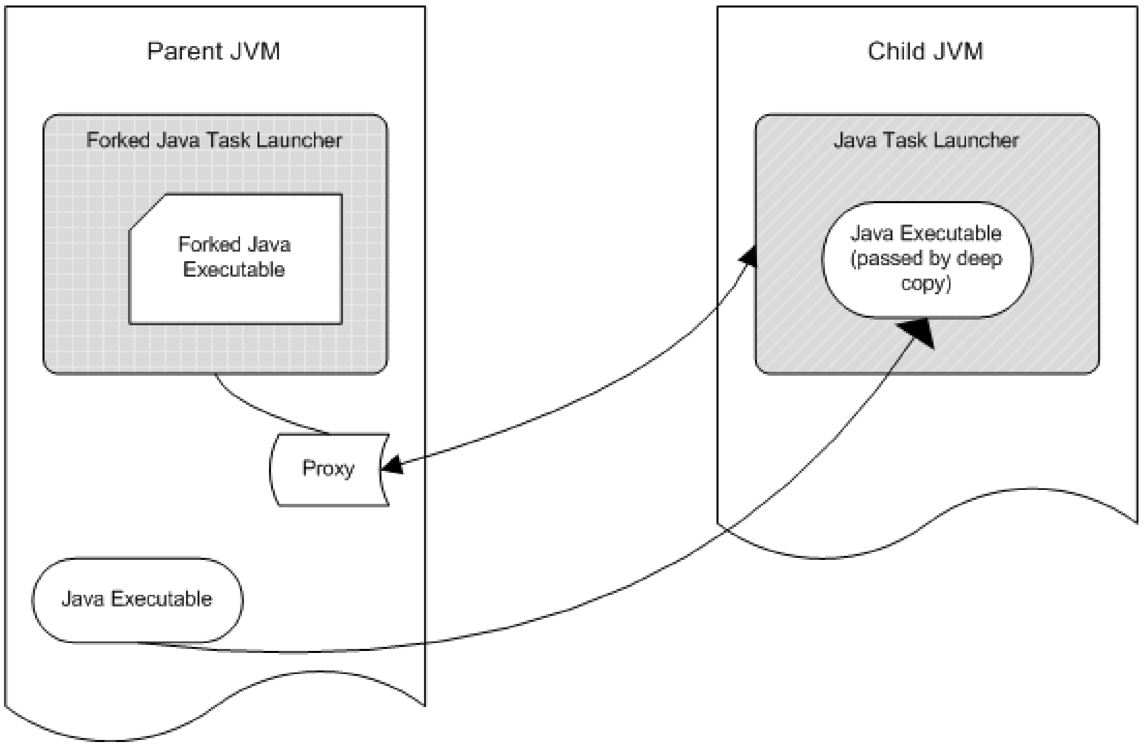


Figure 5.4: Parent and Child JVMs

```
      // step 2: wait until the newly created JVM registers itself
      // at the current JVM
      waitForRegistration();

      // step 3: creation of a ProActive Node on a newly created JVM
      Node node = createNewNode( process );

      // step 4: creation of a remote active object (created on a remote node)
      TaskLauncher launcher = ProActive.createActiveObject(
                                  JavaTaskLauncher.class, node);

      // step 5: creation of a forkedJavaExecutable with a new taskLauncher
      // and a Java Executable
      ForkedJavaExecutable forkedJavaExecutable =
                             new ForkedJavaExecutable(launcher, executable);

      // step 6: task execution
      TaskResult result = forkedJavaExecutable.execute();
      return result;
}
```

As mentioned before, the execution is delegated to the Task Launcher, the result of which is awaited and once it arrives is forwarded to the user. The simplified version of execute() method in ForkedJavaExecutable class looks as follows (the interesting parts of this method are described in the next section Task Walltime):

```
public TaskResult execute() {
        // private member taskLauncher is an active object, residing on
        // a child JVM, javaExecutable is passed by deep-copy
        TaskResult result = taskLauncher.doTask (javaExecutable);
        return result;
}
```

Registration of Java Child Runtime

Before creating a ProActive Node on the child JVM, we must be sure that the JVM started correctly. As stated before, to start a JVM means to start a ProActive StartRuntime routine, which we can ask to register itself in the parent JVM once it is up and running. For the correct registration the special number called deploymentID is used. When starting a process the deploymentID is passed (so both processes - parrent and child know this value):

```
process = Runtime.getRuntime().exec("java StartRuntime -d 123456789");
```

The child JVM notifies the parent JVM that it is running and passes its deploymentID. When a parent JVM receives a notification it checks deploymentID to make sure that it is receiving the notification from a proper process. Then the parent JVM can carry on with its routine of creating a ProActive Node and subsequently Task Launcher active object.

To handle registration the Forked Java Task Launcher (residing in a parent JVM) creates a RegistrationListener which extends a NotificationListener class (from a standard Java library). Then it stops waiting on a semaphore to be realeased. The RegistrationListener overrides a handleNotification method, which is presented below. It checks the notification type and if it is a registration notification it checks the deploymentID. When the deploymentID matches with its own saved deploymentID it will release the semaphore allowing Forked Java Task Launcher to continue:

```
@Override
public void handleNotification(Notification notification, Object handback) {
   String type = notification.getType();

   // if type of a notification is a registration type
   if (NotificationType.GCMRuntimeRegistered.equals(type)) {

      // get data passed
      GCMRuntimeRegistrationNotificationData data =
         (GCMRuntimeRegistrationNotificationData) notification.getUserData();

         // check if deploymentID matches
         if (data.getDeploymentId() == getDeploymentId()) {

            // save a handle to a child runtime process
            childRuntime = data.getChildRuntime();

            // Forked Java Task Launcher waits on a semaphore,
            // so we can realease it now
            semaphore.release();
            return;
         }
     }
 }
```

As stated above, the Forked Java Task Launcher waits on a semaphore to be released after the registration is complete. In fact, if the process simply invoked a method:

```
semaphore.acquire();
```

it could lead to potential problems when registration fails: the child JVM did not start properly or crashed suddenly. To prevent the Forked Java Task Launcher process from hanging (when waiting on a semaphore which will never be released) the below primitive was implemented. It tries in a loop to acquire a semaphore with a timeout. When timeout expires (so no registration was made in the meantime) it checks the process status. The process status is checked by calling an exitValue() primitive. If it still running then exitValue() will throw an IllegalThreadStateException (as we cannot check the exitValue of a running process). So we try to acquire the semaphore again (and so we does for a couple of times). In the end either the child process registers itself or we give up and decide to exit by throwing an exception.

```
private void waitForRegistration() throws SchedulerException {
   int numberOfTrials = 0;
   for (; numberOfTrials < RETRY_ACQUIRE; numberOfTrials++) {
       boolean permit = semaphore.tryAcquire(SEMAPHORE_TIMEOUT);
       // if semaphore was not acquired permit is false
       if (permit)
           break;

       try {
            process.exitValue();
            // process terminated, no registration made, throwing an exception:
            throw new SchedulerException(
                      "Unable to create a separate java process");
       } catch (IllegalThreadStateException e) {
         // if we get here it means the process has not finished yet
       }
   }
   if (numberOfTrials == RETRY_ACQUIRE)
      throw new SchedulerException(
                "Unable to create a separate java process");

}
```

The reasons of registration failure might be different. When a user specifies javaHome or javaOptions in the description of a task he or she can make a mistake which will prevent from a proper process execution. To have any information about what might have gone wrong, we intercept the standard output and error of a process and redirect it to the scheduler. For the interception a ThreadReader object is used. ThreadReader is an object executing in a separate thread, which the only responsibility is stream redirection. It reads data from one stream and prints it out to another :

```
public void redirect (InputStream inputStream, OutputStream outputStream) {
     String line = null;
      while ( (line = inputStream.readLine()) != null ) {
            outputStream.writeLine(line);
      }
}
```

Now, when creating the StartRuntime process we need to take care of creating two ThreadReaders objects to intercept the standard output and error:

```
private void createJVMProcess(String command) {
      process = Runtime.getRuntime().exec(command);
      BufferedReader sout = new BufferedReader(
                                new InputStreamReader(process.getInputStream()));
      BufferedReader serr = new BufferedReader(
                                new InputStreamReader(process.getErrorStream()));
      tsout = new Thread(new ThreadReader(sout, System.out));
      tserr = new Thread(new ThreadReader(serr, System.err));
      tsout.start();
      tserr.start();
}
```

The System.out and System.err in the above code will be redirected to the Scheduler.

Cleaning

No matter whether the execution of a task succeeds or fails (either because of errors during registration or a walltime mechanism described in the next section) the Forked Java Task Launcher must shut down the child environment. In this case cleaning primitive will involve killing all ProActive Nodes on the child JVM runtime, destroying a JVM process itself, and waiting for ThreadReaders to finish (once the process is over the stream will be closed and ThreadReader will detect it):

```
protected void clean() {
        try {
                childRuntime.killAllNodes();
                process.destroy();
                tsout.join();
                tserr.join();
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

5.4. Task Walltime

Task Walltime is a maximum execution time of a task. There might be different reasons for introducing walltime into a task description. For instance on a cluster used by students an administrator might enforce task walltime to provide equal (in terms of time) access for every student. Also when a user knows roughly the expected execution time of a task he or she can specify it in a task descriptor as a favor to other users, so in case application hangs it will be eventually terminated by a scheduler. Task Walltime is also useful in implementing sophisticated scheduling policies like backfilling (discussed in chapter 3), which are based on reservation systems and the maximum execution time of a task is a must.

Walltime can be specified for any task, irrespectively of its type. For native tasks it is relatively easy to terminate a task which is in a state of execution. We assume that for a native task we have a reference to a Process object (from Java standard library), which we can simply terminate. For a Forked Java Task the solution is a bit tricky (described further on) but still relatively easy; it is enough to terminate a ProActive Node service running a dedicated JVM (then the task stops), and subsequently terminate the JVM process. For a Java task it looks more complicated though. An execution of a Java task is simply a thread invoking certain function. As stopping a thread is deprecated there is a possibility only for one mechanism: having a variable which signifies the end of walltime. This variable must be accessible to the execution thread and it is the responsibility of a user to implement a primitive which checks if the variable is set to true. Of course there is no way we can enforce it on the user, and certainly there is no reason why we should put this burden on him or her. Therefore, a workaround was applied. When a user specifies a walltime for a Java task, the scheduler enforces the creation of a Forked Java Task instead. Then the Java task is executed in a new JVM (with default options and copied classpath) and it is possible to easily terminate the task in case the walltime is exceeded.

To start a walltime mechanism user specifies it in a task descriptor. Then, directly before the execution of a task the walltime primitive is started. If a task finishes before the walltime, it is sufficient to cancel the walltime primitive. However, if the walltime mechanism has to stop the execution it will force the executable to finish immediately with an exception. To start a walltime mechanism and (if applicable) cancel it, the Task Launcher is used. As described in previous sections, Task Launcher is responsible for performing preparations, calling an Executable, and performing cleaning operations. The code is as follows:

```
 TaskResult doTask (Executable executable) throws WalltimeExceededException {

        // step 1: preparations
        performPreExecutionScripts();
        if (isWallTime) startWalltimeMechanism( executable );

        // step 2: execution - this method might throw a WalltimeExceededException
        TaskResult result = executable.execute();

        // step 3: cleaning
        finalizeTask();
        if (isWallTime) cancelWalltimeMechanism();

        // step 4: return a result
        return result
}
```

The walltime mechanism is implemented with a use of a Timer and TimerTask classes from standard Java library. It works as follows. Java offers a possibility to schedule an object for an execution after the specified delay. The scheduled object must extend TimerTask class and have a run method overridden (which will be invoked). The delay, in our case, is simply a walltime. The class which extends a TimerTask class is called a KillProcess. To start a timer and create a KillProcess a KillTask object is used.

KillTask object is created in a simple way. All it needs is a reference to an Executable for which a walltime was specified and a walltime itself:

```
public KillTask(Executable executable, long walltime) {
      this.executable = executable;
      this.walltime   = walltime;
}
```

Once the KillTask object is created it is enough to invoke a schedule method on this object to create and start a timer:

```
public void schedule() {
      timer = new Timer();
      timer.schedule(new KillProcess(), walltime);
}
```

Subsequently, in the startWalltimeMechanism primitive (in a doTask method in Task Launcher class) we will create a KillTask object and invoke a specified above schedule method:

```
void startWalltimeMechanism(Executable executable) {
      killTask = new KillTask(executable, wallTime);
      killTask.schedule();
}
```

Complementary to this, there is a cancelling function (in Task Launcher class) :

```
void cancelWalltimeMechanism() {
       killTask.cancel();
}
```

which invokes (in KillTask class):

```
public void cancel() {
       timer.cancel();
}
```

An object which extends TimerTask class and which method is called when a walltime expires is as simple as that:

```
class KillProcess extends TimerTask {
        public void run() {
                executable.kill();
        }
}
```

When killing an executable we have to be more careful. It is possible to obtain an interlace where an Executable finished normally and the walltime mechanism is just about to kill it. To prevent any errors in this situation, the killing method must be implemented reasonably. We have to guarantee that when a task is complete, starting a kill method in a walltime mechanism will not end up in any exceptions or undesirable behavior.

For example a kill method implemented in a NativeExecutable firstly checks whether the process is finished (by checking exit status) and in case it is not, it destroys it:

```
public void kill() {
      if (process != null) {
            try {
                  // exitValue throws an exception if process is still running
                  process.exitValue();
            } catch (IllegalThreadStateException e) {
                  process.destroy();
            }
       }
}
```

Also, the kill method in Forked Java Executable is safe. It sets attribute killed to true, but as the task is already finished it will not react on the change of this variable:

```
public void kill() {
      // killing an (unfinished) executable requires additional handling
      // described further on
      this.killed = true;
}
```

This way, we can guarantee that if unexpected interlaces take place they will not affect the execution of the task.

Walltime in Forked Java Executable

As mentioned in the introductory part the implementation of a walltime mechanism for a Forked Java Task is a bit tricky. When a walltime is specified for a Forked Java Task, then the Forked Java Executable must provide additional handling for this mechanism. Basically, the executable delegates the execution of a task to a remote active object TaskLauncher, and receives instantly a future object as a result:

```
TaskResult result = taskLauncher.doTask(executable);
```

Basically, a future object is an immediately returned placeholder for an object which is later updated with a real value. After receiving a result future object, Forked Java Executable waits for an update, however in the meantime it might be terminated by a walltime mechanism (implemented in a KillProcess class). The walltime mechanism for a Forked Java Executable is implemented only by setting a killed attribute to true. Therefore, it cannot just wait for an update:

```
PAFuture.waitFor(result);
```

but it must provide handling for a walltime mechanism as well (checking the status of a killed attribute).

To be able to react on both events the below loop is implemented. It checks whether attribute killed was set to true and if not, it waits for an update of a future object (but waits with a timeout). It loops until it is notified by either event:

```
while (!isKilledByWalltimeMechanism()) {
         try {
                  // the following method throws an exception if timeout expires,
                  // if no exception is thrown then the below break will exit
                  // the loop
                  PAFuture.waitFor(result, TIMEOUT);
                  break;
         } catch (ProActiveTimeoutException e) {
         // if we get here, then the future has not yet been updated,
         // loop again
         }
}
```

We could exit the above loop because of a walltime mechanism or a normal completion of a task. In case we exited because of a walltime mechanism we need to remove an internal monitoring primitive for future objects and exit a function returning an exception. Internal monitoring primitive for future objects is provided by ProActive library and allows to notify about an update of a future object. As the execution of the task was stopped, this monitoring must be removed.

```
if (isKilled()) {
           FutureMonitoring.removeFuture(
                            ((FutureProxy) ((StubObject) result).getProxy()));
           throw new SchedulerException("Walltime exceeded");
}
```

Of course, when a task is finished without an interference of a walltime mechanism, a result of a remote execution is returned and passed to the user.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

My contributions involved work on the ProActive Scheduler. The ProActive Scheduler is composed of two components: a job management component and a resource management component. For the job management component I developed a Task Walltime mechanism as shown in section 5.4. Task Walltime is a maximum allowed execution time of any task (native, java or ProActive task). When a task does not finish before its walltime it is terminated by the Scheduler. Task Walltime for the Java tasks needed special handling as stopping threads in Java is deprecated. The second contribution to the job management component of the Scheduler was a new type of a task: Forked Java Task (section 5.3). The Forked Java Task is a Java task executed in a dedicated JVM. It allows for more flexible execution environment as we can specify JVM providers or use different Java options. In this implementation a forked JVM registers itself in the parent JVM and then executes a task. After the execution the JVM process is terminated. With respect to the resource management component I integrated Microsoft Compute Cluster Server into the ProActive deployment framework. Deployment framework is used by the ProActive Scheduler to allocate resources. The integration (Chapter 4) resembles gliding-in mechanism in Condor. It uses long-term reservations of resources and short-term job scheduling on the allocated resources. In the end of my internship at INRIA all my implementations were incorporated into the trunk versions of the source code. Also upon successful completion of integrating CCS into the ProActive Scheduler the Microsoft renewed the collaboration with the OASIS team.

6.2. Future Work

The ProActive Scheduler is composed of two major components: a component responsible for job management (scheduling, executing) and a component responsible for resource management (acquisition, monitoring). My contributions involved work on both components, and each of the contribution can be further developed. For the resource management layer I added support for Microsoft Compute Cluster Server and new development is mainly related to the features provided with a new version of Microsoft CCS: Microsoft High Performance Computing Server 2008. As to the job management layer, the Task Walltime mechanism can be subsequently used in sophisticated scheduling polices. The detailed further development of the contributions is discussed below.

6.2.1. Support for Microsoft CCS

At present, the Microsoft cluster at INRIA runs a Windows Server 2003 Compute Cluster Edition. There is a plan to install the Windows High-Performance Computing Server 2008 ([HPC]). Among many interesting features like high availability (through multiple head nodes), Windows PowerShell and new scheduling policies, there is one feature for which the support could be instantly added. This is job scheduling granularity at processor core, processor socket, and compute node levels. By now, users specify how many processing units they need and receive a collection of CPUs in response. However when a task is multithreaded, then the allocation a whole compute node could be more efficient (depending on number of threads). As good example is as follows. During the deployment on CCS one JVM is started for each allocated processor. Then on each JVM a ProActive Node is created and hosts active objects. Active objects located on different ProActive Nodes might communicate with each other. Even if the communication takes place within the same host it still involves communication between many virtual machines (up to the number of processors). It could be optimized by starting only one JVM and creating all ProActive Nodes on that JVM. Then, any communication between active objects hosted by any ProActive Node on the same machine is local (carried out within the same virtual machine). This can be achieved with a new version Windows HPC 2008.

6.2.2. Scheduling policies

Jobs submitted to any scheduler are served in an order according to a scheduling policy. Some schedulers provide a few policies ranging from the simplest FIFO policy to more sophisticated ones like backfilling (described in section 3.1.6). By now the ProActive Scheduler supports a FIFO policy. The goal for the future would be an implementation of a backfilling policy, and providing an option for a user to choose which policy to use. The module Task Walltime which I implemented in the Scheduler is the first step in the backfilling implementation. Second step would be an implementation of a reservation system which would be in charge of computing the earliest availability of the resources to start the job and placing job executions as reservations. The last step would be an implementation of a process that would be finding jobs in the queue that can be executed as backfills (executed before intially planned) taking advantage of the unused resources.

Bibliography

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BBRZ05] L. Baduel, F. Baude, N. Ranaldo, and E. Zimeo. Effective and Efficient Communication in Grid Computing with an Extension of ProActive Groups. In *JPDC, 7th International Worshop on Java for Parallel and Distributed Computing at IPDPS*, Denver, Colorado, USA, apr 2005.

[BCM+02] Françoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[BLE] Blender: cross platform suite of tools for 3d creation. `http://www.blender.org/`.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[CHS03] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous sequential processes. Technical report, INRIA Sophia Antipolis, 2003. RR-4753.

[CLI] *Compute Cluster Server Command Line Interface Reference.* `http://technet2.microsoft.com/windowsserver/en/library/e544d9e2-e1db-447c-8a31-6bd0c30d809c1033.mspx?mfr=true`.

[CQ03] Denis Caromel and Romain Quilici. Proactive: From active objects to components, towards pse. In *EURESCO Conference on Advanced Environments and Tools for High Performance Computing*, EuroConference on Problem Solving Environments and the Information Society, Albufeira, Portugal, June 2003. http://www.esf.org/euresco/03/pc03139.

[DGF95] Larry Rudolph Dror G. Feitelson. *Job Scheduling Strategies for Parallel Processing*, chapter Job scheduling under the Portable Batch System. Springer, April 1995.

[DPR] *ProActive Library download.* `http://proactive.inria.fr/index.php?page=download_proactive_latest`.

[DSC] *ProActive Scheduler Download.* `http://proactive.inria.fr/index.php?page=download_scheduler_latest`.

[HJC99] K.A. Hawick H.A. James and P.D. Coddington. Scheduling independent tasks on metacomputing systems. Parallel and Distributed Computing Systems, Fort Lauderdale, Florida, USA, 1999.

[HPC] *Windows HPC Server 2008 Technical Overview.* `http://download.microsoft.com/download/2/d/4/2d471958-1971-4183-933c-0ed1888a9093/Windows%20HPC%20Server%202008%20Technical%20Overview.doc`.

[PBS] *Windows HPC Server 2008 Technical Overview.* `http://www.pbsgridworks.com/PBSTemp1.3.aspx?top_nav_name=Products&item_name=OpenPBS&top_nav_str=1&AspxAutoDetectCookieSupport=1`.

[RAN] *R&D centres ranking.* `http://www.webometrics.info/top1000_r&d.asp`.

[SCH] *Windows Compute Cluster Server 2003 Job Scheduler.* `http://technet2.microsoft.com/windowsserver/en/library/4fb21c96-759c-493a-bb29-f14bd491160d1033.mspx?mfr=true`.

[Sch02] Jennifer M. Schopf. *A General Architecture for Scheduling on the Grid.* 2002. `http://www.mcs.anl.gov/~schopf/Pubs/sched.arch.2002.pdf`.

[SCR] *Scripting for Windows HPC.* `http://www.microsoft.com/technet/scriptcenter/hubs/ccs.mspx`.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.