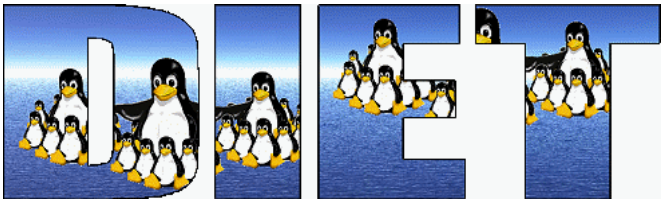

USER'S MANUAL

VERSION	2.6
DATE	April 2010
PROJECT MANAGER	Frédéric Desprez.
EDITORIAL STAFF	Yves Caniou, Eddy Caron and David Loureiro.
AUTHORS STAFF	Abdelkader Amar, Raphaël Bolze, Éric Boix, Yves Caniou, Eddy Caron, Pushpinder Kaur Chouhan, Philippe Combes, Sylvain Dahan, Holly Dail, Bruno Delfabro, Benjamin Depardon, Peter Frauenkron, Georg Hoesch, Benjamin Isnard, Mathieu Jan, Jean-Yves L'Excellent, Gal Le Mahec, Christophe Pera, Cyrille Pontvieux, Alan Su, Cédric Tedeschi, and Antoine Vernois.

Contents

Introduction

Resource management is one of the key issues for the development of efficient Grid environments. Several approaches co-exist in today's middleware platforms. The granularity of computation (or communication) and dependencies between computations can have a great influence on the software choices.

The first approach provides the user with a uniform view of resources. This is the case of GLOBUS [?] which provides transparent MPI communications (with MPICH-G2) between distant nodes but does not manage load balancing issues between these nodes. It's the user's task to develop a code that will take into account the heterogeneity of the target architecture. Grid extensions to classical batch processing provide an alternative approach with projects like Condor-G [?] or Sun GridEngine [?]. Finally, peer-to-peer [?] or Global computing [?] can be used for fine grain and loosely coupled applications.

A second approach provides a semi-transparent access to computing servers by submitting jobs to servers offering specific computational services. This model is known as the Application Service Provider (ASP) model where providers offer, not necessarily for free, computing resources (hardware and software) to clients in the same way as Internet providers offer network resources to clients. The programming granularity of this model is rather coarse. One of the advantages of this approach is that end users do not need to be experts in parallel programming to benefit from high performance parallel programs and computers. This model is closely related to the classical Remote Procedure Call (RPC) paradigm. On a Grid platform, RPC (or GridRPC [?, ?]) offers easy access to available resources from a Web browser, a Problem Solving Environment (PSE), or a simple client program written in C, Fortran, or Java. It also provides more transparency by hiding the selection and allocation of computing resources. We favor this second approach.

In a Grid context, this approach requires the implementation of middleware to facilitate client access to remote resources. In the ASP approach, a common way for clients to ask for resources to solve their problem is to submit a request to the middleware. The middleware will find the most appropriate server that will solve the problem on behalf of the client using a specific software. Several environments, usually called Network Enabled Servers (NES), have developed such a paradigm: NetSolve [?], Ninf [?], NEOS [?], OmniRPC [?], and more recently DIET developed in the GRAAL project. A common feature of these environments is that they are built on top of five components: clients, servers, databases, monitors and schedulers. Clients solve computational requests on servers found by the NES. The NES schedules the requests on the different servers using performance information obtained by monitors and stored in a database.

DIET stands for Distributed Interactive Engineering Toolbox. It is a toolbox for easily developing Application Service Provider systems on Grid platforms, based on the Client/Agent/Server scheme. Agents are the schedulers of this toolbox. In DIET, user requests are served

via RPC.

DIET follows the GridRPC API defined within the Open Grid Forum [**?**].

Chapter 1

Deploying a DIET platform

Deployment is the process of launching a DIET platform including agents and servers. For DIET, this process includes writing configuration files for each element and launching the elements in the correct hierarchical order. There are three primary ways to deploy DIET.

Launching **by hand** is a reasonable way to deploy DIET for small-scale testing and verification. This chapter explains the necessary services, how to write DIET configuration files, and in what order DIET elements should be launched. See Section 1.1 for details.

GoDIET is a Java-based tool for automatic DIET deployment that manages configuration file creation, staging of files, launch of elements, monitoring and reporting on launch success, and process cleanup when the DIET deployment is no longer needed. See Section 1.2 for details.

Writing your own scripts is a surprisingly popular approach. This approach often looks easy initially, but can sometimes take much, much longer than you predict as there are many complexities to manage. Learn GoDIET– it will save you time!

1.1 Deployment basics

1.2 GoDIET

GoDIET is an cross-platform tool that helps you automate ad-hoc deployment and management procedures for DIET infrastructure. It manages configuration file creation, staging of files, launch of elements, monitoring and reporting. GoDIET is extremely useful for large deployments on a complex physical infrastructure. The mains features are:

- Complete command line interface.
- Distributed command execution via SSH.
- Real time monitoring applications state.
- Complex physical infrastructure management with firewall and multiple network lan.

1.2.1 Quickstart

For hurry people.

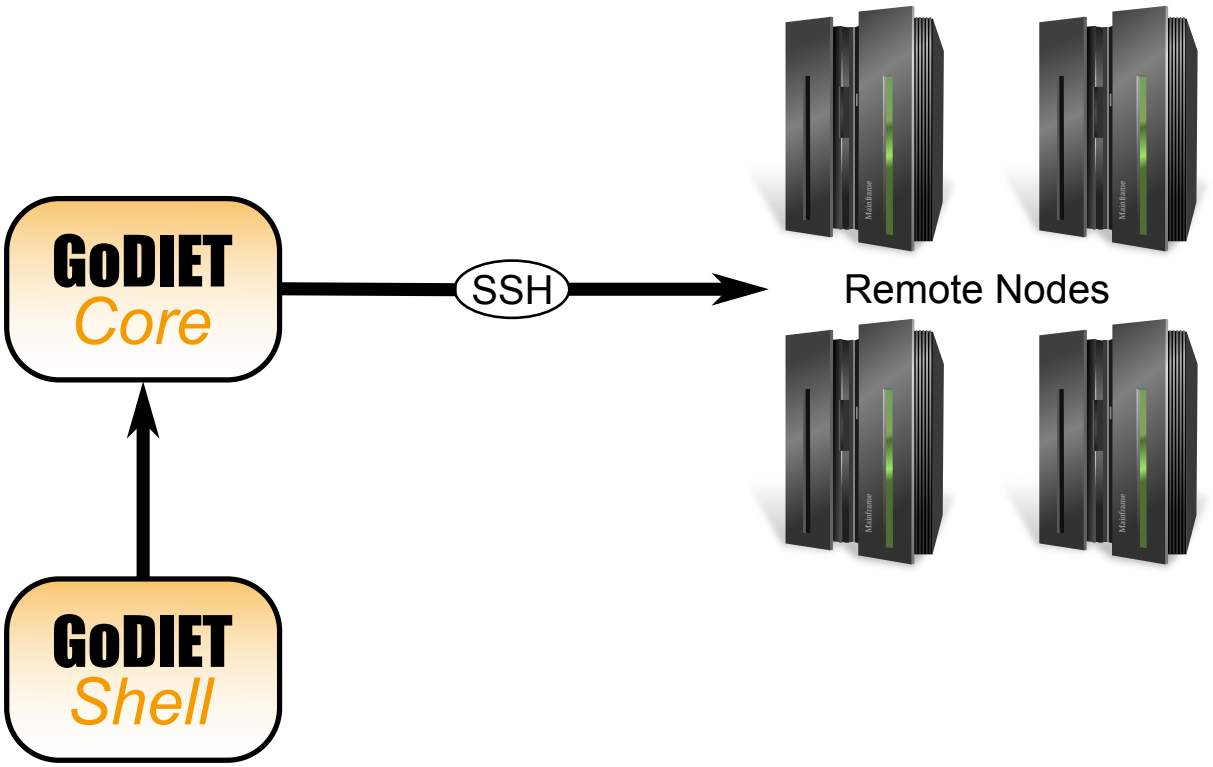

Figure 1.1: Design principle of GoDiet.

Installing GoDiet

The following operating systems are known to support GoDiet:

- Linux: Most recent distributions are likely to work
- Mac OS X 10.4 or later

You need to have the Sun Java 6 or openJDK6 installed. All operating system with Java 6 must support First download GoDiet on the project website[1].

Extract the archive just and launch run.bat or run.sh script. You need to load your physical platform on which will GoDiet will running. This platform must be describe in a XML file based on *Platform.xsd grammar. You can find an simple file in the example directory*

The user of GoDiet *could describes the desired deployment in an XML file including all needed external services (*e.g., *omniNames and LogService); the desired hierarchical organization of agents and servers is expressed directly using the hierarchical organization of XML. The user also defines all machines available for the deployment, disk scratch space available at each site for storage of configuration files, and which machines share the same disk to avoid unecessary copies.*

An example input XML file is shown in Figure 1.2; see [?] for a full explanation of all entries in the XML. You can also have a look at the fully commented XML example file provided in the GoDiet *distribution under examples/commented.xml, each option is explained. To*

[1] http://graal.ens-lyon.fr/DIET/godiet.html

launch GoDiet *for the simple example XML file provided in the* GoDiet *distribution under examples/example1.xml, run:*

```
~ > java -jar GoDIET-x.x.x.jar example1.xml
XmlScanner constructor
Parsing xml file: example1.xml
GoDIET>
```

GoDiet *reads the XML file and then enters an interactive console mode. In this mode you have a number of options:*

We will now launch this example; note that this example is intentionally very simple with all components running locally to provide initial familiarity with the GoDiet *run procedure. Deployment with* GoDiet *is especially useful when launching components on multiple remote machines.*

```
GoDIET> launch
* Launching DIET platform at Wed Jul 13 09:57:03 CEST 2005

Local scratch directory ready:
        /home/hdail/tmp/scratch_godiet

** Launching element OmniNames on localHost
Writing config file omniORB4.cfg
Staging file omniORB4.cfg to localDisk
Executing element OmniNames on resource localHost
Waiting for 3 seconds after service launch

** Launching element MA_0 on localHost
Writing config file MA_0.cfg
Staging file MA_0.cfg to localDisk
Executing element MA_0 on resource localHost
Waiting for 2 seconds after launch without log service feedback

** Launching element LA_0 on localHost
Writing config file LA_0.cfg
Staging file LA_0.cfg to localDisk
Executing element LA_0 on resource localHost
Waiting for 2 seconds after launch without log service feedback

** Launching element SeD_0 on localHost
Writing config file SeD_0.cfg
Staging file SeD_0.cfg to localDisk
Executing element SeD_0 on resource localHost
Waiting for 2 seconds after launch without log service feedback
* DIET launch done at Wed Jul 13 09:57:14 CEST 2005 [time= 11.0 sec]
```

The `status` *command will print out the run-time status of all launched components. The* `LaunchState` *reports whether* GoDiet *observed any errors during the launch itself. When*

the user requests the launch of LogService in the input XML file, GoDiet *can connect to the LogService after launching it to obtain the state of launched components; when available, this state is reported in the* ***LogState*** *column.*

```
GoDIET> status
Status   Element   LaunchState   LogState   Resource     PID
         OmniNames running       none       localHost    1232
         MA_0      running       none       localHost    1262
         LA_0      running       none       localHost    1296
         SeD_0     running       none       localHost    1329
```

Finally, when you are done with your Diet *deployment you should always run* ***stop****. To clean-up each element,* GoDiet *runs a* ***kill*** *operation on the appropriate host using the stored PID of that element.*

```
GoDIET> stop

* Stopping DIET platform at Wed Jul 13 10:05:42 CEST 2005
Trying to stop element SeD_0
Trying to stop element LA_0
Trying to stop element MA_0
Trying to stop element OmniNames

* DIET platform stopped at Wed Jul 13 10:05:43 CEST 2005[time= 0.0 sec]
* Exiting GoDIET. Bye.
```

One of the main problem when writing a GoDiet *XML input file is to be compliant with the dtd. A good tool to validate a* GoDiet *file before using* GoDiet *is* ***xmllint****. This tool exist on most platforms and with the following command:*

```
$ xmllint your_xml_file --dtdvalid path_to_GoDIET.dtd -noout
```

you will see the different lines where there is problem and a clear description of why your XML file is not compliant.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE diet_configuration SYSTEM "../GoDIET.dtd">
<diet_configuration>
  <goDiet debug="2" saveStdOut="yes" saveStdErr="yes" useUniqueDirs="no" log="no"/>
  <resources>
    <scratch dir="/tmp/GoDIET_scratch"/>
    <storage label="disk-1">
        <scratch dir="/tmp/run_scratch"/>
        <scp server="res1" login="doe"/>
    </storage>
    <storage label="disk-2">
        <scratch dir="/tmp/run_scratch"/>
        <scp server="res2" login="foo"/>
    </storage>
    <storage label="disk-3">
        <scratch dir="/tmp/run_scratch"/>
        <scp server="res3" login="bar"/>
    </storage>
    <compute label="res1" disk="disk-1">
        <ssh server="res1" login="doe"/>
        <env>
              <var name="PATH" value=""/>
              <var name="LD_LIBRARY_PATH" value=""/>
        </env>
    </compute>
    <compute label="res2" disk="disk-2">
        <ssh server="res2" login="foo"/>
              <env>
                      <var name="PATH" value=""/>
                      <var name="LD_LIBRARY_PATH" value=""/>
              </env>
    </compute>
        <cluster label="res3" disk="disk-3" login="bar">
      <env>
        <var name="PATH" value=""/>
        <var name="LD_LIBRARY_PATH" value=""/>
      </env>
      <node label="res3_host1">
        <ssh server="host1.res3.fr"/>
        <end_point contact="192.5.80.103"/>
      </node>
      <node label="res3_host2">
        <ssh server="host2.res3.fr"/>
      </node>
    </cluster>
  </resources>
  <diet_services>
          <omni_names contact="res1_IP" port="2121">
        <config server="res1" remote_binary="omniNames"/>
    </omni_names>
  </diet_services>
  <diet_hierarchy>
    <master_agent label="MA">
        <config server="res1" remote_binary="dietAgent"/>
           <cfg_options>
             <option name="traceLevel" value="1"/>
           </cfg_options>
            <SeD label="SeD1">
                <config server="res2" remote_binary="server_dyn_add_rem"/>
           <cfg_options>
             <option name="traceLevel" value="1"/>
           </cfg_options>
            </SeD>
        <SeD label="SeD2">
            <config server="res3_host1" remote_binary="server_dyn_add_rem"/>
           <cfg_options>
             <option name="traceLevel" value="30"/>
           </cfg_options>
           <parameters string="T"/>
        </SeD>
        <SeD label="SeD3">
            <config server="res3_host2" remote_binary="server_dyn_add_rem"/>
           <cfg_options>
             <option name="traceLevel" value="1"/>
           </cfg_options>
        </SeD>
    </master_agent>
  </diet_hierarchy>
</diet_configuration>
```

.

Figure 1.2: Example XML input file for GoDiet.

1.3 Shape of the hierarchy

A recurrent question people ask when first using DIET*, is* how should my hierarchy look like? *There is unfortunately no universal answer to this. The shape highly depends on the performances you want* DIET *to attain. The performance metric that we often use to characterize the performance of* DIET *is the* throughput *the clients get,* i.e., *the number of serviced requests per time unit.*

Several heuristics have been proposed to determine the shape of the hierarchy based on the users' requirements. We can distinguish two main studies. The first one focused on deploying a single service in a DIET *hierarchy [***?***,* **?***]. The shape of the best hierarchy on a fully homogeneous platform is a* Complete Spanning d-ary tree *(CSD tree). The second study focused on deploying several services alongside in a single hierarchy. Heuristics based on linear programming and genetic algorithm have been proposed for different kinds of platform [***?***,* **?***].*

Even though the above mentioned studies can provide good, if not the best, deployments, they heavily rely on modelizations and benchmarks of DIET *and the services. This process can be quite long. Thus, we propose in this section a simple but somehow efficient way of deploying a hopefully "good" hierarchy. (Note that if you do not care about performances, a simple star graph should be enough,* i.e., *an MA and all your SeDs directly connected to it.) Here are a few general remarks on* DIET *hierarchies:*

- *The more computations your service needs, the more powerful server you should choose to put a SeD on.*
- *Scheduling performances of agents depends on the number of children they have, as well as on the number of services each of the child knows. An agent will be more efficient when all its children know one and the same service. Thus it is a good idea to group all SeDs having the same characteristics under a common agent.*
- *The services declared by an agent to its parent is the union of all services present in its underlying hierarchy.*
- *It is usually a good idea to group all* DIET *elements having the same services and present on a given cluster under a common agent.*
- *There is usually no need for too many levels of agents.*
- *If you suspect that an agent does not schedule the requests fast enough, and that it is overloaded, then add two (or more) agents under this agent, and divide the previous children between these new agents.*