

DIET Installation

DIET Team

October 19, 2010

Chapter 1

DIET Installation

1.1 Dependencies

1.1.1 General remarks on DIET platform dependencies

DIET is itself written in C/C++. DIET is based on CORBA and thus depends on the chosen CORBA implementation. Additionally, some of DIET extensions make a strong use of libraries themselves written in C/C++. Thus, we could expect DIET to be effective on any platform offering decent version of such compilers.

DIET undergoes daily regression tests (see <http://graal.ens-lyon.fr/DietDashboard>) on various hardwares, a couple of Un*x based operating systems (under different distributions), MacOSX and AIX, and mainly with GCC. But, thanks to users reports (punctual deployments and special tests conducted before every release), DIET is known to be effective on a range of platforms.

Nevertheless, if you encounter installation difficulties don't hesitate to post on DIET's users mailing list: `diet-usr@listes.ens-lyon.fr` (for the archives refer to <http://graal.ens-lyon.fr/DIET/mail-lists.html>). If you find a bug in DIET, please don't hesitate to submit a bug report on <http://graal.ens-lyon.fr/bugzilla>. If you have multiple bugs to report, please make multiple submissions, rather than submitting multiple bugs in a single report.

1.1.2 Hardware dependencies

DIET is fully tested on Linux/i386 and Linux/i686 platforms. DIET is known to be effective on Linux/Sparc, Linux/i64, Linux/amd64, Linux/Alpha, Linux/PowerPC, AIX/PowerPC and MacOS/PowerPC platforms. At some point in DIET history, DIET used to be tested on the Solaris/Sparc platform...

1.1.3 Supported compilers

DIET is supported on GCC with versions ranging from 3.2.X to 4.1.x. Note that due to omniORB 4 (see [1.2.1](#)) requirements towards thread-safe management of exception

handling, compiling DIET with `gcc` requires at least `gcc-2.96`. DIET is also supported on XL compiler(IBM) and Intel compiler.

1.1.4 Operating system dependencies

DIET is fully tested on Linux [with varying distributions like Debian, Red Hat Enterprise Linux (REL-ES-3), Fedora Core (5)], on AIX (5.3) and on MacOSX (Darwin 8).

1.2 Communications layer

Middleware environments can be implemented using a classic socket communication layer. Several problems to this approach have been pointed out such as the lack of portability or limits on the number of sockets that can be opened concurrently. Our aim is to implement and deploy a distributed NES environment that works at a wider scale. Distributed object environments, such as *Java*, *DCOM* or *CORBA* have proven to be a good base for building applications that manage access to distributed services. They not only provide transparent communications in heterogeneous networks, but they also offer a framework for the large scale deployment of distributed applications. Being open and language independent, *CORBA* was chosen as the communication layer in DIET.

As recent implementations of *CORBA* provide communication times close to that of sockets, *CORBA* is well suited to support distributed applications in a large scale Grid environment. New specialized services can be easily published and existing services can also be used. DIET is based upon *OmniORB 3/4* or later, a free *CORBA* implementation that provides good communication performance.

1.2.1 Software dependencies

As explained in Section 1.2, *CORBA* is used for all communications inside the platform. The implementations of *CORBA* currently supported in DIET is **omniORB 4** which itself depends on **Python**.

NB: We have noticed that some problems occur with **Python 2.3**: the C++ code generated by `idl` could not be compiled. It has been patched in DIET, but some warnings may still appear.

omniORB 4 itself also depends on **OpenSSL** in case you wish to secure your DIET platform. If you want to deploy a secure DIET platform, SSL support is not yet implemented in DIET, but an easy way to do so is to deploy DIET over a VPN.

In order to deploy *CORBA* services with *omniORB*, a configuration file and a log directory are required: see Section 2.1.1 for a complete description of the services. Their paths can be given to *omniORB* either at runtime (through the well-known environment variables `$OMNIORB_CONFIG` and `$OMNINAMES_LOGDIR`), and/or at *omniORB* compile time (with the `--with-omniORB-config` and `--with-omniNames-logdir` options.)

1.3 Compiling the platform

DIET compilation process moved away from the traditional `autotools` way of things to a tool named `cmake` (mainly to benefit from `cmake`'s built-in regression tests mechanism).

Before compiling DIET itself, first install the above mentioned (cf Section 1.2.1) dependencies. Then untar the DIET archive and change current directory to its root directory.

1.3.1 Obtaining and installing `cmake` per se

DIET requires using `cmake` at least version 2.4.3. For many popular distributions `cmake` is incorporated by default or at least `apt-get` (or whatever your distro package installer might be) is `cmake` aware. Still, in case you need to install an up-to-date version `cmake`'s official site distributes many binary versions (alas packaged as tarballs) which are made available at <http://www.cmake.org/HTML/Download.html>. Optionally, you can download the sources and recompile them: this simple process (`./bootstrap; make; make install`) is described at <http://www.cmake.org/HTML/Install.html>.

1.3.2 Configuring DIET's compilation: `cmake` quick introduction

If you are already experienced with `cmake` then using it to compile DIET should provide no surprise. DIET respects `cmake`'s best practices e.g. by clearly separating the source tree from the binary tree (or compile tree), by exposing the main configuration optional flag variables prefixed with `DIET_` (and by hiding away the technical variables) and by not postponing configuration difficulties (in particular the handling of external dependencies like libraries) to compile stage.

`Cmake` classically provides two ways for setting configuration parameters in order to generate the makefiles in the form of two commands `ccmake` and `cmake` (the first one has an extra "c" character):

```
ccmake [options] <path-to-source>
```

in order to specify the parameters interactively through a GUI interface

```
cmake [options] <path-to-source> [-D<var>:<type>=<value>]
```

in order to define the parameters with the `-D` flag directly from the command line.

In the above syntax description of both commands, `<path-to-source>` specifies a path to the top level of the source tree (i.e. the directory where the top level `CMakeLists.txt` file is to be encountered). Also the current working directory will be used as the root of the build tree for the project (out of source building is generally encouraged especially when working on a CVS tree).

Here is a short list of `cmake` internal parameters that are worth mentioning:

- `CMAKE_BUILD_TYPE` controls the type of build mode among which `Debug` will produce binaries and libraries with the debugging information
- `CMAKE_VERBOSE_MAKEFILE` is a Boolean parameter which when set to `ON` will generate makefiles without the `.SILENT` directive. This is useful for watching the invoked commands and their arguments in case things go wrong.
- `CMAKE_C[XX]_FLAGS*` is a family of parameters used for the setting and the customization of various C/C++ compiler options.
- `CMAKE_INSTALL_PREFIX` variable defines the location of the install directory (defaulted to `/usr/local` on `un*x`). This is `cmake`'s portable equivalent of the `auto-tools` `configure`'s `--prefix=` option.

Eventually, here is a short list of `ccmake` interface tips:

- when lost, look at the bottom lines of the interface which always summarizes `ccmake`'s most pertinent options (corresponding keyboard shortcuts) depending on your current context
- hitting the `"h"` key will direct you `ccmake` embedded tutorial and a list of keyboard shortcuts (as mentioned in the bottom lines, hit `"e"` to exit)
- up/down navigation among parameter items can be achieved with the up/down arrows
- when on a parameter item, the line in inverted colors (close above the bottom of the screen) contains a short description of the selected parameter as well as the set of possible/recommended values
- toggling of boolean parameters is made with `enter`
- press `enter` to edit path variables
- when editing a `PATH` typed parameter the `TAB` keyboard shortcut provides an emacs-like (or bash-like) automatic path completion.
- toggling of advanced mode (press `"t"`) reveals hidden parameters

1.3.3 A `ccmake` walk-through for the impatient

Assume that `SVN_DIET_HOME` represents a path to the top level directory of DIET sources. Additionally, assume we created a build tree directory and `cd` to it (in the example below we chose `SVN_DIET_HOME/Bin` as build tree, but feel free to follow your conventions):

- `cd SVN_DIET_HOME/Bin`
- `ccmake ..` to enter the GUI

- press `c` (equivalent of `bootstrap.sh` of the autotools)
 - specify the `CMAKE_INSTALL_PREFIX` parameter (if you wish to install in a directory different from `/usr/local`)
 - press `c` again, for checking required dependencies
 - check all the parameters preceded with the `*` (star) character whose value was automatically retrieved by `cmake`.
 - provide the required information i.e. fill in the proper values for all parameters whose value is terminated by `NOT-FOUND`
 - iterate the above process of parameter checking, toggle/specification and configuration until all configuration information is satisfied
 - press `g` to generate the makefile
 - press `q` to exit `ccmake`
- `make` in order to classically launch the compilation process
 - `make install` when installation is required

1.3.4 DIET's main configuration flags

Here are the main configuration flags:

- `OMNIORB4_DIR` is the path to the omniORB4 installation directory (only relevant when omniORB4 was not installed in `/usr/local`).
Example: `cmake .. -DOMNIORB4_DIR:PATH=$HOME/local/omniORB-4.0.7`
- `DIET_BUILD_LIBRARIES` which is enabled by default, activates the compilation of the DIET libraries. Disabling this option is only useful if you wish to restrict the compilation to the construction of the documentation.

1.3.5 DIET's advanced configuration flags

Eventually, some configuration flags control the general result of the compilation or some developers extensions:

- `BUILD_SHARED_LIBS` is a `cmake` internal variable which specifies whether the libraries should be dynamics as opposed to static (on Mac system this option is automatically set to `ON`, as static compilation of binaries seems to be forbidden on these systems)

1.3.6 Compiling and installing

Summarizing the configuration choices

Once the configuration is properly made one can check the choices made by looking the little summary proposed by cmake. This summary should look like ([...] denotes eluded portions):

```
~/DIET > ./cmake ..
[...]
- Install prefix: /home/diet/local/diet
- OmniORB found: YES
  * OmniORB directory: /home/diet/local/omniORB-4.0.7
  * OmniORB includes: /home/diet/local/omniORB-4.0.7/include
  * OmniORB libraries: /home/diet/local/omniORB-4.0.7/lib/libomniDynamic4.so;
    [...]libomniORB4.so; [...]libomnithread.so; [...]libCOS4.so; [...]
- General options:
  * Dynamics Libraries: ON
- Options set:
[...]
```

A more complete, yet technical, way of making sure is to check the content of the file named `CMakeCache.txt` (generated by cmake in the directory from which cmake was invoked). When exchanging with the developers list it is a recommendable practice to join the content of this file which summarizes your options and also the automatic package/library detections made by cmake.

Compiling stage

You are now done with the configuration stage (equivalent of both the `bootstrap.sh` and `./configure` stage of the `autotools`). You are now back to your platform level development tools i.e. `make` when working on Unices. Hence you can now proceed with the compiling process by launching `make`.

Installation stage

After compiling (linking, and testing) you can optionally proceed with the installation stage with the `make install` command.

Chapter 2

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.

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 2.1 for details.

2.1 Deployment basics

2.1.1 Using CORBA

CORBA is used for all communications in DIET and for communications between DIET and accessory services such as LogService, VizDIET, and GoDIET. This section gives basic information on how to use DIET with CORBA. Please refer to the documentation of your ORB if you need more details.

The naming service

DIET uses a standard CORBA naming service for translating an user-friendly string-based name for an object into an Interoperable Object Reference (IOR) that is a globally unique identifier incorporating the host and port where the object can be contacted. The naming service in omniORB is called **omniNames** and it must be launched before any other DIET entities. DIET entities can then locate each other using only a string-based name and the <host:port> of the name server.

To launch the omniORB name server, first check that the path of the omniORB libraries is in your environment variable `LD_LIBRARY_PATH`, then specify the log directory, through the environment variable `OMNINAMES_LOGDIR` (or, with **omniORB 4**, at compile time, through the `--with-omniNames-logdir` option of the omniORB configure script). If there are no log files in this directory, **omniNames** needs to be initialized. It can be launched as follows:


```
~ > omniNames -start
```

```
Tue Jun 28 15:56:50 2005:
```

```
Starting omniNames for the first time.
```

```
Wrote initial log file.
```

```
Read log file successfully.
```

```
Root context is IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f4e61
6d696e67436f6e746578744578743a312e30000001000000000000060000000010102000d0000003134
302e37372e31332e34390000f90a0b0000004e616d6553657276696365000200000000000008000000
0100000000545441010000001c0000000100000001000100010000000100010509010100010000000901
0100
```

```
Checkpointing Phase 1: Prepare.
```

```
Checkpointing Phase 2: Commit.
```

```
Checkpointing completed.
```

This sets an omniORB name server which listens for client connections on the default port 2809. If omniNames has already been launched once, *ie* there are already some log files in the log directory, using the `-start` option causes an error. The port is actually read from old log files:

```
~ > omniNames -start
```

```
Tue Jun 28 15:57:39 2005:
```

```
Error: log file '/tmp/omninames-toto.log' exists. Can't use -start option.
```

```
~ > omniNames
```

```
Tue Jun 28 15:58:08 2005:
```

```
Read log file successfully.
```

```
Root context is IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f4e61
6d696e67436f6e746578744578743a312e30000001000000000000060000000010102000d0000003134
302e37372e31332e34390000f90a0b0000004e616d6553657276696365000200000000000008000000
0100000000545441010000001c0000000100000001000100010000000100010509010100010000000901
```

```
Checkpointing Phase 1: Prepare.
```

```
Checkpointing Phase 2: Commit.
```

```
Checkpointing completed.
```

CORBA usage for DIET

Every DIET entity must connect to the CORBA name server: it is the way services discover each others. The reference to the omniORB name server is written in a CORBA configuration file, whose path is given to omniORB through the environment variable `OMNIORB_CONFIG` (or, with **omniORB 4**, at compile time, through the configure script option: `--with-omniORB-config`). An example of such a configuration file is given in the directory `src/examples/cfgs` of the DIET source tree and installed in `<install.dir>/etc`. The lines concerning the name server in the omniORB configuration file are built as follows:

```
omniORB 3:
```

```
ORBInitialHost <name server hostname>
ORBInitialPort <name server port>
```

omniORB 4:

```
InitRef = NameService=corbaname::::<name server
port>
```

The name server port is the port given as an argument to the `-start` option of `omniNames`. You also need to update your `LD_LIBRARY_PATH` to point to `<install_dir>/lib`. So your `LD_LIBRARY_PATH` environment variable should now be :

```
LD_LIBRARY_PATH=<omniORB_home>/lib:<install_dir>/lib.
```

NB1: In order to avoid name collision, every agent must be assigned a different name in the name server; since they don't have any children, SeDs do not need names assigned to them and they don't register with the name server.

NB2: Each DIET hierarchy can use a different name server, or multiple hierarchies can share one name server (assuming all agents are assigned unique names). In a multi-MA environment, in order for multiple hierarchies to be able to cooperate it is necessary that they all share the same name server.

2.1.2 DIET configuration file

A configuration file is needed to launch a DIET entity. Some fully commented examples of such configuration files are given in the directory `src/examples/cfgs` of the DIET source files and installed in `<install_dir>/etc` ¹. Please note that:

- comments start with `'#'` and finish at the end of the current line,
- meaningful lines have the format: `keyword = value`, following the format of configuration files for omniORB 4,
- for options that accept 0 or 1, 0 means no and 1 means yes, and
- keywords are case sensitive.

Tracing API

```
traceLevel default = 1
```

This option controls debugging trace output. The following levels are defined:

level = 0	Print only errors
level < 5	Print errors and messages for the main steps (such as "Got a request") - default
level < 10	Print errors and messages for all steps
level = 10	Print errors, all steps, and some important structures (such as the list of offered services)
level > 10	Print all DIET messages AND omniORB messages corresponding to an omniORB traceLevel of (level - 10)

¹if there isn't `<install_dir>/etc` directory, please configure DIET with `--enable-examples` and/or run `make install` command in `src/examples` directory.

Client parameters

`MAName default = none`

This is a **mandatory** parameter that specifies the name of the Master Agent to connect to. The MA must have registered with this same name to the CORBA name server.

Agent parameters

`agentType default = none`

As DIET offers only one executable for both types of agent, it is **mandatory** to specify which kind of agent must be launched. Two values are available: `DIET_MASTER_AGENT` and `DIET_LOCAL_AGENT`. They have aliases, respectively `MA` and `LA`.

`name default = none`

This is a **mandatory** parameter that specifies the name with which the agent will register to the CORBA name server.

LA and SeD parameters

`parentName default = none`

This is a **mandatory** parameter for Local Agents and SeDs, but not for the MA. It indicates the name of the parent (an LA or the MA) to register to.

Endpoint Options

`dietPort default = none`

This option specifies the listening port of an agent or SeD. If not specified, the ORB gets a port from the system. This option is very useful when a machine is behind a firewall. By default this option is disabled.

`dietHostname default = none`

The IP address or hostname at which the entity can be contacted from other machines. If not specified, let the ORB get the hostname from the system; by default, omniORB takes the first registered network interface, which is not always accessible from the exterior. This option is very useful in a variety of complicated networking environments such as when multiple interfaces exist or when there is no DNS.

2.1.3 Example

Launching the MA

For such a platform, the MA configuration file could be:

```
# file MA_example.cfg, configuration file for an MA
agentType      =  DIET_MASTER_AGENT
name           =  MA_example
#traceLevel    =  1                # default
#dietPort      =  <port>           # not needed
#dietHostname  =  <hostname|IP>    # not needed
```

This configuration file is the only argument to the executable `dietAgent`, which is installed in `<install_dir>/bin`. Provided `<install_dir>/bin` is in your `PATH` environment variable, run

```
~ > dietAgent MA_example.cfg
```

Master Agent MA_example started.

Launching an LA

For such a platform, an LA configuration file could be:

```
# file LA_example.cfg, configuration file for an LA
agentType      =  DIET_LOCAL_AGENT
name           =  LA_example
parentName     =  MA_example
#traceLevel    =  1                # default
#dietPort      =  <port>           # not needed
#dietHostname  =  <hostname|IP>    # not needed
```

This configuration file is the only argument to the executable `dietAgent`, which is installed in `<install_dir>/bin`. This LA will register as a child of MA_example. Run

```
~ > dietAgent LA_example.cfg
```

Local Agent LA_example started.

Launching a server

For such a platform, a *SeD* configuration file could be:

```
# file SeD_example.cfg, configuration file for a SeD
parentName     =  LA_example
#traceLevel    =  1                # default
#dietPort      =  <port>           # not needed
#dietHostname  =  <hostname|IP>    # not needed
```

The *SeD* will register as a child of LA_example. Run the executable that you linked with the DIET SeD library, and do not forget that the first argument of the method call `diet_SeD` must be the path of the configuration file above.

Launching a client

Our client must connect to the MA_example:

```
# file client.cfg, configuration file for a client
MAName      =  MA_example
#traceLevel  =  1          # default
```

Run the executable that you linked with the DIET client library, and do not forget that the first argument of the method call `diet_initialize` must be the path of the configuration file above.