DIET Tutorial

Raphaël Bolze, Eddy Caron, Philippe Combes, Holly Dail and Christophe Pera
INRIA Rhone-Alpes

1 Introduction

This is a tutorial for DIET (Distributive Interactive Engineering Toolbox). It comes with three directories : `exercise2`, `exercise3` and `solutions`. The exercise directories contain the skeletons for the programs you will have to write, `solutions` contains the solutions to the exercises.

2 Exercise 1 : Installing and compiling DIET

The installation process is described in full in the User's Manual. In the following section we provide a quick start guide. Install DIET in the directory of your choice, for instance `${HOME}/DIET`. For this tutorial, FAST will not be used, and as regards omniORB, please add the `${OMNIORB_HOME}/bin` directory in your `PATH`, or give the following option to the `configure` script : `--with-omniORB=${OMNIORB_HOME}`

2.1 Dependencies

2.1.1 Hardware dependencies

DIET has fully tested on Linux i386 and i686 platforms and has tested on Solaris/Sparc, Linux/Sparc, Linux/i64, Linux/amd64 and Linux/Alpha platform and seems to be supported.

If you found a bug in DIET, please submit a bug report at `http://graal.ens-lyon.fr/bugzilla`. If you have multiple bugs to report, please make multiple submissions, rather than submitting multiple bugs in one report.

2.1.2 Software dependencies

CORBA is used for all communications inside the platform. So all of the three parts depend on it. The implementations of CORBA currently supported in DIET are :

- **omniORB 3** which depends on **Python 1.5**
- **omniORB 4** which depends on **Python 2.1** or later, and on **OpenSSL** if you would like your DIET platform to be secure.

We strongly recommend omniORB 4, as it is easier to install, and provides SSL support.

So far, only one CORBA service is needed : the Naming Service. This service is provided by a name server that must be launched before all DIET entities. Then, DIET entities must be passed a reference to the <host :port> of your name server. Later you will have to define the following environment variables (`$OMNIORB_CONFIG` and `$OMNINAMES_LOGDIR`), and/or, for **omniORB 4** only, at compile time, with the `--with-omniORB-config` and `--with-omniNames-logdir` options.

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

Since omniORB needs a thread-safe management of exception handling, compiling DIET with `gcc` requires at least `gcc3`.
Some examples provided in the DIET sources depend on the BLAS and ScaLAPACK libraries. However theses examples do not have to be compiled.

2.2 Compiling the platform

Once all dependencies are satisfied, untar the DIET archive and change to its root directory. A configure script will prepare DIET for compiling : its main options are described below, but please, run `configure --help` to get an up-to-date and complete usage description.

```
~> tar xzf DIET_1.1.tgz
~> cd DIET
~/DIET> ./configure --help=short
Configuration of DIET 1.1 :
...
```

2.2.1 Optional features for configuration

```
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --enable-maintainer-mode enable make rules and dependencies not useful
                          (and sometimes confusing) to the casual installer

  --enable-doc            enable the module doc, documentation about DIET
```

This option activates the compilation and installation of the DIET documents, which is disabled by default, because it is very sensitive to the version of your LaTeX compiler. The output postscript files are provided in the archive.

```
  --disable-examples      disable the module examples, basic DIET examples
```

This option deactivates the compilation of the DIET examples, which is enabled by default.

```
  --enable-BLAS           enable the module BLAS, an example for calling BLAS
                          functions through DIET
```

This option activates the compilation of the DIET BLAS examples, as a sub-module of examples (which means that this option has no effect if examples are disabled) - disabled by default.

```
  --enable-logservice     enable monitoring through LogService
```

This option enables the connection from DIET to the LogService monitoring software. It will enable DIET to generate monitoring data and send it to the LogService. Note that the support of LogService is built in. You do not have to install the LogService package to compile with this option.

```
 --enable-ScaLAPACK      enable the module ScaLAPACK, an example for calling
                          ScaLAPACK functions through DIET
```

This option activates the compilation of the DIET ScaLAPACK examples, as a sub-module of examples (which means that this option has no effect if examples are disabled) - disabled by default.

```
  --enable-Cichlid        enable generation of communication logs for Cichlid
  --enable-stats          enable generation of statistics logs
  --enable-multi-MA       enable multi-MA architecture
  --disable-dependency-tracking Speeds up one-time builds
  --enable-dependency-tracking  Do not reject slow dependency extractors
  --enable-shared=PKGS    build shared libraries default=yes
  --enable-static=PKGS    build static libraries default=yes
  --enable-fast-install=PKGS optimize for fast installation default=yes
  --disable-libtool-lock  avoid locking (might break parallel builds)
```

2.2.2 Optional packages for configuration

```
--with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
--without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
--with-gnu-ld           assume the C compiler uses GNU ld default=no
--with-pic              try to use only PIC/non-PIC objects default=use both
```

omniORB

```
--with-omniORB=DIR      specify the root installation directory of omniORB
--with-omniORB-includes=DIR
                        specify exact header directory for omniORB
--with-omniORB-libraries=DIR
                        specify exact library directory for omniORB
--with-omniORB-extra=ARG|"ARG1 ARG2 ..."
                        specify extra conftest.c -o conftest for the linker
                        to find the omniORB libraries (use "" in case of
                        several conftest.c -o conftest)
```

This group of options lets the user define all necessary paths to compile with omniORB. Generally, `--with-omniORB=DIR` should be enough, and the other options are provided for ugly installations of omniORB.
NB : having the executable `omniidl` in the PATH environment variable should be enough in most cases.

FAST

```
--with-FAST=DIR         installation root directory for FAST (optional)
--with-FAST-bin=DIR     installation directory for fast-config (optional)
```

This group of options lets the user define all necessary paths to compile with FAST. Generally, `--with-FAST=DIR` should be enough, and the other options are provided for difficult installations of FAST. There is no need to specify includes, libraries and extra arguments, since FAST provides a tool `fast-config` that does this job automatically.
NB1 : having the executable `fast-config` in the PATH environment variable should be enough in most cases.
NB2 : it is possible to specify `--without-FAST`, which overrides `fast-config` detection.

BLAS The BLAS [1] (Basic Linear Algebra Subprograms) are high quality ”building block” routines for performing basic vector and matrix operations. Level 1 BLAS do vector-vector operations, Level 2 BLAS do matrix-vector operations, and Level 3 BLAS do matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they’re commonly used in the development of high quality linear algebra software.

```
--with-BLAS=DIR  specify the root installation directory of BLAS (Basic
                 Linear Algebric Subroutines)

--with-BLAS-includes=DIR
                 specify exact header directory for BLAS
--with-BLAS-libraries=DIR
                 specify exact library directory for BLAS
--with-BLAS-extra=ARG|"ARG1 ARG2 ..."
                 specify extra conftest.c -o conftest for the linker to find
                 the BLAS libraries (use "" in case of several
                 conftest.c -o conftest)
```

This group of options lets the user define all necessary paths to compile with the BLAS libraries. Generally, `--with-BLAS=DIR` should be enough, and the other options are provided for difficult installation of the BLAS.
NB : these options have no effect if the module example and/or its sub-module BLAS are disabled.

ScaLAPACK The ScaLAPACK[2] (or Scalable LAPACK) library includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. It is currently written in a Single-Program-Multiple-Data style using explicit message passing for interprocessor communication. It assumes matrices are laid out in a two-dimensional block cyclic decomposition.

[1] `http://www.netlib.org/blas/`
[2] `http://www.netlib.org/scalapack/`

Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movements between different levels of the memory hierarchy. (For such machines, the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor.) The fundamental building blocks of the ScaLAPACK library are distributed memory versions (PBLAS) of the Level 1, 2 and 3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS and the BLACS. One of the design goals of ScaLAPACK was to have the ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

```
  --with-ScaLAPACK=DIR    specify the root installation directory of ScaLAPACK
                          (parallel version of the LAPACK library)

  --with-ScaLAPACK-includes=DIR
                          specify exact header directory for ScaLAPACK
  --with-ScaLAPACK-libraries=DIR
                          specify exact library directory for ScaLAPACK
  --with-ScaLAPACK-extra=ARG|"ARG1 ARG2 ..."
                          specify extra conftest.c -o conftest for the linker to find
                          the ScaLAPACK libraries (use "" in case of several
                          conftest.c -o conftest)
```

This group of options lets the user define all necessary paths for compilation with the ScaLAPACK libraries. Normally, `--with-ScaLAPACK=DIR` could be enough, but the other options are provided because the installation of the ScaLAPACK libraries is often difficult, and thus difficult to detect automatically. For instance, the `--with-ScaLAPACK-extra` option is useful to integrate BLACS and MPI libraries, which are useful for ScaLAPACK.
NB : these options have no effect if the module example and/or its sub-module ScaLAPACK are disabled.

2.2.3 From configuration to compilation

An important option for configure scripts is `--prefix=`, which specifies where binary files, documents and configuration files will be installed. It is important to set this option : it otherwise defaults to `DIET/install`.

The configuration will return with an error if no ORB was found. So please help the configure script to find the ORB with the `--with-omniORB*` options.

If everything went OK, the configuration ends with a summary of the options that were selected and what it was possible to get. The output will look like :

```
~/DIET > ./configure --enable-doc --enable-BLAS
DIET successfully configured as follows:
 - documents:       yes
 - examples:        dmat_manips, file_transfer, scalars
 - FAST:            found /home/username/DIET/FAST/bin/fast-config
 - ORB:             omniORB 4 in /usr
 - prefix:          /home/username/DIET/install
 - multi-MA:        no
 - LogService:      no
 - target platform: i686-pc-linux-gnu

Please run make help to get compilation instructions.
~/DIET > make help
======================================================================
 Usage : make help     : shows this help screen
         make agent    : builds DIET agent executable
         make SeD      : builds DIET SeD library
         make client   : builds DIET client library
         make examples : builds basic examples
         make          : builds everything configured
         make install  : copy files into /home/username/DIET/install
======================================================================
```

This helps the user to choose the DIET parts for installation and compilation. It is recommended for beginners to compile the client, the SeD and the agent in one single step with `make all`. But please, pay attention to the fact that `make all` does not install DIET in the prefix provided at configuration time. To do this, run `make install`.

`make install` will run the compilation for all DIET entities before the installation itself. Thus, if the user wants to compile only the agent, for instance, and install it, he must run :

```
~/DIET > cd src/agent
~DIET/src/agent > make install
```

3 Exercise 2 : An example of matrix computation

With this example we learn how to program a simple client/server application that uses DIET. We will use the context of matrix computation to make this program more realistic. We will implement a basic scalar by matrix product. Then, we will test this program in different execution schemes.

3.1 File skeletons

The exercise2 directory, located in your home directory contains all the skeleton files needed for a quick start. Useful pieces of software are also included in them. This directory contains the following files :

Makefile management of dependencies between source and compiled files

server.c program implementing the service (scalar by matrix product)

client_smprod.c program using the service defined in server.c : the matrix is stored in memory

client_smprod_file.c same program as client_smprod.c, except that the matrix is passed as a file to the server

3.2 Server-side implementation

Using the skeleton of program `server.c`, write a service of scalar by matrix product. This service will have the following parameters :

parameter	type
a scalar	double
a matrix to be multiplied	double
the time needed for the product to compute	float

The intial matrix is overwritten by the result. The matrix will be stored in memory.

To start, try to define a detailed interface for the service, i.e. a precise definition of the service *profile*. To do so, look for **in**, **inout** and **out** parameters.

Next, program the solve function `solve_smprod`, and also the initialization of the service in the `main` function.

```
int solve_smprod(...) {

}
```

The following function is given to help you :

```
int scal_mat_prod(double alpha, double *M, int nb_rows, int nb_cols, float *time)
```

It mutliplies the scalar `alpha` by the matrix `M` a `nb_rows` by `nb_cols` matrix. The results are the matrix `alpha` $\times$ `M` and the time of this operation in seconds.

3.3 Client-side implementation

Using the `client_smprod.c` skeleton file, write a client for the service defined above. You will need to initialize a matrix and a scalar with known values. That way, you will be able to verify if the answer is correct or not.

You will have to remember that the profile used in the client must match exactly the server profile.

3.4 Setting up and testing the client/server

The file `env_vars.bash.in` for bash shell (respectively `env_vars.csh.in` for tcsh shell) contains all the environment variables needed for the execution of the programs. Verify the values of those variables, then load this file using the following method :

```
~/> source env_vars.bash
```

When done with this operation, you need to start the name server of omniORB : omniNames. To do that, you must give a port number with the `-start` option, on which the service will be opened (and on which the server "listens") :

```
$ omniNames -start <port>

Tue Dec 11 14:10:28 2003:

Starting omniNames for the first time.
Wrote initial log file.
Read log file successfully.
Root context is
IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e674
36f6e746578744578743a312e3000000100000000000000600000000010102000d000000313430
2e37372e31332e36310000554f0b0000004e616d655365727669636500020000000000000080
0000001000000005454410100000001c000000010000000100010001000000010001050901010 0
0100000009010100
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
$
```

Then, you have to copy this port number in the omniORB configuration file : the name and location of this file is given by the environment variable `OMNIORB_CONFIG` which is defined in the `env_vars.bash` file.

Using the examples configuration files found in `$DIET_HOME/src/examples/cfgs` write your own configuration files (suggestion : place them in a `cfgs` directory). You will want to create a hierarchy of agents to make it interesting. This hierarchy should contain at least one MA and one LA.

Compile server and client with the `Makefile`, then, launch the server and the client.

Finally, launch several servers in different windows (you can use same or different distributed computers). With different windows, you will see which one is activated. You should experiment with different hierarchies.

3.5 Another version of the service

In this part, you will modify the server to make it support a slightly different version of the scalar by matrix product. The matrix will be transmitted as a file, and not anymore in memory.

DIET doesn't impose anything about the data format of files, but it would be a good idea to facilitate your work to use the data format used in the skeleton files. This format is just simple text : the file contains a series of numbers, separated by 'space' characters. The meaning of the numbers is as follows :

– matrix dimensions (number of rows, number of columns)
– matrix values

Create a file containing a matrix, then implement a new service "`smprod_file`" with the following parameters :

parameter	type
a scalar	double
a file containing matrix to be multiplied	double
the time needed for the product to compute	float

The file is overwritten by the result.

4 Exercise 3 : yet another service : The BLAS dgemm

To compile programs of this exercise, the BLAS library (Basic Linear Algebric Subroutines) is required. You also should have configured DIET with `--enable-BLAS` option.

DGEMM - perform one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C
The `dgemm_` function is part of the BLAS. It performs the following matrix-matrix computation :

$$C := \alpha AB + \beta C$$

α and β are scalars, and *A*, *B* and *C* are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix

This exercise aims at adding a new service in a DIET platform, that performs the `dgemm_` computation. The idea is to interface the existing `dgemm_` function to a DIET SeD. Here is its prototype :

```
void dgemm_(char   *tA,
            char   *tB,
            int    *m,
            int    *n,
            int    *k,
            double *alpha,
            double *A,
            int    *lda,
            double *B,
            int    *ldb,
            double *beta,
            double *C,
            int    *ldc);
```

All parameters are given by address. Parameters `alpha`, `beta`, `m`, `n`, `k`, `A`, `B` and `C` correspond exactly to their respective roles in the computation. `lda`, `ldb` and `ldc` are the *leading dimensions* of the corresponding matrices. Since matrices are stored in a classical one-dimension array, it is important to specify if they are stored by rows or by columns. `*tA` and `*tB` are characters which have the following semantics :

tA	Storage order of A (*m*,*k*)	
'T'	row-major	[row 1, row 2, ... , row *m*]
'N'	column-major	[col 1, col 2, ... , col *k*]

For this exercise, there is no need to explore all possibilities offered by the storage order or the leading dimension. Just set `*tA` and `*tB` to 'N', and `lda`, `ldb` and `ldc` to the number of rows of the corresponding matrix.

Once you have specified the *profile* of the service, program a server that implements this service, and a test client, using the file skeletons in the **exercise3** directory. Matrices will be stored in memory. Eventually, test the client/server architecture, through DIET, in different contexts of execution.