Distributed Systems
Introduction to DIET: Useful information

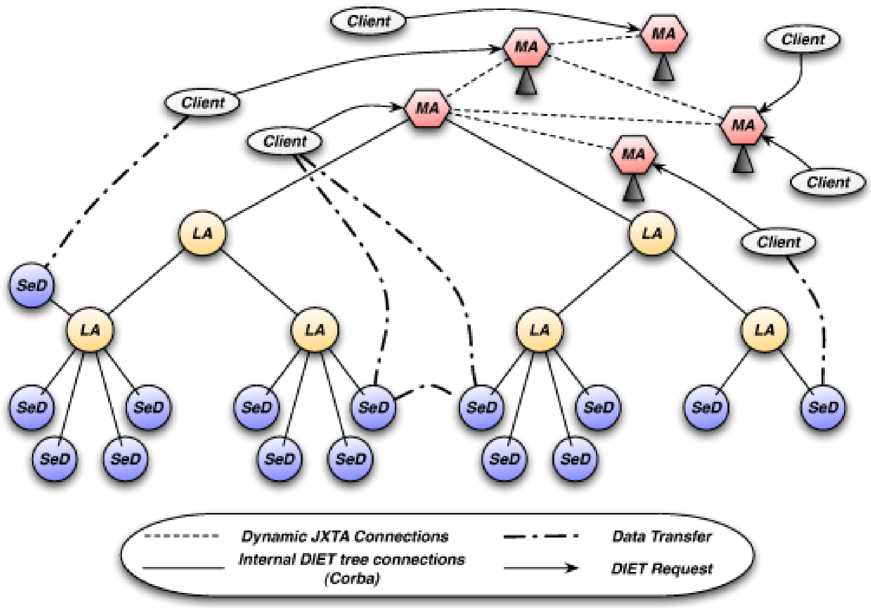
Mehdi.Diouri@ens-lyon.fr

26 novembre 2012

Introduction

The Distributed Interactive Engineering Toolbox (Diet) project is focused on the development of a scalable middleware with initial efforts focused on distributing the scheduling problem across multiple agents. This middleware is able to find an appropriate server according to the information given in the client's request (e.g., problem to be solved, size of the data involved), the performance of the target platform (e.g., server load, available memory, communication performance) and the local availability of data stored during previous computations.

The Diet component architecture is structured hierarchically for improved scalability. Such an architecture is flexible and can be adapted to diverse environments, including arbitrary heterogeneous computing platforms. The Diet toolkit is implemented in CORBA and thus benefits from the many standardized and stable services provided by the various freely-available CORBA implementations. CORBA systems provide a remote method invocation facility with a high level of transparency.



The DIET framework is composed of several components :

- A **Client** is an application that uses the Diet infrastructure to solve problems using an RPC approach. Clients access Diet via various interfaces : web portals, Problem Solving Environments (PSEs) such as Scilab, or programmatically using published C or C++ APIs.
- A **SeD**, or server daemon, acts as the service provider, exporting functionalities via a standardized computational service interface ; a single SeD can offer any number of computational services. A SeD can also serve as the interface and execution mechanism for either a stand-alone interactive machine or a parallel supercomputer, by interfacing with its batch scheduling facility.
- **Agents**, which are the components that facilitate the service location and invocation interactions of clients and SeDs.

Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single **Master Agent (MA)** and **several Local Agents (LA**). The figure shows an example of a DIET hierarchy.

The **Master Agent** of a Diet hierarchy serves as the distinguished entry point from which the services contained within the hierarchy may be logically accessed. Clients identify the Diet hierarchy using a standard CORBA naming service. Clients submit requests - composed of the name of the specific computational service they require and the necessary arguments for that service - to the MA. The MA then forwards the request to its children, who subsequently forward the request to their children, such that the request is eventually received by all SeDs in the hierarchy. SeDs then evaluate their own capacity to perform the requested service ; capacity can be measured in a variety of ways including an application-specific performance prediction, general server load, or local availability of datasets specifically needed by the application. SeDs forward this capacity information back up the agent hierarchy. Based on the capacities of individual SeDs to serve the request at hand, agents at each level of the hierarchy reduce the set of server responses to a manageable list of server choices with the greatest potential. The server choice can be made specific to any kind of application using plug-in schedulers at each level of the hierarchy.

Installing DIET

Some softwares and libraries are required : CMake, omniORB and BLAS.
The installation process is described in the User's Manual.
Install Diet in the directory of your choice, for instance $\${HOME}\backslash DIET$.

To install Diet, basically what you need to do is the following :
- create a temporary working directory, and change directory to it
- use CMake to run diet configuration : ***ccmake diet_src/***

Press 'c' to run initial configuration, and select compilation options, for this tutorial you only need to change the installation directory (CMAKE_INSTALL_PREFIX), turn on examples compilation if you want to (DIET_BUILD_EXAMPLES and DIET_BUILD_BLAS_EXAMPLES).

Then press 'c' (configure), then 'g' (generate), CMake will exit, you can finally type ***make && make install***.

You will need to set your PATH and LD_LIBRARY_PATH respectively to <DIET_HOME>/bin and <DIET_HOME>/lib.

Server-side implementation

You can have a look at in, inout and out parameters :
`http://graal.ens-lyon.fr/DIET/UsersManualDIET2.3/node60.html`
You can also have a look at the page concerning data management.
`http://graal.ens-lyon.fr/DIET/UsersManualDIET2.3/node33.html`

Data Types

There are two kinds of data types : base types, and composite types. Base types have the following semantics :

Type	Description	Size in octets
DIET_CHAR	Character	1
DIET_BYTE	Octet	1
DIET_INT	Signed integer	4
DIET_LONGINT	Long signed integer	8
DIET_FLOAT	Simple precision real	4
DIET_DOUBLE	Double precision real	8
DIET_SCOMPLEX	Simple precision complex	8
DIET_DCOMPLEX	Double precision complex	16

Here are the composite types you can use :

Type	Possible base types
DIET_SCALAR	all base types
DIET_VECTOR	all base types
DIET_MATRIX	all base types
DIET_STRING	DIET_CHAR
DIET_FILE	DIET_CHAR

Persistence Modes

Various persistence modes for data
– DIET_VOLATILE : No persistency at all.
– DIET_PERSISTENT_RETURN : (valid for INOUT and OUT arguments only) Data are saved on the server and a copy is sent back to the client after the computation is complete.
– DIET_PERSISTENT : Data are saved on the server and nothing is brought back to the client.
– DIET_STICKY : Data are saved on the server, they cannot been moved from there to another server, and thus cannot be sent back to the client.

Initializing the Service Table

You have the following functions to define the number of services a server can handle, and add the services to the service table. We do not need the diet_convertor_t *cvt, so you can set it to NULL.

```
int diet_service_table_init(int max_size);

int diet_service_table_add(diet_profile_desc_t *profile,
                           diet_convertor_t *cvt,
                           diet_solve_t solve_func);

void diet_print_service_table();
```

Creating a Profile

Here are the main functions available to create a profile.

```
diet_profile_desc_t* diet_profile_desc_alloc(const char* path,
                                             int last_in,
                                             int last_inout,
                                             int last_out);

int diet_generic_desc_set(diet_arg_t *desc,
                          diet_data_type_t  type,
                          diet_base_type_t  base_type );

int diet_profile_desc_free(diet_profile_desc_t* desc);

diet_arg_t* diet_parameter(diet_profile_t* profile, int position);
```

You need to define the profile of your service. last_in, last_inout and last_out correspond to the index in an array of the last IN, INOUT and OUT parameters (for example if last_in = 0 then we have 1 IN parameter, if last_in = last_inout, then there is no INOUT parameter). diet_generic_desc_set has to be called for each parameter in the profile. For example :

```
  diet_generic_desc_set(diet_param_desc(profile,0),
                        DIET_SCALAR, DIET_DOUBLE);
```

Getting and Setting IN, INOUT and OUT Arguments

You have to get IN and INOUT arguments using functions having the following syntax :

```
int diet_*_get(diet_arg_t* arg, void** value,
               diet_persistence_mode_t* mode);
```

For the time being, you can set the mode to NULL. For example you could have for an IN parameter :

```
  double* coeff;
  ...
  diet_scalar_get(diet_parameter(pb,0), &coeff, NULL);
```

The diet_*_get functions are defined in DIET_data.h.

Do not forget that the necessary memory space for OUT arguments is allocated by DIET. So the user should call the diet_*_get functions to retrieve the pointer to the zone his/her program should write to.

You have to set INOUT and OUT arguments using functions having the following syntax :

```
int diet_*_desc_set(diet_arg_t* arg, ...);
```

where * can be for example scalar, matrix...

For example, you could have for an OUT parameter :

```
  float* time;
  ...
  diet_scalar_get(diet_parameter(pb,2), &time, NULL);
  ...
  diet_scalar_desc_set(diet_parameter(pb,2), time);
```

You can also have a look at diet_basic/include/DIET_server.h and diet_basic/include/DIET_data.h.

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 so as to 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 `http://graal.ens-lyon.fr/DIET/UsersManualDIET2.3/node51.html`

You can also write another version of the client, using asynchronous calls to Diet.

Initializing and Finalizing a DIET Session

The client program must open its DIET session with a call to diet_initialize, which parses the configuration file to set all options and get a reference to the DIET Master Agent. The session is closed with a call to diet_finalize, which frees all resources associated with this session on the client.

Note that memory allocated for all INOUT and OUT arguments brought back onto the client during the session is not freed during diet_finalize ; this allows the user to continue to use the data, but also requires that the user explicitly free the memory.

The user must also free the memory he or she allocated for IN arguments.

```
/* Before any call to DIET */
diet_error_t diet_initialize(char* config_file_name, int argc, char* argv[]);

/* When you do not need DIET anymore */
diet_error_t diet_finalize();
```

Create Profile and Set Arguments

Allocate a DIET profile with memory space for its arguments. pb_name is deep-copied. If no IN argument, please give -1 for last_in. If no INOUT argument, please give last_in for last_inout. If no OUT argument, please give last_inout for last_out. Once the profile is allocated, please use set functions on each parameter. For example, the nth argument is a matrix :

```
diet_matrix_set(diet_parameter(profile,n),
                mode, value, btype, nb_r, nb_c, order);

diet_profile_t* diet_profile_alloc(char* pb_name,
                                   int last_in, int last_inout, int last_out);

/* Once you do not need the profile anymore */
int diet_profile_free(diet_profile_t* profile);
```

You have to set IN and INOUT arguments using functions having the following syntax :

```
int diet_*_set(diet_arg_t* arg, ...);
```

where * can be for example scalar, matrix...

Calling a Service

Just call :

```
diet_error_t diet_call(diet_profile_t* profile);
```

This function will return once the service has finished. You can also make asynchronous calls.

```
diet_error_t diet_call_async(diet_profile_t* profile, diet_reqID_t* reqID);
```

You then have to wait for the end of the services.
You can either wait for one call to finish :

```
diet_error_t  diet_wait(diet_reqID_t reqID);
```

Or several calls, either all of them, or any of them, on all submitted requests or on a subset of them :

```
diet_error_t diet_wait_and(diet_reqID_t* IDs, size_t length);
diet_error_t diet_wait_or(diet_reqID_t* IDs,size_t length,diet_reqID_t* IDptr);
diet_error_t diet_wait_all();
diet_error_t diet_wait_any(diet_reqID_t* IDptr);
```

Freeing Data

Free the amount of data pointed at by the value field of an argument. This should be used ONLY for VOLATILE data,
– on the server for IN arguments that will no longer be used
– on the client for OUT arguments, after the problem has been solved, when they will no longer be used.

NB : for files, this function removes the file and frees the path (since it has been dynamically allocated by DIET in both cases)

```
int diet_free_data(diet_arg_t* arg);
```

You can also have a look diet_basic/include/DIET_client.h and diet_basic/include/DIET_data.h.

Setting up and testing the client/server locally

The syntax of an agent configuration file is presented in the appendix.
An agent can be run using the following command :

```
diet_basic/bin/dietAgent agent.cfg
```

If an element crashes and return an error such as :

```
DIET ERROR: exception caught (NO_RESOURCES)
while attempting to connect to the CORBA name server.
DIET ERROR: cannot locate Master Agent MA1.
DIET initialization failed !
terminate called after throwing an instance of 'omni_thread_fatal'
```

make sure your OMNIORB_CONFIG environment variable is set to a valid omniORB configuration file (the one used for the deployment of the hierarchy).

Deploying on several nodes

Deploying a whole DIET hierarchy on many nodes may be quite tiresome. A tool has been specifically designed to deploy DIET platforms : GoDIET. The DIET hierarchy is described in an XML file. You won't have to write the whole XML file. We provide a script which creates a basic DIET platform composed of an MA, and one or many SeDs ?. The script can be found in scripts/make_diet_platform_basic.py :

```
Usage: ./make_diet_platform_basic.py <nb SeD> <nodefile> <output_file>
                                <GoDIEt_scratch_runtime> <path to SeD>
  Creates a simple DIET hierarchy: 1 MA, <nb SeD> SeDs
   - <nb SeD>: number of SeDs
   - <nodefile>: file containing the list of nodes,
   - <output_file>: the XML file to create
   - <GoDIEt_scratch_runtime>: path to an existing directory
    where GoDIET will write the configuration files
   - <path to SeD>: path to SeD binary
```

Then, you just need to run :

```
godiet <XML file>
```

Appendix

```
#*****************************************************************************#
# traceLevel for the DIET agent:
#    0  DIET prints only warnings and errors on the standard error output.
#    1  [default] DIET prints information on the main steps of a call.
#    5  DIET prints information on all internal steps too.
#   10  DIET prints all the communication structures too.
#  >10  (traceLevel - 10) is given to the ORB to print CORBA messages too.
#*****************************************************************************#
traceLevel = 1
#*****************************************************************************#
# (*) agentType: Master Agent or Local Agent ? As there is only one executable
#   for both agent types, it is COMPULSORY to specify the type of this agent.
#*****************************************************************************#
agentType = DIET_MASTER_AGENT # or MA if this agent is a master agent
                              # or LA or DIET_LOCAL_AGENT if this agent is a
                              # local agent
#*****************************************************************************#
# (*) name: the name of this MA. The ORB configuration files of the clients
#   and the children of this MA (LAs and SeDs) must point at the same CORBA
#   Naming Service as the one pointed at by the ORB configuration file of
#   this agent.
#*****************************************************************************#
name = MA1
#*****************************************************************************#
# (*) parentName: the name of the agent to which the LA will register. This
#   agent must have registered at the same CORBA Naming Service that is
#   pointed to by your ORB configuration.
#*****************************************************************************#
# parentName = MA1 only useful for a local agent
```