Data management in DIET

GRAAL Research Team

February 11, 2005

Abstract

As an example to illustrate this proposition, and particularly to point out the paradigm of data management transparency, we present the Data Management Service called DTM (Data Tree Manager) of DIET.

1 Motivations

The data management API we have developed in DIET follows the requirements of transparency and long data lifetime. We think that a data must survive to a single client session and that the data management can to be hidden to the end-user. An unique identifier is associated with each persistent data.

We are currently working on the way to manage heterogeneity of data sources. We so will give the possibility to write a data to or read a data from a storage resource or a specific computational server. We are also thinking about the problematics of data lifetime.

2 Client API

Persistence mode Our approach is based on the persistence mode of a data. A client can choose whether a data will be persistent inside the platform or not. This property is called the `persistence mode` of a data. We have defined several modes of data persistence as shown in the following table.

mode	Description
DIET_VOLATILE	not stored
DIET_PERSISTENT_RETURN	stored on server, movable and copy back to client
DIET_PERSISTENT	stored on server and movable
DIET_STICKY	stored and non movable
DIET_STICKY_RETURN	stored, non movable and copy back to client

Table 1: Persistence Mode.

Once this mode has been set for a data, the agent has the responsibility to assign the data identifier to the persistent data. After the call is performed, the client has the knowledge of the identifier assigned to the data. It has only to provide it in next calls.

The data identifier The data identifier is a string field. It is unique and so data is fully identified. This identifier is not bound to a location but is only a reference to the data. This provides flexibility for data redistribution between computational servers. In DIET, a data is not only a value plus eventually an identifier. A data is the value and some characteristics as the size, the type, the path and the base_type. Currently, we are adding the `host` value to be GridRPC compliant.

Standard DIET call vs persistence DIET call The advantages of our approach is that the client API is not modified by the use of data persistence. Only few methods are added.

The `store_id()` method allows the handle of a data to be stored in a local client file. This will be helpful to use data in other session for the same client or for others clients.

```
store_id(char *handle, char *msg);
```

The `diet_use_data()` method allows the use of a data stored in the platform identified by its handle. The description of the data (its characteristics) is also stored, therefore the client has not to build when it submits other problems.

```
diet_use_data(char *handle);
```

The `diet_free_persistent_data()` method allows to free the persistent data identified by `handle` from the platform.

```
diet_free_persistent_data(char *handle);
```

The `diet_read_data(char * handle)` method allows to read a data identified by `handle` already stored inside the platform.

```
diet_data_t diet_read_data(char *handle);
```

2.1 Client API Examples

without data persistence The figure 1 shows that when clients do not need to manage data persistence they do not need to use the data API. The standard code is not modified.

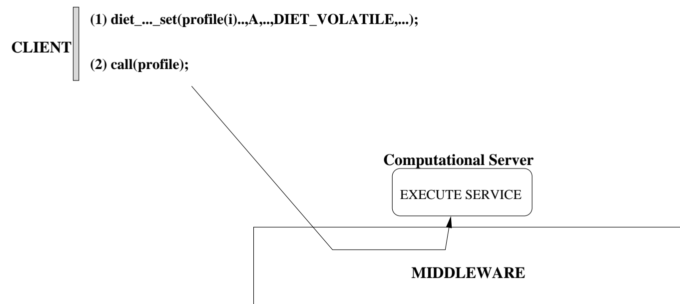

Figure 1: Simple DIET call with an INPUT data.

Add input persistent data to the platform Adding input data to the platform consist in just giving the `PERSISTENT` value to the persistent flag.

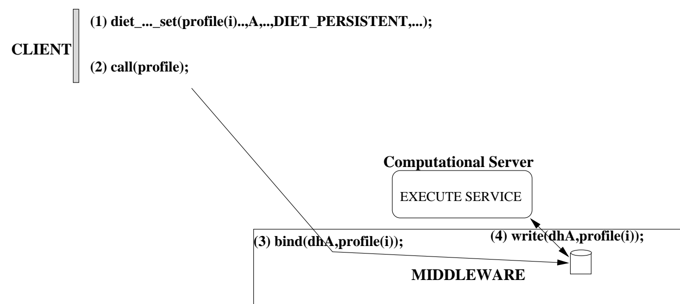

Figure 2: INPUT persistent data.

Add ouput persistent data to the platform In the case of output data, we implement the possibility for the client to retrieve or not the result of its computations. It is very useful for intermediate results. Thus, for a sequence of calls, all and intermediate results can be defined as `PERSISTENT` whereas the output data which interest the client are set to `PERSISTENT_RETURN`.

Managing INOUT arguments We note that only the result will be stored inside the platform.

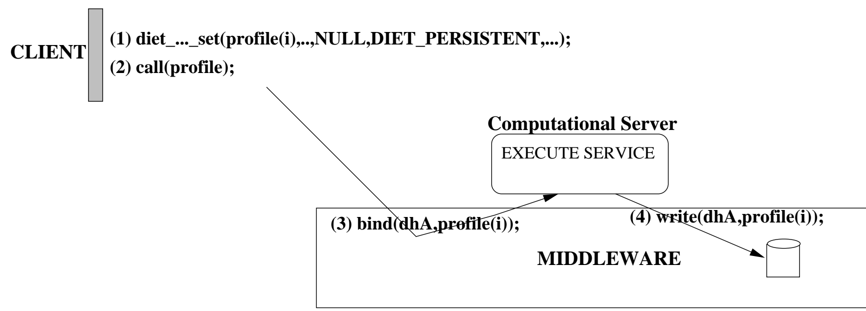

Figure 3: OUTPUT persistent data.

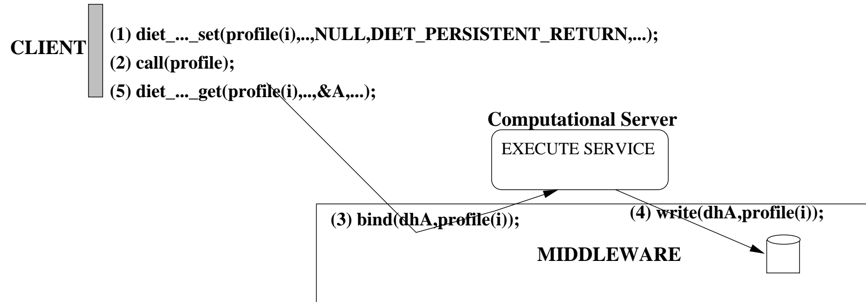

Figure 4: OUT persistent data with return value.

Multiple calls For a sequence of calls with persistent data, the client can write its code in several ways. In fact, once the first call performed, the client has the knowledge of the value of the data identifier. To use it for next calls in the session, he needs to use the `use_data` function. The parameter of this function can be : (1) the string value of the identifier, (2) a reference to the argument passed in the former call (i.e profile[i].desc.id which contains the data identifier, the profile must not be freed). The figure 6 illustrates different possibilities.

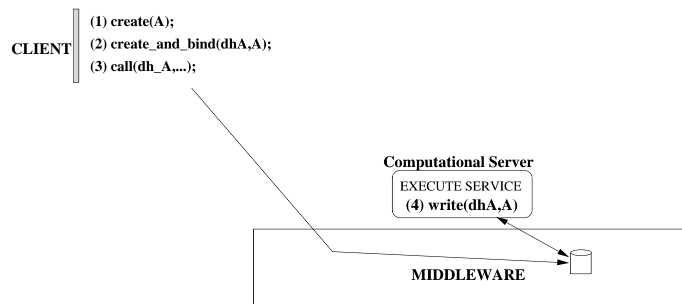

Figure 5: INOUT persistent data.

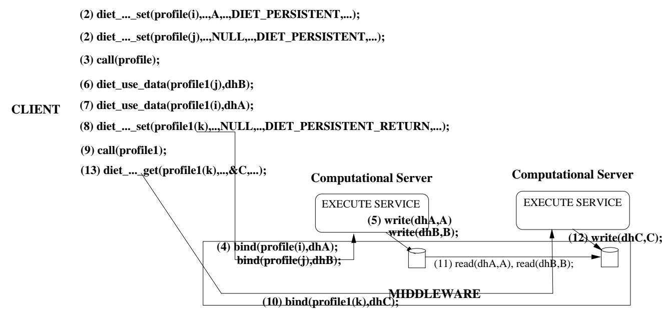

Figure 6: Multiple calls.

2.2 DIET code examples

Simple call In the first example of a DIET code using persistence an application computes the number of occurrences of a given letter in a given file.

```
int
main(int argc, char* argv[])
{
  char* path = NULL;
  char letter;
  diet_profile_t* profile = NULL;
  int *size = NULL;
  int *occurence = NULL;
  path = argv[2];
  letter = argv[3][0];
  if (diet_initialize(argv[1], argc, argv)) {
    fprintf(stderr, "DIET initialization failed !\n");
    return 1;
  }
  /* profile allocation : 2 INPUT data, 0 INOUT data, 2 OUTPUT DATA */
  profile = diet_profile_alloc("Number_of_letter_2", 1, 1, 3);

/* first argument : INPUT file */
  if (diet_file_set(diet_parameter(profile,0), DIET_PERSISTENT, path)) {
    printf("diet_file_set error\n");
    return 1;
  }
  /* 2nd INPUT parameter : letter */
  diet_scalar_set(diet_parameter(profile,1), &letter, DIET_VOLATILE, DIET_CHAR);
  diet_scalar_set(diet_parameter(profile,2), NULL, DIET_VOLATILE, DIET_INT);
  diet_scalar_set(diet_parameter(profile,3), NULL, DIET_VOLATILE, DIET_INT);


  if (!diet_call(profile)) {
/* stores the identifier in a local file */
    store_id(profile->parameters[0].desc.id,"text file");
    diet_scalar_get(diet_parameter(profile,2), &size, NULL);
    diet_scalar_get(diet_parameter(profile,3), &occurrence, NULL);
  }

  printf("file name : %s (size = %d ) \n",path,*size);
  printf("Number of %c in file %s is : %d  \n",letter,path,*occurrence);
```

```
  printf("DIET_id of file : %s\n",profile->parameters[0].desc.id);

  diet_profile_free(profile);
  diet_finalize();
  return 0;
}
```

Data already stored The second example shows how to use a data already stored inside the platform. The target application is the same as for the first experiment.

```
/* argv[1]: client config file path
   argv[2]: path of the file to transfer */

int
main(int argc, char* argv[])
{
  char* data_id = NULL;
  char letter;
  diet_profile_t* profile = NULL;
  int *size = NULL;
  int *occurrence = NULL;

  data_id = argv[2];
  letter = argv[3][0];
  if (diet_initialize(argv[1], argc, argv)) {
    fprintf(stderr, "DIET initialization failed !\n");
    return 1;
  }
  profile = diet_profile_alloc("Number_of_letter_2", 1, 1, 3);

  /* just giving the identifier of the file stored inside the platform */
  diet_use_data(diet_parameter(profile,0),data_id);

  diet_scalar_set(diet_parameter(profile,1), &letter, DIET_VOLATILE, DIET_CHAR);
  diet_scalar_set(diet_parameter(profile,2), NULL, DIET_VOLATILE, DIET_INT);
  diet_scalar_set(diet_parameter(profile,3), NULL, DIET_VOLATILE, DIET_INT);
  diet_scalar_set(diet_parameter(profile,4), NULL, DIET_VOLATILE, DIET_FLOAT);

  if (!diet_call(profile)) {
    diet_scalar_get(diet_parameter(profile,2), &size, NULL);
    diet_scalar_get(diet_parameter(profile,3), &occurrence, NULL);
    diet_scalar_get(diet_parameter(profile,4), &server_time, NULL);
  }

  printf("file name : %s (size = %d ) \n",data_id,*size);
  printf("Number of %c in file %s is : %d  \n",letter,data_id,*occurrence);
  printf("DIET_id of file : %s\n",profile->parameters[0].desc.id);

  diet_profile_free(profile);
  diet_finalize();
  return 0;
}
```

Multiple calls The third example shows how to use persistence in a single session with several calls. Note that each service is running on a single server.

```
int
main(int argc, char* argv[])
{
  ..........
  ............
  if (diet_initialize(argv[1], argc, argv)) {
    fprintf(stderr, "DIET initialization failed !\n");
    return 1;
  }
  .............
  ..............
  strcpy(path,"MatPROD");
  profile = diet_profile_alloc(path, 1, 1, 2);
  diet_matrix_set(diet_parameter(profile,0),
  A, DIET_PERSISTENT, DIET_DOUBLE, mA, nA, oA);
  print_matrix(A, mA, nA, (oA == DIET_ROW_MAJOR));
  diet_matrix_set(diet_parameter(profile,1),
  B, DIET_PERSISTENT, DIET_DOUBLE, mB, nB, oB);
  print_matrix(B, mB, nB, (oB == DIET_ROW_MAJOR));
  diet_matrix_set(diet_parameter(profile,2),
  NULL, DIET_PERSISTENT, DIET_DOUBLE, mA, nB, oC);
  if (!diet_call(profile)) {
    diet_matrix_get(diet_parameter(profile,2),&C, NULL, &mA, &nB, &oC);
    store_id(profile->parameters[2].desc.id,"C double matrix");
    store_id(profile->parameters[1].desc.id,"B double matrix");
    store_id(profile->parameters[0].desc.id,"A double matrix");
    print_matrix(C, mA, nB, (oC == DIET_ROW_MAJOR));
  }
  strcpy(path,"MatSUM");
  profile2 = diet_profile_alloc(path, 1, 1, 2);

  printf("second pb\n\n");
  diet_use_data(diet_parameter(profile2,0),profile->parameters[2].desc.id);
  diet_matrix_set(diet_parameter(profile2,1),
  E, DIET_PERSISTENT, DIET_DOUBLE, mA, nB, oE);
   print_matrix(E, mA, nB, (oE == DIET_ROW_MAJOR));
  diet_matrix_set(diet_parameter(profile2,2),
  NULL, DIET_PERSISTENT, DIET_DOUBLE, mA, nB, oD);

  if (!diet_call(profile2)) {
    diet_matrix_get(diet_parameter(profile2,2), &D, NULL, &mA, &nB, &oD);
   print_matrix(D, mA, nB, (oD == DIET_ROW_MAJOR));
   store_id(profile2->parameters[2].desc.id,"D double matrix");
   store_id(profile2->parameters[1].desc.id,"E double matrix");
  }

  strcpy(path,"T");
  printf("third  pb  = T\n\n");
  strcpy(path,"T");
  profile1 = diet_profile_alloc(path, -1, 0, 0);

  diet_use_data(diet_parameter(profile1,0),profile->parameters[0].desc.id);
  if (!diet_call(profile1)) {
    diet_matrix_get(diet_parameter(profile1,0), NULL, NULL, &mA, &nA, &oA);
     print_matrix(A, mA, nA, (oA == DIET_ROW_MAJOR));

    }

  printf ("next....");

  printf(" \nRemoving all persistent data..........." );

  diet_free_persistent_data(profile->parameters[0].desc.id);
```

```
  diet_free_persistent_data(profile->parameters[1].desc.id);
  diet_free_persistent_data(profile->parameters[2].desc.id);
  diet_free_persistent_data(profile2->parameters[1].desc.id);
  diet_free_persistent_data(profile2->parameters[2].desc.id);
  printf(" \n..................data removed\n\n");
  diet_profile_free(profile);
  diet_profile_free(profile1);
  diet_profile_free(profile2);
  diet_finalize();

  return 0;
}
```