
Project acronym: EDGeS

Project full title: Enabling Desktop Grids for e-Science

Grant agreement no.:RI 211727


# Tutorial:

# Application Porting with the DC-API

# Server side

Author: Gabor Szmetanko/ Attila Csaba Marosi

Date: 03/07/2009

# 1 Preparation

Now that we have a working client application, we can start writing the master. We are still in **~boinc-sample/edges-tut-files/uppercase_dg.** The directory contains the skeleton of the master application (master.cpp). We will extend that skeleton with new code to create the master application for the Uppercase application.

## *1.1 The skeleton*

To make the application porting process easier for you, we have already prepared the skeleton of the master. The skeleton implements all the functionality not related to the DC-API.

The skeleton has the following functions:

- **createWork**: creates and submits WUs
- **processResult**: processes a work unit
- **createFinalResult**: generates the final output
- **printHelp**: prints the help text
- **main**: connects them all together

It also contains some basic include statements, constants, and global variables.

Some of the functions are already implemented, while others should be completed by you. Places in the source code, where you should insert new code, are marked with a comment similar to the following:

```
/* TODO: Declare the global variables for the householding of the WUs */
```

# 2 Initialisation of the DC-API

## *2.1 Open the master.cpp source file in your preferred editor*

## *2.2 Insert the new include statement at the top of the file*

```
#include <stdlib.h>
#include <getopt.h>
#include <string.h>
#include <limits.h>
#include <stdio.h>
#include <errno.h>
#include <dc.h>

#include "common.h"
```

New lines are coloured **green** in the text. Leave everything else unchanged! In the following sections, new code blocks will always be coloured **green**.

### *2.3 Insert a call to the DC-API initialisation function after the command line processing section in the main function*

```
int main(int argc, char *argv[]) {
      …
      if (!configFile) {
            fprintf(stderr, "You must specify "
                  "the config file\n");
            exit(1);
      }
      if (DC_initMaster(configFile) != DC_OK) {
            fprintf(stderr, "DC_initMaster failed, exiting.\n");
            DC_log(LOG_CRIT, "DC_initMaster failed, exiting.");
            exit(1);
      }
      return 0;
}
```

# 3 Work unit generation

The master should create and submit work units. To create work units you should use the *DC_createWU* function. The function has the following parameters:

- Application name
- Arguments
- Number of subresults
- A unique identifier tag for the WU

It returns a *DC_Workunit*, which might be used to register inputs/outputs with the WU.

We will split the original input file into smaller chunks. Each chunk contains a subset of the lines from the original file. Each work unit will process one chunk. We will use a constant to specify the number of lines per WU. This value will also specify the total number of WUs.

## *3.1 Declare a global variable to hold the current index*

```
#include <stdio.h>
#include <errno.h>
#include <dc.h>

#include "common.h"

#define LINES_PER_WU  20

int createdWUs;
```

### 3.2 Locate the createWork function

### 3.3 Generate the tag of the WU

```
while (!feof(input)) {
      char wuTag[16]; /* each WU has a unique tag */
      char buffer[1024]; /* a buffer to read the lines in */
      DC_Workunit *wu; /* WU struct to configure the WUs */
      FILE *wuInput; /* WU input file (chunk) */

      snprintf(wuTag, sizeof(wuTag), "%d", createdWUs + 1);
}
```

### 3.4 Generate the work unit

```
while (!feof(input)) {
      char wuTag[16]; /* each WU has a unique tag */
      char buffer[1024]; /* a buffer to read the lines in */
      DC_Workunit *wu; /* WU struct to configure the WUs */
      FILE *wuInput; /* WU input file (chunk) */

      snprintf(wuTag, sizeof(wuTag), "%d", createdWUs + 1);

      wu = DC_createWU("uppercase", NULL, 0, wuTag);
      if (!wu) {
            fprintf(stderr, "Work unit creation has failed.\n");
            DC_log(LOG_CRIT, "Work unit creation has failed.");
            exit(1);
      }
}
```

### *3.5 The WU input file is ready, register it with the WU (createWork)*

```
for (int i = 0; i < LINES_PER_WU; i++) {
      if (!fgets(buffer, sizeof(buffer), input)) { break; }
      fprintf(wuInput, "%s", buffer);
}
fclose(wuInput);

if (DC_addWUInput(wu, INPUT_LABEL, "wu-input.txt",
            DC_FILE_VOLATILE)) {
      fprintf(stderr, "Failed to register WU input file\n");
      DC_log(LOG_CRIT, "Failed to register WU input file");
      exit(1);
}
```

### *3.6 We should also register the output file with the WU*

```
if (DC_addWUInput(wu, INPUT_LABEL, "wu-input.txt",
            DC_FILE_VOLATILE)) {
      fprintf(stderr, "Failed to register WU input file\n");
      DC_log(LOG_CRIT, "Failed to register WU input file");
      exit(1);
}
if (DC_addWUOutput(wu, OUTPUT_LABEL)) {
      fprintf(stderr, "Failed to register WU output file\n");
      DC_log(LOG_CRIT, "Failed to register WU output file");
      exit(1);
}
```

# 4 Submitting the work unit

## *4.1 We should submit the WU into the DG with a call to the DC_submitWU function*

```
if (DC_addWUOutput(wu, OUTPUT_LABEL)) {
      fprintf(stderr, "Failed to register WU output file\n");
      DC_log(LOG_CRIT, "Failed to register WU output file");
      exit(1);
}

if (DC_submitWU(wu)) {
      fprintf(stderr, "Failed to submit WU\n");
      DC_log(LOG_CRIT, "Failed to submit WU");
      exit(1);
}
createdWUs++;
```

### 4.2 Invoke the createWork function from the main function

```
int main(int argc, char *argv[]) {
      …
      if (DC_initMaster("dcapi.conf") != DC_OK) {
            fprintf(stderr, "DC_initMaster failed, exiting.\n");
            DC_log(LOG_CRIT, "DC_initMaster failed, exiting.");
            exit(1);
      }

      createWork();

      return 0;
}
```

# 5 Setting up callbacks

Now that we have submitted work units into the DG, the clients can start working. The incoming results should be processed to produce a meaningful output. Using the DC-API it is done by setting up callback functions. To process the results, you should set up a result callback function.

Callback functions can be implemented to handle various tasks:

- Result processing
- Subresult processing
- Message processing

These functions should be registered with the DC-API. They can be set individually (*DC_setResultCb*, *DC_setSubresultCb*, *DC_setMessageCb*) or simultaneously (*DC_setMasterCb*).

### 5.1 Register our result callback function with the DC-API

```
int main(int argc, char *argv[]) {
      …
      if (DC_initMaster(configFile) != DC_OK) {
            fprintf(stderr, "DC_initMaster failed, exiting.\n");
            DC_log(LOG_CRIT, "DC_initMaster failed, exiting.");
            exit(1);
      }

      DC_setResultCb(processResult);
      createWork();

      return 0;
}
```

## 6 Processing events

In its main loop the master application should periodically call the *DC_processMasterEvents* function. The previously set callback functions will be invoked this way.

### *6.1 Declare a global variable to keep track of the number of already processed WUs*

```
#include <stdio.h>
#include <errno.h>
#include <dc.h>

#include "common.h"

#define LINES_PER_WU  20

int createdWUs;
int processedWUs;
```

### *6.2 Set up a loop in the main function to check for events until all of the results arrive*

```
int main(int argc, char *argv[]) {
      …

      DC_setResultCb(processResult);
      createWork();

      while (processedWUs < createdWUs) {
            DC_processMasterEvents(60);
      }

      return 0;
}
```

# 7 Processing the results

Previously we registered a function (*processResult*) as a callback. The next task is to implement this function. This function will simply make copy of the output file of the WU. The result callback function should have the following prototype:

**void (*DC_ResultCallback) (DC_Workunit *wu, DC_Result *result);**

*DC_processMasterEvents* calls the callback function.

Tasks:

- Find out which WU has finished (*DC_getWUTag*)
- Process the outputs (*DC_getResultOutput*)
- Destroy the WU (*DC_destroyWU*)

## *7.1 Increment the processedWUs variable and extract the tag of the WU*

```
void processResult(DC_Workunit *wu, DC_Result *result) {
      char *output; /* the output file name of the WU */
      char *tag; /* the tag of the WU */
      char cmd[256]; /* the copy command */

      processedWUs++;
      tag = DC_getWUTag(wu);
}
```

### 7.2 Make sure the received result is not failed

```
void processResult(DC_Workunit *wu, DC_Result *result) {
      char *output; /* the output file name of the WU */
      char *tag; /* the tag of the WU */
      char cmd[256]; /* the copy command */

      processedWUs++;
      tag = DC_getWUTag(wu);

      if (!result) {
            fprintf(stderr, "Work unit %s has been failed\n",
                  tag);
            DC_log(LOG_WARNING, "Work unit %s has been failed",
                  tag);
            free(tag);
            return;
      }
}
```

### 7.3 Get the name of the result output file

```
if (!result) {
      fprintf(stderr, "Work unit %s has been failed\n", tag);
      DC_log(LOG_WARNING, "Work unit %s has been failed", tag);
      free(tag);
      return;
}

output = DC_getResultOutput(result, OUTPUT_LABEL);
if (!output) {
      fprintf(stderr, "Work unit %s contains no output file\n",
            tag);
      DC_log(LOG_WARNING, "Work unit %s "
            "contains no output file", tag);
      free(tag);
      return;
}
```

### 7.4 Clean up

```
snprintf(cmd, sizeof(cmd), "/bin/cp '%s' 'result_%s.txt'", output,
       tag);
system(cmd);

DC_log(LOG_NOTICE, "Work unit %s has been completed", tag);
DC_destroyWU(wu);

free(output);
free(tag);
```

# 8 Creating the final result

The last thing to do is to produce the final result. No DC-API specific function calls are required to do that. The function has been already defined in the skeleton file. You do not have to define it, just insert a call to it at the end of the main function.

## *8.1 Insert a call to the calculateFinalResult function at the end of the main function*

```
while (processedWUs < createdWUs) {
      DC_processMasterEvents(60);
}

createFinalResult();

return 0;
```

# 9 Compiling the master

You will find a ready to use make file in the tutorial directory. In order to compile DC-API master applications, you have to pass some additional libraries to the linker. You can obtain these library flags with the following command:

**pkg-config -libs dcapi-boinc-master**

You do not have to change the make file, it has been prepared for you.

## *9.1 Compile the master with the supplied make file*

```
make master
```

## *9.2 If you have done everything right, the master will compile and a binary with the same name will come into existence*

## 10 Cheating

The completed *master.cpp* can be found in the same directory named *master_final.cpp*.

```
mv master.cpp master_my.cpp
cp master_final.cpp master.cpp
```