DICElib User's Guide

DIstributed
CAVE
Engine
Library

Bruno Barberi Gnecco *brunobg@lsi.usp.br*

16th December 2001

Contents

1 Introduction **3**

1.1 License . . . 3
1.2 Requirements . . . 3
1.3 Portability . . . 3
1.4 Where to get . . . 4
1.5 Examples . . . 4

2 Basics **5**

2.1 Initializing and closing . . . 5
2.2 dicerc file . . . 6
2.3 Types . . . 6

3 Synchrony **11**

3.1 Synchronizing clients . . . 11
3.2 Synchronous variables . . . 11
3.3 Pitfalls . . . 12

4 Asynchronous variables **15**

5 Miscellaneous **17**

5.1 Spawning . . . 17
5.2 DICE_function . . . 18

6 Adapting your applications to use DICElib **19**

6.1 Porting 3D apps to multiview systems . . . 19

7 Performance **21**

8 Internal protocol **23**

9 FAQ **25**

9.1 Why don't you use libdl/libltdl? . . . 25
9.2 The compiler complains that there's an "undefined reference to 'DICE_MyFunctions"'! . . . 25

9.3 The compiler complains that there's an "undefined reference to 'DICE_MyTypes"'! 26
9.4 The program segfaults with no apparent reason. Any ideas? . . . 26
9.5 What about these float754 and double754 types? 26
9.6 I can't run X applications! I get the following error: "GLUT: Fatal Error in (unamed): could not open display:" 27
9.7 The windows of my GLUT application are not updated (or only one of them is updated). 27

Chapter 1

Introduction

DICElib is an effort to ease the use of clusters to drive applications that need constant synchrony, such as graphical applications, CAVE^{tm} systems, etc. The library's main features are:

- synchronization;
- shared memory, which can be synchronous or asynchronous;
- transparent functions (same function may do different things in different clients);
- implemented with sockets, uses TCP/IP.

This user's guide is almost a tutorial and introduces DICElib. It is not intended to be a reference guide; if you are looking for reference, try the automatically generated docs in the `doc/` directory.

1.1 License

DICElib is released under the GNU Public License.

1.2 Requirements

If you have the file ieee754.h in your computer, you'll be able to exchange floating point numbers without loss of precision. This file is present in Linux.

1.3 Portability

DICElib was developed in a Linux/x86 platform, which is what most of the clusters run, anyway. It should run in any Unix that fulfills the requirements (see section 1.2). I have successfully compiled and ran it in a SGI/Irix machine, and under Cygwin. The library doesn't care about the underlying OS or the machine, so you may run clients in different architectures and they will work together.

If you have ieee754.h in your system, floats and doubles will be transferred portably, without loss of precision (use the *float754* and *double754* types; see section 9.5).

1.4 Where to get

DICElib's official homepage is:

`http://www.lsi.usp.br/~.....`

You can contact the authors:

Bruno Barberi Gnecco brunobg@lsi.usp.br

1.5 Examples

In the `tests/` subdirectory you will find several small programs. They are a good source for learning DICElib, and some of them are used in this guide. You'll find there a README with descriptions of each program. DICElib's homepage has links to applications that use DICElib.

Chapter 2

Basics

This chapter presents the basic functionality of DICElib.

2.1 Initializing and closing

To use DICElib functions, include:

```
#include <dicelib.h>
```

Before any other commands, you must call `DICE_init`:

```
int DICE_init  ( int *argc, char **argv );
```

refer to the reference guide for supported arguments. The function cleans any arguments that it uses from the list, so you should call it before parsing your program's own arguments. All DICElib functions that return an integer follow this standard:

0	OK.
negative	Error.
positive	A value that you are probably interested in.

When your application quits, call:

```
void DICE_close  ( void );
```

which automatically quits every other client and the server.

To get and set attributes:

```
int DICE_set_attribute ( DICE_AttributeType t, void *value );
void * DICE_get_attribute ( DICE_AttributeType t );
```

Again, the reference guide has a complete, up-to-date list of the attributes supported.

2.2 dicerc file

Once started, the program parses arguments, and then tries to read a file named `dicerc`. This file is searched in the following directories, in this order:

- ./dicerc
- ~/dicerc
- /etc/dicerc

This file should contain a list of the computers that should run a copy of the program. The format of each entry is the following:

```
[machine name or IP] [path of program] [special function name]
```

Where:

[machine name or IP] Name or IP of the machine to spawn a process.

[path of program] Full path of the program in that machine. Use % if it's the same of the machine spawning.

[special function name] Name of a special function to be run in that machine (see section 5.2).

Fields are separated by any number of spaces or tabs. Each line must contain only one entry. Blank lines or lines starting with # are ignored. You may ignore either the two last or just the last field. The same computer may appear many times in the list, and the same number of processes will be spawned there. Spawning is covered in section 5.1.

A sample dicerc:

```
machine1 % function1
machine2 /usr/bin/programname function2
machine3 /usr/local/bin/programname
machine1
```

The hierarchy of the clients is set by their order in dicerc: machines closer to the top of the file have a higher priority. The first node on the file is called *original*, and has higher priority than any other node.

2.3 Types

A *type* in DICElib is a variable type. To share variables between computers, DICElib has to store them internally in such a way that it may be sent to the other nodes.

Types are registed automatically when you start your program. Once registered, a type will exist until DICElib is closed. Types consume a very small amount of memory, and you may register as many as you like. Note that types are case sensitive.

If you want to define a new type, you must write three functions:

```
char *foo_pack (void *data, int *n);
void *foo_unpack (char *data, int *n);
void foo_free (void *data);
```

if any of the three is not available, the type won't be registered.

Now, what do these functions do?

foo_pack receives the data of the type you registered, cast to void *, and a pointer to an integer. Its function is to return a string, which you made from the data, and to store the string size in the pointer.

foo_unpack receives a string that was packed by foo_pack, and a pointer to an integer. Its function is to parse the string and convert it to the variable type foo, returning this value, and to return the number of bytes that were used of the original string. Remember: there may be more data in the string that is part of another variable. The only guarantee you have is that the first character of the string is also the first character of your packed data.

foo_free frees the foo type. It's useful for composed types, such as structures, etc.

You can access these three functions indirectly, using respectively:

```
char *DICE_type_pack ( char *type, void *data, int *n );
void *DICE_type_unpack ( char *type, char *packet, int *n );
int DICE_type_free ( char *type, void *data );
```

They are work exactly like the above description, but receive the type as an extra argument.

DICElib provides some default types[1]. All of them receive pointers to the data, so pass by reference:

unsigned_char Unsigned char. Supposed to be a one-byte, unsigned integer.

[1] Note about portability: as of current version, DICElib assumes that the C types follow the specifications below, and takes care of endianess (all implemented types follow RFC1014). This is not too bad, since 32 bit computers usually follow these specifications, but your mileage may vary. If the integer type sizes are bigger than the described here, they are likely to work for the unsigned types. While I believe that most current architectures follow the IEEE754 standard for floating point, there are always exceptions. If you have ieee754.h in your computer, all is well; otherwise, try to hack it. Future versions will try to do this automatically.

signed_char Signed char. Supposed to be a one-byte, signed integer.

unsigned_short Unsigned short. Supposed to be a two-byte, unsigned integer.

signed_short Signed short. Supposed to be a two-byte, signed integer.

unsigned_long Unsigned long. Supposed to be a four-byte, unsigned integer.

signed_long Signed long. Supposed to be a four-byte, signed integer.

float Floating number, 6 digits of precision (uses 6 bytes).

double Floating number, 16 digits of precision (uses 12 bytes).

float754 IEEE 754 float (4 bytes). No loss of precision.

double754 IEEE 754 float (8 bytes). No loss of precision.

string A '\0' terminated string of `char`s.

Last, but not least, in order for DICElib to find your new types automatically, you must define the `DICE_MyTypes` variable:

```
DICE_Type DICE_MyTypes[] = {

    { "foo type", foo_pack, foo_unpack, foo_free },
    { NULL, NULL, NULL, NULL }
};
```

It's a simple array of structures, and each structure should get (in this order): the type name, the pack functions, the unpack function, and the free function. *Important*: the last element of the array *must* be `{ NULL, NULL, NULL, NULL }`.

If you don't create any new types, you *have* to add this line to your code, declaring it as you would declare a global variable:

```
DICE_no_types;
```

Since I'm sure you are completely lost, it's time for an example. Suppose your application works with 3D floating points vectors, like:

```
float vec[3];
```

and you want to share this type. You may consider it as 3 floats, etc, but it's easier to work it as a whole thing. So, here are the three functions:

```
char *float3_pack ( void *data, int *n ) {
```

```
    char   *buf, *t;
    float   *vec;
    int   m;
    /* reconvert the data to float, so we can work with it */
    vec = (float *)data;
    /* allocate memory for the packet *//
    buf = (char *)malloc(3*4);
    if ( !buf )
       return NULL;
    /* now pack it using DICElib's floats */
    t = DICE_type_pack("float", (void *)&vec[0], &m);
    memcpy(buf, t, 4);
    free(t);
    t = DICE_type_pack("float", (void *)&vec[1], &m);
    memcpy(buf+4, t, 4);
    free(t);
    t = DICE_type_pack("float", (void *)&vec[2], &m);
    memcpy(buf+8, t, 4);
    free(t);
    *n = 12;
    return buf;
}
void *float3_unpack ( char *data, int *n ) {
    float   *vec, *f;
    int   m;
    /* malloc memory for the float */
    vec = (float *)malloc(3*sizeof(float));
    /* three floats */
    if ( !vec )
       return NULL;
    f = (float *)DICE_type_unpack("float", data, &m);
    vec[0] = *f;
    free(f);
    f = (float *)DICE_type_unpack("float", data+4, &m);
    vec[1] = *f; free(f);
    f = (float *)DICE_type_unpack("float", data+8, &m);
    vec[2] = *f;
    free(f);
    *n = 12;
    return (void *)vec;
}
void float3_free  ( void *data ) {
    if ( data ) free(data);
}
```

```
DICE_Type DICE_MyTypes[] = {

    { "float3", float3_pack, float3_unpack, float3_free },
    { NULL, NULL, NULL, NULL }
};
```

The new type name is `float3`. The pack and unpack functions use the existing float type to send it. This is a simple example, and it supposes that the float type is 4 bytes long, etc.

Take a look at the `tests/` subdirectory, in special the `var_test` and `rotate` programs, for more examples.

Declaration of variables is dealt in the next chapters. Take notice that, while it's practical to setup arrays as a type, the entire array is transferred even if just one variable is changed, so it might not be the wisest solution.

Chapter 3

Synchrony

3.1 Synchronizing clients

Synchronization is quite easy to perform. At the point of the code that you want to synchronize all processes, call:

```
int DICE_sync ( void );
```

When a client calls DICE_sync, it communicates with the server, updating synchronous variables (see sections 3.2 and 8), and blocks, waiting for a message. Blocking is done with `read(2)`, so you may need to take care in threaded programs.

Once *all* clients call `DICE_sync`, the server broadcasts a message, and they all unblock.

3.2 Synchronous variables

Synchronous variables are updated only when a call to `DICE_sync()` is made.

You can create them using:

```
int DICE_sync_new ( char *name, char *type, void *value );
```

where `name` is the variable name (case sensitive, any string), `type` is the variable type (see section 2.3; if the type was not registered, it's automatically done so), and `value` is the variable value. Note that the variable will only be available after you call `DICE_sync()`. You can only free or alter the value after you call DICE_sync[1], since only then DICElib makes a copy of it.

You can delete the variable using:

```
int DICE_sync_delete ( char *name );
```

[1]This limitation is likely to change in future versions.

the function cares to free the variable value.

To update variables with a new value, call:

```
int DICE_sync_update ( char *name, void *value );
```

Remember, the update will be scheduled and done only after the next call to `DICE_sync()`. The previous value is automatically freed once the update is made.

```
void *DICE_sync_get ( char *name );
```

This function returns the variable's value, cast to `void *`. You must always use this function to access the new value after calling `DICE_sync()`, since the variable value may have changed.

```
void *DICE_sync_get_next ( char *name );
```

This function returns the value that is scheduled for update, or NULL if none.

3.3 Pitfalls

There are some considerations to be taken when using synchronous variables.

Every time you call `DICE_sync`, you should update your variables with `DICE_sync_update`. That is because, if some variable value changed, the variable itself was freed and reallocated, so your pointers are likely to be outdated. The reason for this apparently stupid behavior is that some types have variable sizes (such as strings), so sometimes the pointer would have to be updated. To avoid clumsiness (this type needs to be updated, this one doesn't), it was decided to make this a standard. Future versions may improve this *internally*, avoiding the reallocation iff explicitly declared. You can solve this problem easily, and avoid any kind of trouble, declaring your own sync function, and letting it take care of updates, like in this example:

```
extern FooType *f;

int my_sync ( void ) {
    int i;
    DICE_sync_update("foo", f);
    i = DICE_sync();
    if (i)
        return i;
    f = DICE_sync_get("foo");
}
```

So whenever you want to synchronize you call `my_sync`, which does all the dirty job for you. See the `tests/` directory for more information.

You also have to take care in threaded programs, but you always have to take care when you write a threaded program anyway. I believe that you just need to pay attention to two things:

1. call `DICE_init` before you fork the threads. Since `DICE_init` will fork when spawning clients, it's wise not to mix forking and threading.

2. when `DICE_sync` blocks, waiting for a reply from the server to continue, it does so using `read(2)`. This is a quote from the Linux glibc info pages:

   > This function [`read(2)`] is a cancelation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time 'read' is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this calls to 'read' should be protected using cancelation handlers.

So that's another reason to have a `my_sync` function.

Threaded programs may also have problems with asynchronous functions, so be sure to read section 4 too.

Chapter 4

Asynchronous variables

Not working yet; only partially implemented.

Chapter 5

Miscellaneous

5.1 Spawning

Client spawning can be done by DICElib. There are currently three ways of spawning:

- rsh
- ssh
- bproc[1]

You should use the one that suits you best. The first two require a copy of the program to be available in the clients (but may be in other path; see section 2.2), while the latter doesn't, but requires BProc to be installed.

If you'd rather spawn the processes yourself, you can do it too. First of all, you still need the dicerc file, which is used to gather information such as the special functions, number of clients, etc. Then run your program in the control:

```
[program] --dice-spawn none
```

And in the clients:

```
[program] --dice-client-only --dice-control-is [control address]
```

You can easily implement a script to spawn your program in a way that fits your setup. Take a look at `tests/spawner`, a small script that spawns processes in localhost.

[1] Bproc is not supported yet.

5.2 DICE_function

Sometimes is desirable to run a different function in each client, such as in CAVE systems, where each node will render a different view. DICElib provides this functionality, letting each machine have a special function, which can be accessed by calling:

```
int DICE_function(void *data);
```

This special function is a pointer to the function you provided in the dicerc file (see section 2.2), and is automatically setup upon spawning. If no special function was set and you call `DICE_function`, it will return -1. See the tests/around.c file for an example of how special functions work.

DICElib needs a little help from you to find the functions. You *must* declare the following variable:

```
DICE_Function DICE_MyFunctions[] = {
    { "func1_name", func1 },
    { "func2_name", func2 },
    { NULL, NULL }
};
```

You may provide any number of functions, whether they are used or not by the dicerc file. *Important*: always finish the array with `{ NULL, NULL }`. Failure to do so will lead to a segmentation fault. If you don't need DICE_function, declare:

```
DICE_no_functions;
```

as if it were a global variable.

See the `tests/` subdirectory for some examples.

Chapter 6

Adapting your applications to use DICElib

In computers, nothing is ever created, nor ever destroyed, everything is cop... patched.

6.1 Porting 3D apps to multiview systems

Most 3D applications can be easily ported to run in multiview environments.

There are only two requirements:

1. You must be able to render the view in any direction;
2. You must have variable with the viewer's position and orientation.

Not that hard, huh? Some applications that use hacks (such as direct 2D drawing, magics in the z-buffer, etc) may be harder to port. In short, if you can render the view in any direction you want, then it's very easy to port.

Below is a step by step procedure that is likely to work in virtually any 3D application.

1. Define the special functions. They are responsible for setting the view in each client. For example, for setting orthogonal views:

   ```
   int front (void *data) {
       return 0;
   }
   int left (void *data) {
       glRotated(90, 0, -1, 0);
       return 0;
   }
   int right (void *data) {
   ```

```
    glRotated(90, 0, 1, 0);
    return 0;
}
int back (void *data) {
    glRotated(180, 0, -1, 0);
    return 0;
}DICE_Function DICE_MyFunctions[] = {
{ "front", front }, { "left", left }, { "right", right }, { "back", back }, { NULL, NULL } };
```

2. Add DICE_init()/DICE_close() calls. The DICE_init() call should be one of the first calls in your program. The sooner, the better. DICE_close() should be called everywhere your application may exit, ensuring that all clients and the server are automatically closed.

3. Declare your global variables as shared variables. In general, you'll want to share: position, viewing direction, viewing options (such as wireframe/solid, textures on/off). I suggest that physic simulation is handled only in one client (original). It's quite easy to do it: just add this code to the beginning of every function you want to run in only one client:

```
if (!DICE_is_original())
    return;
```

4. I suggest that you write a mysync() function, to take care of the updates/synchronization of the shared variables whenever you synchronize. Call it instead of DICE_sync(). Something like:

```
int my_sync ( void ) {
    int i;
    DICE_sync_update("axisRot", &axisRot);
    DICE_sync_update("rotate", &rotate);
    i = DICE_sync();
    if ( i < 0 )
        return i;
    axisRot = *((int *)DICE_sync_get("axisRot"));
    rotate = *((float *)DICE_sync_get("rotate"));
}
```

That's it. It usually takes about five minutes to have a simple application converted.

Chapter 7

Performance

DICElib was developed to synchronize video frames in a graphical cluster, so it was expected to have a high performance. Tests realized with it show that it's more than adequate for this use.

The test was ran in two Dual Pentium III 933MHz, 1GB RAM, optical Gigabit-Ethernet. Given a vector of integers (4 bytes), update the vector and synchronize the applications ten thousand times. We ran this test for several different vector sizes. The results proved that the time spent linear with the number of variables, proving that there's no overhead for large packets, and the time depends only of the amount of data to be transferred.

The results are quite good: even transferring 4000 variables (that is, 16kb of data) per frame, we achieved 550.1 updates per second. It is more than enough for video synchrony, and most virtual reality applications are likely to transfer only a few bytes of data, containing information such as position, orientation, etc. The highest update frequency was for 250 variables (which was the lowest number ran), with an average of 5988.0 updates per second. The bandwidth taken is small: only 8.8Mb/s. The limitation seems to be imposed by TCP/IP buffering constraints. The program used is in tests/bench.c.

Small applications, such as test/around.c and test/rotate.c were run in two and three nodes, and there was no noticeable difference in the synchrony, independent of the frame rate.

Chapter 8

Internal protocol

Here's a description of the protocol used by DICElib.

DICElib uses (internally) a server-client system, which saves network resources and avoid conflicts between the client's data. The server (which we call *control*) is a very light program, which basically works only as a bridge and a referee between the clients; you don't have to know about it, except that it runs in the computer you start your DICElib program.

Everything is done using TCP/IP.

[more to come]

Chapter 9

FAQ

9.1 Why don't you use libdl/libltdl?

Versions prior to 1.0 used libdl/libltdl, and that made types and functions much easier to use, since you didn't have to define DICE_Functions and DICE_Types. Types were registered only when a variable of that type was created, saving memory.

The problem is that libltdl is not 100% portable. Although I successfully tested in Linux x86 and IRIX, Windows needed Cygwin to run. To tell you the truth, I didn't care too much, but other people did. ;-) So, I decided to get rid of libltdl by default, leading to a more cumbersome library. I believe that Windows LoadLibrary functions could be used instead.

If you don't need to run your programs in Windows and would like to use libltdl, run "configure –with-ltdl"[1]. Here are the differences:

- DICE_Functions and DICE_Types variables are ignored.
- Types are automatically registered when you declare a new variable (with DICE_sync_new or DICE_async_new). The caveat is that, if your type is *foo*, the functions must be called *foo_pack*, *foo_unpack* and *foo_free*.
- Functions are automatically found, you don't have to do anything else.

In theory, programs compiled with libltdl should be able to talk with programs compiled without it.

9.2 The compiler complains that there's an "undefined reference to 'DICE_MyFunctions"'!

If you are not using `DICE_function`(), then add:

[1] Don't use ltdl support, really. The code will probably be deleted soon.

```
DICE_no_functions;
```

as if it were a global variable.

If you are using `DICE_function()`, see section 5.2.

9.3 The compiler complains that there's an "undefined reference to 'DICE_MyTypes"'!

If you did not create any new variable types, then add:

```
DICE_no_types;
```

as if it were a global variable.

If you defined new types, see section 2.3.

9.4 The program segfaults with no apparent reason. Any ideas?

Did you define `DICE_MyFunctions` or `DICE_MyTypes`? If you did, make sure that the last elements of the array are, respectively, { NULL, NULL } and { NULL, NULL, NULL, NULL }

9.5 What about these float754 and double754 types?

IEEE 754 is an international standard for representing floating point numbers. Many of today's computers follow the norm, but not always *exactly*. You know that standards are so good that you should have many of them.

In short, if your computer either follows IEEE 754 or you have someway to convert a 754 to your internal type, you can exchange floating point numbers without loss of precision. Unfortunatedly the only OS that I know that has float754.h is Linux, so if you are running another OS you may have to do your own tweaking. Let me know if you manage to use these types in some platform other than Linux/x86.

Floatint point exchange has been a problem since it was first implemented, but things have been improving lately. I hope someday people will agree that exchanging floats is important, and that everybody should follow the standard.

The usual solution for this problem is to 'printf' the number and send as a string. This ensures 100% portability, at the cost of precision and bandwidth. DICElib supports this solution, using a special encoding to cut bandwidth. The float type has 6 digits of precision, and the double has 16, which should be enough for anybody sane. The issue is that float754 takes 4 bytes against 6 of float, and double754 uses 8 against 12.

I did one million transfers using random numbers. The maximum error was $5,35442.10^{-7}$ for the float type, and less than 10^{-15} for the double. So, unless you really need the bandwidth, you'll be fine with the non-754 types.

9.6 I can't run X applications! I get the following error: "GLUT: Fatal Error in (unamed): could not open display:"

The programs are being spawned, but since the DISPLAY environment variable is not set, they can't open an window. When you run X programs, add '`--dice-environment DISPLAY=:0.0`' as an extra argument (or change DISPLAY's value to suit your setup). This should solve it. If you still have problems, avoid the automatic spawning, and write a small script to run the copies, such as the `spawner` script in the tests/ subdirectory.

9.7 The windows of my GLUT application are not updated (or only one of them is updated).

What's happening is simple: only your original window is calling glutPostRedisplay(). A simple solution is to make sure that the `glutIdleFunc` callback is calling your display function.