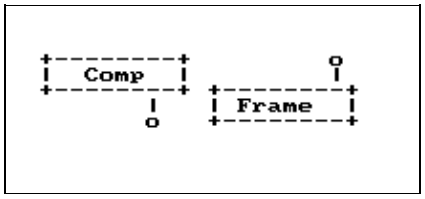


CompFrame

the component framework ... that keeps it simple

Table of Contents

CompFrame

Author:
Peter R. Torpman (ptorpman at sourcefourge.net)

Version:
0.2

Date:
2005−06−02

Note:
CompFrame is written in C. The standard used is the GNU implementation of C99.

0 Terms and Abbreviations

Instance − An occurence of a component (class).

Interface − A set of functions used to manipulate the state of a component, or, to tell the component to perform a specific task.

Class − A set of attributes and functions to manipulate those attributes.

Component − A part of the component framework, e.g a class, interface etc.

Component library − A set of components grouped together, e.g in a shared object library file.

1 Background

Having worked in several projects that made use of some component framework, I came to see that it would really be beneficial for the OpenSource community to have a component framework. A component framework that would initially contain my experiences and, hopefully, not contain the pitfalls that I run into in

the past with other component frameworks. In short, a straight−forward, easy to use piece of software that would empower developers instead of restraining them.

I also happen to think that component based softwares are the right way to construct softwares. Why?! Because if you do it right, you can keep functionality separated and specialized. Furthermore, the maintenance will be easier. Also, functionality can be added easily in separate containers instead of patching some obscurely and loosly ordered source code files.

My idea with CompFrame is that it can be used, extended and improved over time, without the need for re−inventing the wheel (or paying lots of cash) every time a component based software is supposed to be assembled. Thus, saving money and giving the poor developer time to address what's really important (and interesting).

2 Introduction

Note:

In CompFrame the textual name of a component must be unique. And, in CompFrame the textual name of an interface must be unique.

2.1 Component Class

A component class is realized using structs. All values needed by the component should be kept within this struct. Do not use static variables in the source code files, because they will be open to all instances of the component.

The example below depicts a component class with two attributes − one for the instance name and one for the interface implemented by the component 'iface_test2'.

```
typedef struct TestComp
{
  char*         instance_name:
  iface_test2*  test_iface2;
} TestComp;
```

2.2 Interface

A CompFrame interface needs two things − a unique textual name and a unique C type.

A component that wants to implement an interface must implement the functions implicated by the function pointers declared in the interface type. For example, a component that would like to implement the interface in the example below, would have to supply an implementation for the 'printHelloWorld' function.

```
#define TEST2_IFACE_NAME "TEST2"

typedef struct iface_test2
{
  void (*printHelloWorld)(void);
} iface_test2;
```

2.3 Component Library

A CompFrame component library is a shared object library (so−file), that may contain the implementation of one or many components.

The library must have the following function defined:

```
void dlopen_this(void)
```

The library file will upon opening be scanned for the **dlopen_this()** function. If the function cannot be found, it is not a valid CompFrame component library.

2.4 Component Registration

A CompFrame component must register itself into the **Registry** of CompFrame before it can be used.

Registration can be done in the **dlopen_this()** function.

```
cf_component_register("COMPNAME", create_me, set_me_up);
```

In this example the **COMPNAME** component is registered. The *create_me()* function will be called when a user wants to create an instance of the component. *set_me_up()* will be called after the instance has been created, in order to let it set itself up and get ready for business.

The library file will upon opening be scanned for the *dlopen_this* function. If the function cannot be found, it is not a valid CompFrame component library.

2.5 Interface Registration

A component can implement no or many interfaces. Any interfaces implemented by the component must be registered to the **Registry**.

```
static iface_test2 iTest2;

iTest2.printHelloWorld = myHelloWorld;

CfIfaceToReg iface[] = {
  {TEST2_IFACE_NAME, &amp;iTest2},
  {NULL,NULL}
}

cf_interface_register(comp,iface);
```

In the example above, the component registers that it implements the *iface_test2* interface. And, it does so by the *myHelloWorld* function.

2.6 Command Registration

Commands that can do all sorts of things, may be registered into CompFrame and made available on the

CompFrame command line interface (CLI).

The **C** system component is the Command Handler of CompFrame, and it is in that component we need to register our commands. That is done in the following way:

```
static int
my_cmd(int argc,char** argv);

void* cObj    = cf_component_get(CF_C_NAME);
cfi_c* cIface = cf_interface_get(cObj, CF_C_IFACE_NAME);

cIface->add(cObj, "create", create_cmd, "Usage: create <class> <name>");
```

In the example the command *create* is registered. The *create_cmd()* is the function that is to perform all the work. Also, a help string is provided.

3 Command Line

The command line of CompFrame is currently like this:

```
compframe -d <componentdir>
          [-c <configuration>]
          [-t <level>]
```

-d is used to point out the directory where the component libraries are located.

-c is used to point out the configuration file.

-t is used to specify trace level (0-3)

3.1 create

This command is used to create instances of components.

```
CompFrame 0.2 (c)2005 Peter R. Torpman
>> create TESTCOMP x1
```

An instance of TESTCOMP with the name 'x1' is created.

3.2 list

This command is used to list many things, such as registered classes, implemented interfaces etc.

```
CompFrame 0.2 (c)2005 Peter R. Torpman
>> list -c
```

The registered classes will be printed.

```
CompFrame 0.2 (c)2005 Peter R. Torpman
>> list −i x1
```

Lists the implemented interfaces of 'x1'.

3.3 connect

This command is used to connect instances of components.

```
CompFrame 0.2 (c)2005 Peter R. Torpman
>> connect x1 x2 TEST 1 2 "hello"
```

Connects instance 'x1' to instance 'x2' on its TEST interface, passing 1, 2 and "hello" as parameters.

3.4 help

Will print the help text supplied to the 'create' command.

```
CompFrame 0.2 (c)2005 Peter R. Torpman
>> help create
```

Shows the help text for the create command.

4 S − Scheduler

The thought of a "common time" for all components is nice. A time that is independent of the wall clock time. A virtual time, that might mean that one component gets to execute 300 real seconds per cycle, while an other just uses up 10 real seconds. But, they both think they executed 1 second, or whatever the cycle time is set to.

S, the CompFrame scheduler, makes sure that all components *that chooses to be scheduled*, are scheduled in a round−robin type of fashion.

A component that wishes to be scheduled, must implement the *cfi_s_client* interface. interface:

```
static cfi_s_client sClient;

static void
set_me_up(void* comp)
{
  ::
  // Scheduler client interface
  sClient.execute = s_client_execute;

  // Register our interfaces
  CfIfaceToReg iface[] = {
    {CF_S_CLIENT_IFACE_NAME, &sClient},
    {NULL,NULL}
  };

  cf_interface_register(comp, iface);

  // Add ourself to the S scheduling loop
```

```
  void* sObj = cf_component_get(CF_S_NAME);

  cfi_s_server* sIface = cf_interface_get(sObj, CF_S_SERVER_IFACE_NAME);
  sIface->add(sObj,this);

  ::
}
```

After that registration, we will be called int the *s_client_execute* function, when **S** decides that it is our turn to execute. In that function, we can do something like this:

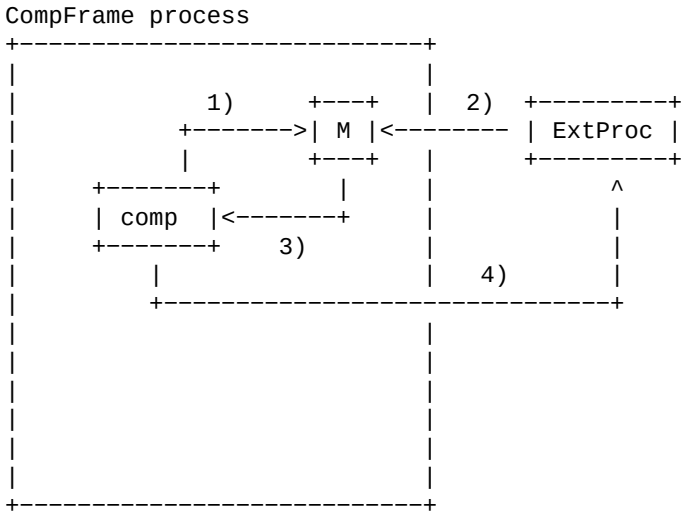
```
static void
s_client_execute(void* obj, uint32_t slice, int* sliceRemain )
{
  // do some stuff

  // Inform S that we have used up all our time
  *sliceRemain = 0;
}
```

If we do not set *sliceRemain* to zero, we will get the remainder of the slice, plus the ordinary time slice, when we are called upon again.

5 M − Message Handler

M, is the Message Handler of CompFrame. It is used to establish connections to components that reside within CompFrame, from other applications that are somewhere else.



In the image above, the basics are shown for how a connection is established between a component within CompFrame and an external program. First, in 1) the component registers itself within **M**. When doing that, the component passes parameters to make itself unique − a UUID and a name. The name does not make this,

so−called *message receiver* unique, it is the UUID and name together that makes it unique.

Second, the external program connects to **M** by using the mechanisms provided by the *M client library* (see below). The program specifies which message receiver and UUID it is interested in communicating with.

M then looks up the registered message receiver and informs it that an external program is interested in some communication.

After all this, in 4) the communication is established between the two entities.

M opens up a server socket during CompFrame initialization, that is used by the *M client library*. You can get a hold of this port by looking for something like this:

```
*** INFO # M server started on hammer:35545
CompFrame 0.2 (c)2005 Peter R. Torpman
>>
```

Or, you can get the port by entering the following command on the CLI:

```
CompFrame 0.2 (c)2005 Peter R. Torpman
>>  m −l
M server located at hammer:35545
```

Connections and Channels

M uses the concept of *connections* and *channels*. By definition, a connection is a socket connection between a CompFrame internal component and an external entity. And, a channel is a logical path on that connection. **M** allows for 255 channels for each connection.

Protocols

A protocol, in the world of **M**, is just a byte stream where the bytes have been given a specific meaning. Nothing revolutionary at all. In a protocol, over a channel between A and B, the bytes could mean one thing in one direction, and something completely different in the other.

For example, this could be a protocol description (yes, you need it for your own good):

```
+------+--------+--------+----------+---+
| WHAT | LEN LB | LEN HB | PAYLOAD  | 0 |
+------+--------+--------+----------+---+
WHAT    − 1 byte  − Content selector. (What is carried..)
LEN LB  − 1 byte  − Total length low byte of payload
LEN HB  − 1 byte  − Total length high byte of payload
PAYLOAD − n bytes − The payload of info.
```

And, this could be another:

```
+----------+---+
| PAYLOAD  | 0 |
+----------+---+
PAYLOAD − n bytes (NULL terminated) − Good information.
```

In short, you decide, the important thing is that sender and receiver agrees on what the different bytes of the protocol means.

Interface

A component that wants to provide a *message receiver* needs to implement *cfi_m_client* inteface.

```
// A sample protocol identifier...
#define TEST_UUID "433e76d0-77d1-460d-9321-e2dc8dc8bd59"

static void
set_me_up(void* comp)
{
  ::
  // Set up implementation of M client interface
  mifClient.connected    = m_cli_connected;
  mifClient.disconnected = m_cli_disconnected;
  mifClient.message      = m_cli_message;

  CfIfaceToReg iface[] = {
    {CF_M_CLIENT_IFACE_NAME, &mifClient},
    {NULL,NULL}
  };

  // Register interface
  cf_interface_register(comp, iface);

  // Store reference to M
  void* mObj = cf_component_get(CF_M_NAME);

  // Get server interface of M
  cfi_m_server* m = cf_interface_get(mObj, CF_M_SERVER_IFACE_NAME);

  // Add our message receiver
  m->mr_add(mObj,this,TEST_UUID,"sample1",NULL);

  ::
}
```

In this example, the component registers the *sample1* message receiver,that implements the TEST_UUID protocol. After this, the component is reachable from the outside world.

When a connection is established, the component will be called in the *m_cli_connected* function.

```
static int
m_cli_connected(void* comp, CfMConn* conn, uint8_t chan,void* userData)
{
  TestComp* this = (TestComp*) comp;

  cf_info_log("Connected comp=%p conn=%p chan=%u \n",comp,conn,chan);

  // Disable our receiver if we only support one connection per
  // receiver
  this->m->mr_disable(this->mObj,TEST_UUID,(char*)userData);

  return 1;
}
```

And, when a connection is broken, the component will be called in the *m_cli_disconnected* function.

```
static int
m_cli_disconnected(void* comp, CfMConn* conn, uint8_t chan,void* userData)
{
  TestComp* this = (TestComp*) comp;

  cf_info_log("Disconnected comp=%p conn=%p chan=%u \n",comp,conn, chan);

  // Re-enable our receiver if we only support one connection per  receiver
  this->m->mr_enable(this->mObj,TEST_UUID,(char*)userData);

  return 1;
}
```

The component will receive messages in the *m_cli_message* function.

```
static int
m_cli_message(void*          comp,
              CfMConn*       conn,
              uint8_t        chan,
              int            len,
              unsigned char* msg,
              void*          userData)
{
  TestComp* this = (TestComp*) comp;

  cf_info_log("Got message to %s: conn=%p chan=%u len=%d MSG=%s\n",
              (char*) userData,
              conn,
              chan,
              len,
              msg);

  unsigned char newMsg[256];

  strcpy(newMsg,"Hello, yourself!\n");

  this->m->send(this->mObj,conn,chan,strlen(newMsg)+1,newMsg);

  return 1;
}
```

5.1 M Client Library

Coming Soon!

6 C − Command Handler

Coming Soon!

7 Environment Variables

CF_COMP_DIR − Can be used to point out the directory where components are stored.

```
setenv CF_COMP_DIR dir1:dir2:dir3
```

8 Legal

Generated on Fri Jun 3 00:00:55 2005 by  1.4.0