

Robex and Robocomp:
Building intelligent robots

Robolab, 2008

Content on this document is licensed under a Creative Commons Attribution-Share Alike 3.0 license.

Pilar Bachiller. Robolab. University of Extremadura. <pilarb at unex.es>
Pablo Bustos. Robolab. University of Extremadura. <pbustos at unex.es>
Luis J. Manso. <luis.manso at gmail.com>

Content on this document is licensed under a Creative Commons Attribution-Share Alike 3.0 license.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	What is a component?	5
1.3	Component oriented programming remarks	6
1.3.1	Interface declaration	6
1.3.2	Component design	7
1.4	RobEx	7
2	Building our first component	8
2.1	Defining the interface	8
2.2	Component code	8
2.3	Component execution	10
2.4	Using our new component	11
3	Building our second component	13
3.1	Defining the interface	13
3.2	Language dependant issues	13
3.3	Component code	14
3.4	Compilation and usage	17
4	Introduction to RoboComp	19
4.1	Motivation	19
4.2	Component list	19
4.3	Fast component development with componentGenerator	19
4.4	Component manager	19
5	RoboComp Reference	20
5.1	AcpiComp	20
5.2	BaseComp	20
5.3	CamaraComp	20
5.4	CamMotionComp	20
5.5	JoystickComp	20
5.6	LaserComp	20
5.7	LinevisionComp	20
5.8	LocalNavigatorComp	21
5.9	RoimantComp	21
5.10	SpeechComp	21
5.11	VDescriptorsComp	21

5.12 VisionComp 21

Chapter 1

Introduction

This book describes the work behind RoboComp, the component oriented software system that is inside the Robex series of mobile robots. The motivation to build Robocomp is three fold:

1. The need for a software technology able of

1.1 Motivation

Some of the major problems to face when programming computer software, particularly complex software as robotics software, are scalability and reusability. Usually, people tend to produce giant and monolithic programs, which are unlikely to be reused. As projects grow to some thousands of lines, they become monsters out of control. This is not about concurrent version systems, or development team management, but about how the software is organized. In *Object Oriented* languages classes are the natural software elements, but when the number of classes and their interdependences grow it takes too much effort to understand the whole system. It is desirable to have some kind of recursive organization: recursive groupable elements within recursive groupable elements, so that pieces of software can be put together with a common interface. The startling point was unstructured software, then it passed away when structured programming came in. Object oriented programming came after structured programming. It can be said that component oriented programming is the next step.

To organize robotic software we use the concept of *components*. The use of *Component Oriented Programming* allows us to mitigate all the mentioned problems, encouraging bug isolation and detection as well.

1.2 What is a component?

A component is a program providing an interface which other components (programs) can use. They can run on different cores, processors within the same computer and, depending on the technology used, in different computers over the network. From a design point of view, they can be seen as a class providing some public methods, the only difference from this point of view is the complexity of the class: components should span some classes. How the communication between components take place depends on the technology used. Once implemented, a component may consist of a big amount of classes and threads, the only restriction is that it must offer a public interface as a *big class* would do.

After this introduction to the component concept, we are going to discover which technology do we use, and which characteristics does it have. The *framework* we are talking about is called *Ice*, it is an open-source (GPL licensed) product, mainly from the ZeroC company.

Besides its open-source nature, there are two main reasons that made us choose Ice. The first reason is its ease of use: unlike other technologies as Corbe, Ice philosophy is to not to provide rarely used features which mostly makes the software slow and difficult to learn. The second reason is that the generated network traffic is reasonably lightweight and fast to process, unlike XML based technologies as SOAP, the traffic is efficiency oriented not human-readable oriented.

Ice components can be called by subscription or through remote calls, RPC style. Since the subscription mechanism introduce a higher latency, which may be affordable in other fields but, very undesirable in robotics, in the book we will focus the RPC style calls. Besides, this method is easier to configure and use.

In order to use a remote component, its location must be specified through its *endpoint*: a text string which consists of all the information needed to reach it. The endpoint of a component follows the following structure:

```
componentname:protocol -p port -h host
```

componentname is the name of the component we want to communicate with. *protocol* is whether *tcp* or *udp*. *port* is the port number that the remote component is listening to. *host* is the host where the remote component runs, if not specified *localhost* will be assumed.

1.3 Component oriented programming remarks

Una vez se ha estructurado en componentes el software a construir, el siguiente paso es hacerlos realidad. A partir de este punto, a no ser que se especifique lo contrario, usaremos Ice.

El interfaz que los componentes ofrecen permiten que otros se comuniquen con él como si éste fuera un objeto más. Esta idea de usar un objeto remoto dentro de un programa no es nueva. Antes de que hubiera objetos y programación orientada a objetos, ya había un protocolo en Unix llamado RPC para hacer llamadas remotas a procedimientos o funciones. Más tarde surgieron tecnologías como RMI, CORBA, DCOM, .NET, Ice y algunas otras.

Para que esta tecnología funcione entre máquinas potencialmente distintas y entre procesos programados en diferentes lenguajes hay que resolver varios problemas difíciles. Hay dos fáciles de entender y que dan una idea de la magnitud del asunto. El primero es el de los tipos de datos. Cuando instanciamos en nuestro proceso o componente un objeto remoto, obtenemos un *proxy* o variable de acceso del tipo del objeto remoto. Una vez tenemos el proxy, se llama a los métodos remotos usando el operador `->` en C++, o `.` en Python. Para poder instanciar los proxys incluimos un fichero de cabecera en nuestro código que define los tipos remotos. Esto implica que necesitamos un mecanismo para definir tipos comunes y un mecanismo para traducir esos tipos comunes a los tipos nativos en tiempo de ejecución. Cuando dos componentes se comunican desde máquinas distintas, uno espera que si un método remoto devuelve un entero, llegue un entero a nuestro proceso, independientemente de cómo están internamente representados. Esa gestión tiene que hacerse cubriendo muchos casos distintos y de forma eficiente en tiempo de ejecución. El segundo problema es el del establecimiento y mantenimiento de la comunicación, y la gestión de excepciones entre máquinas conectadas por Internet. En este caso, cada proceso debe utilizar los mecanismos que su sistema operativo ofrece -protocolos TCP, UDP y otros- junto con estrategias de llamada, esperas, establecimiento de conexión, recuperación de caídas, etc. que finalmente permiten conectar dos o varios procesos entre sí a nivel de aplicación y mantenerlos en comunicación.

Hay dos aspectos en el que la construcción de un componente difiere de la de un programa común:

1.3.1 Interface declaration

Este paso no se da en otros paradigmas, hay que definir un interfaz para la funcionalidad que engloba el componente. Es muy común excederse haciendo un interfaz muy amplio, sin embargo, es una mala

decisión. Ofrecer un interfaz demasiado grande probablemente confundirá a los programadores que vayan a usarlo y nos dará más trabajo del necesario. Quedarse corto, evidentemente, tampoco es una buena opción.

1.3.2 Component design

As previously seen in 1.1, Ice is a framework. A framework is software concept similar to a library. En lugar de ampliar nuestro código enlazándolo con código compilado de terceros, lo que hace un *framework* es albergarnos dentro normalmente como una o un conjunto de hebras. De esta forma el programa principal *lo ponen ellos* y asumimos que como hebra tendremos periódicamente el control. A esta filosofía se le llama también *estilo Hollywood* por aquello de *no nos llame, nosotros le llamamos*. Lo que los *frameworks* consiguen es que al incorporar nuestro código el resultado sea algo distinto, nuevo que no es simplemente la suma de dos cosas. Los *frameworks* de comunicaciones para componentes distribuidos permiten convertir fácilmente un algoritmo pasivo en un proceso activo en el sistema operativo que puede responder a peticiones asíncronas desde otros procesos en la misma máquina o desde cualquier otra máquina conectada. El potencial es muy grande ya que nos permite pasar de un gran programa compuesto por cientos de clases y descompuesto en decenas de hebras, que hay que arrancar y parar cada vez que hacemos una modificación, a un grafo de procesos, posiblemente distribuidos en varios procesadores, en el que lo que ya está hecho y funcionando se puede dejar corriendo de forma permanente. Así, siempre trabajamos con partes pequeñas, podemos aprovechar mejor el hardware y la parte del sistema terminada acaba *integrándose* en el sistema operativo como si fueran servicios o demonios siempre a nuestra disposición. Como veremos, para la programación del robot esta forma de ver el ciclo de vida del software simplifica mucho las cosas y aumenta la eficiencia del desarrollo.

A parte de los aspectos a tener en cuenta comunmente, gracias a que los componentes pueden hacerse en distintos lenguajes de programación y combinarse, hay que considerar la posibilidad de reutilizar software hecho en un lenguaje distinto del que estamos acostumbrados. También tendremos que tener en cuenta que, generalmente, los componentes son programas concurrentes que tienen un hilo que satisface las llamadas remotas, y otro hilo que realiza cierto cómputo. Hay que asegurarse de que, si hay información compartida para ambos hilos, el acceso a ésta debe realizarse en exclusión mutua.

1.4 RobEx

Since we will be using the RobEx robot in some examples, this section will introduce you to it. Besides its relation with RoboComp, due to its open nature, RobEx is also an interesting robot platform to study.

Chapter 2

Building our first component

Various programming languages can be used when programming Ice components. In this book we will be using Python and C++. However, the Ice framework also supports Java, Ruby, .NET, and PHP, among others. As a first example, we will develop a slightly modified RoboComp component: `speechComp`. It is really simple but still fairly useful example.

2.1 Defining the interface

Interfaces are written in Slice, a compact specification language for specifying Ice interfaces. At some extent it is similar to C++ (in fact, C++ preprocessor directives can be used), but Slice is just used to define data types and some communication properties. The `speechComp` interface is easy to understand. It consists of a single remote method (*say*) interface (*Speech*), within a namespace named *RoboLabModSpeech*.

```
1 #ifndef SPEECH_ICE
2 #define SPEECH_ICE
3 module RoboLabModSpeech {
4     interface Speech {
5         bool say(string text);
6     };
7 };
8 #endif
```

So far, as you may suppose, the remote method is named *say*. It takes a single string parameter, *text*, and returns a boolean number. What is its purpose? The purpose of the methods are not interface-related, but as in the example, it should be easy to assume.

2.2 Component code

Language dependant code

Depending on the programming language used, there will be some changes. As the purpose of this first example is to be a practical introduction to the concepts behind component oriented programming it does not matter whether or not you will be using Python. Thus, so you should focus just in the concepts.

```
1 #!/usr/bin/env python2.4
2 # -*- coding: iso-8859-15 -*-
3
4 import sys, traceback, Ice, subprocess, threading, time, Queue, os
```

```

5 Ice.loadSlice("./Speech.ice")
6
7 import RobolabModSpeech
8
9 sleep_time = 0.1
10 max_queue = 100
11 charsToAvoid = ["'", '"', '{', '}', '[', '<', '>', '(', ')', '|']

```

The two first lines specify the format used to store source code file. Line 4 imports some Python libraries, C++ would perform this step by `#include` directives. Line 6 and 8 are used to define the data types associated with the component. Note that the *Slice* file should be located in `./Speech.ice`. Lines 10, 11, and 12 create three problem dependant constants.

Handler class

```

1 class SpeechHandler (threading.Thread):
2     def __init__(self):
3         threading.Thread.__init__(self)
4         self.text_queue = Queue.Queue(max_queue)
5     def run (self):
6         while 1:
7             if self.text_queue.empty():
8                 time.sleep(sleep_time)
9             else:
10                text_to_say = self.text_queue.get()
11                for rep in charsToAvoid:
12                    text_to_say = text_to_say.replace(rep, '\\'+rep)
13                shellcommand = "echo " + text_to_say + " | festival --tts"
14                print 'Order: ' + text_to_say
15                print 'Shell: "' + shellcommand + '"'
16                os.system(shellcommand)

```

The `SpeechHandler` class is the component class that do the 'real work'. Since we do not want remote calls to block the component until the speech request is finished, the class implements a thread by inheriting from `threading.Thread`. Otherwise, if another component makes a speech request, it, or other component making speech requests, both would be waiting for the request to end.

The `__init__()` method implements the constructor of the class, it calls the `Thread` constructor and initializes the queue for the speech requests. The `run()` method, which is the body of Python threads, sleeps while the queue is empty, elsewhere, it dequeues an element and executes the Festival[1] program with the text as input.

```

1     def put (self, new_text):
2         self.text_queue.put(new_text)

```

The `put` method will be run by the main thread. It just enqueues the parameter into the speech queue.

Interface class

```

1 class SpeechI (RobolabModSpeech.Speech):
2     def __init__(self):
3         self.handler = SpeechHandler()
4         self.handler.start()
5     def say (self, text, current = None):
6         print text
7         try:
8             self.handler.put(text)
9         except:
10            print 'Full queue.'

```

The *SpeechI* class implements the interface of the component, which is one of the most important classes of every component. It must implement, at least, all the methods the interface has. Thus, besides the constructor (line 34), the *SpeechI* class implements the *say()* method that calls the *put* method of the handler class (line 37).

In most of the cases it very important that the calls to the component interfaces do not block the execution of the calling component. This is achieved by making the interface class non blocking. That is why, usually the best option is to create a threaded handler class, so the only job of the interface class is to call to the handler class.

Server class and main thread

```

1 class Server (Ice.Application):
2     def run (self, argv):
3         status = 0;
4         try:
5             self.shutdownOnInterrupt()
6             adapter = self.communicator().createObjectAdapter('SpeechComp')
7             identity = self.communicator().stringToIdentity('speech')
8             adapter.add(SpeechI(), identity)
9             adapter.activate()
10            self.communicator().waitForShutdown()
11        except:
12            traceback.print_exc()
13            status = 1

1         if self.communicator():
2             try:
3                 self.communicator().destroy()
4             except:
5                 traceback.print_exc()
6                 status = 1
7
8 Server( ).main(sys.argv)

```

The main class of the component must inherit from the *Ice.Application* class. The *shutdownOnInterrupt()* method forces the application to exit when the user presses the *Ctrl+C* key combination (SIGINT signal). Then, an adapter is created (line 49) and a new *SpeechI* interface class is bound to it (line 50). In the line 51 the adapter is activated. In the line 52 the control is transferred to the framework by calling the *waitForShutdown()* method. The function will return at the end of the component execution, the rest of the code is for clean up.

2.3 Component execution

The Ice framework supports an easy way to introduce configuration from file support. You are free whether to use it or not, but the sake of simplicity we will only cover the case in which you want to. Assuming you will be using configuration files, you need to store this text in a file, for example, *speech.conf*:

```

1 SpeechComp.Endpoints=tcp -p 10021
2
3 # Component properties
4 Ice.Warn.Connections=0
5 Ice.Trace.Network=0
6 Ice.Trace.Protocol=0
7 Ice.ACM.Client=10
8 Ice.ACM.Server=10

```

You are now ready to run the component. Note that the component will give you no output but sound when another component calls its interface. We will cover this in the next section. Set the execution bit of the file and invoke it, specifying where the configuration file is located, i.e.:

```

1 user@oliva:~/speechComp$ ls -lah
2 total 20K
3 drwxr-x---  2 user user  4.0K 2008-11-25 10:28 .
4 drwxr-x--- 34 user user  4.0K 2008-11-25 10:28 ..
5 -rw-r----- 1 user user  1.7K 2008-11-25 10:16 speechComp.py
6 -rw-r----- 1 user user   158 2008-11-25 10:18 speech.conf
7 -rw-r----- 1 user user   127 2008-11-25 10:14 Speech.ice
8 user@oliva:~/speechComp$ chmod u+x speechComp.py
9 user@oliva:~/speechComp$ ./speechComp.py --Ice.Config=speech.conf

```

By pressing *Ctrl-C* twice the program will exit.

2.4 Using our new component

The calling program can be another component or a non-componentized program. The following example is not a component, just a program using a component. However, the only difference is that you would need to include this code in the code of another component in the case that you need communication between different component. The latter scenario is the most common one and will be extensively covered in the text. This is an example of the code of a client:

```

1 #!/usr/bin/env python2.4
2 import sys, traceback, Ice
3
4 Ice.loadSlice('./Speech.ice')
5 import RobolabModSpeech

```

The previous lines are mostly the same as in the component.

```

1 class Client (Ice.Application):
2     def run (self, argv):
3         self.shutdownOnInterrupt()
4
5         try:
6             properties = self.communicator().getProperties()
7             proxyString = properties.getProperty('SpeechProxy')
8         except:
9             print 'Cannot get SpeechProxy property.'
10            return

```

Despite that the application is a component or not, the main class must inherit from the *Ice.Application* class. The first part of the *run()* method of the *Client* main class is fairly simple. In the line 11 the endpoint of the proxy we want to communicate with is read from the configuration file. If the attempt fails, the application shows a error message and exists (lines 14 and 15).

```

1         try:
2             basePrx = self.communicator().stringToProxy(proxyString)
3             self.speechPrx = RobolabModSpeech.SpeechPrx.checkedCast(basePrx)
4         except:
5             traceback.print_exc()
6
7         messages = ['I am speechComp, the component for speech synthesis']
8         for message in messages:
9             print 'Say: "' + message + '"'
10            self.speechPrx.say(message)
11            print '\nDone'

```

In the second part of the `run()` method a *base proxy* (a proxy without a specific interface) is created in line 18, and casted to a `RobolabModSpeech.Speech` proxy in line 19. If anything goes wrong, the traceback is printed out in line 21.

At the end of the `run()` method, a list (with a single element) is created and for each list element, it is shown and `say` method of the speech component is called with the element as parameter.

```
1 def stop (self):
2     if self.communicator():
3         try:
4             self.communicator().destroy()
5         except:
6             traceback.print_exc()
7
8 c = Client()
9 c.main(sys.argv)
```

The `stop` method is executed at exit, so the communicator is destroyed. The main program just instantiates a `Client` class and transfer the control to it.

This will be the configuration file for the client program:

```
1 SpeechProxy = speech:tcp -p 10021
2
3 # Component properties
4 #
5 Ice.Warn.Connections=0
6 Ice.Trace.Network=0
7 Ice.Trace.Protocol=0
8 Ice.ACM.Client=10
9 Ice.ACM.Server=10
```

The first line specifies the component *endpoint*, that specify how to reach the component. Before the semicolon goes the component interface name `speech`. After it, the protocol (tcp or udp) and then, using the `-p` flag, the port where the component is listening to. With the `-h` flag different hosts can be specified.

Distributed computation is a key benefit of component oriented programming. If you want the component to run on the `192.168.0.10` host and the client to run somewhere else, you might set the configuration file as:

```
1 SpeechProxy = speech:tcp -p 10021 -h 192.168.0.10
2
3 # Component properties
4 #
5 Ice.Warn.Connections=0
6 Ice.Trace.Network=0
7 Ice.Trace.Protocol=0
8 Ice.ACM.Client=10
9 Ice.ACM.Server=10
```

For the speech component to work, Festival^[1] must be installed. If you want to take a look to the source of a simple component using another component, you can fetch the `acpiComp` component from RoboComp^[4].

Chapter 3

Building our second component

In this chapter we are going to develop a slightly more complex component. In our first example, written in Python in order to provide a better understanding, we built a simple component which provides an interface but do not make use of other components. The component which we are going to develop goes further and use the previous one. In order to not to cover only Python, this example is written in C++.

To avoid mixing up component oriented programming issues with the core of the component, this example is only the skeleton of what the real component would be. If implemented completely, it would provide information about the frequency of the CPU (Mhz) and its cache memory size (KB).

3.1 Defining the interface

The interface of the example consists of two methods returning integers: *getCPUFrequency()* and *getCacheSize()*. Obviously, the first one will return the current frequency of the CPU (it may vary) and its cache size, which is constant. Said that, there is nothing more to take into account, the next step is to code the Slide file.

```
1 #ifndef CPU_ICE
2 #define CPU_ICE
3 module RobolabModCPU {
4     interface CPU {
5         int getCPUFrequency();
6         int getCacheSize();
7     };
8 };
9 #endif
```

3.2 Language dependant issues

Unlike Python, C++ is not a dynamic language. That is why we must compile the Slice code of the *.ice* file into C++ before the C++ code is compiled into machine code. This only means that you must use *slice2cpp* to generate the files that C++ is going to use. For CPU.ice these are CPU.h and CPU.cpp.

```
1 luisj@oliva:~/cpuComp$ ls
2 CPU.ice
3 luisj@oliva:~/cpuComp$ slice2cpp CPU.ice
4 luisj@oliva:~/cpuComp$ ls
5 CPU.cpp CPU.h CPU.ice
6 luisj@oliva:~/cpuComp$
```

As our component will make use of a second component (the one of the first example), we must generate the correspondent code with the Slice file of the component we want to use:

```

1 luisj@oliva:~/cpuComp$ ls
2 CPU.cpp CPU.h CPU.ice
3 luisj@oliva:~/cpuComp$ slice2cpp ../speechComp/Speech.ice
4 luisj@oliva:~/cpuComp$ ls
5 CPU.cpp CPU.h CPU.ice Speech.cpp Speech.h
6 luisj@oliva:~/cpuComp$

```

3.3 Component code

main.cpp

```

1 #include <Ice/Ice.h>
2 #include <Ice/Application.h>
3
4 #include "cpuworker.h"
5 #include "cpuI.h"
6
7 using namespace std;
8 using namespace RobolabModCPU;
9 using namespace RobolabModSpeech;

```

The previous lines just included some headers and made default use of some name spaces. The next listing is the definition of the CPUComp class, a child class of *Ice::Application* which is needed in every Ice component. It only must implement the run method:

```

1 class CPUComp : public Ice::Application {
2 public:
3     virtual int run(int, char*[]);
4 };

```

The following listing is the first part of the *run()* method. It gets the property *SpeechProxy* from the configuration file and use it as the endpoint of the component we want to use (a description of how to reach it). If the property is not found the program report the error and stops. If the property fetch was succesful, then it attemps to connect to the other component, reporting wether the result was succesful or not:

```

1 int CPUComp::run(int argc, char* argv[]) {
2     int status = EXIT_SUCCESS;
3     string proxy;
4     SpeechPrx speech_proxy;
5
6     try {
7         proxy = communicator()->getProperties()->getProperty( "SpeechProxy" );
8         if (proxy.empty()) {
9             cerr << "Error loading proxy config!" << endl;
10            return EXIT_FAILURE;
11        }
12
13        speech_proxy = SpeechPrx::uncheckedCast( communicator()->stringToProxy( proxy ) );
14        if (!speech_proxy) {
15            cerr << "Error loading proxy!" << endl;
16            return EXIT_FAILURE;
17        }
18    } catch(const Ice::Exception& ex) {
19        cerr << "Exception: " << ex << endl;
20        return EXIT_FAILURE;

```

```

21     }
22     cout << "SpeechProxy initialized Ok!" << endl;

```

Once we have established the connection to the speech component, we can use it on our objects (*worker* on the first line of the following listing). The next step is to create the interface of the component. Usually, the interface class (`cpuI` in this case) is provided with a worker class (the one doing the real work). Once the interface class is created, we can attempt to bind its class interface to the component interface by adding it to the component adapter and activating the adapter. Since the `CPUComp` inherits from `Ice::Applicaition` which is a thread class, the last lines only implement a non-ending loop in order to not to end the execution once connections are established:

```

1     CPUWorker *worker = new CPUWorker(speech_proxy);
2     try {
3         Ice::ObjectAdapterPtr adapter = communicator()->createObjectAdapter("CPUComp");
4         CPUI *cpuI = new CPUI(worker);
5         adapter->add(cpuI, communicator()->stringToIdentity("cpu"));
6         adapter->activate();
7         cout << "CpuComp started" << endl;
8     } catch(const Ice::Exception& ex) {
9         status = EXIT_FAILURE;
10        cout << "Exception raised on main thread: " << endl;
11        cout << ex;
12    }
13
14    for (;;)usleep(500000) {
15        cout << "beep" << endl;
16    }
17    return status;
18 }

```

The main function is fairly easy to understand once the `CPUComp` class is understood. It gets the configuration file path and runs the thread of the `CPUComp` class.

```

1 int main(int argc, char* argv[]) {
2     bool hasConfig = false;
3     string arg;
4     CPUComp app;
5
6     for (int i = 1; i < argc; ++i) {
7         arg = argv[i];
8         if ( arg.find ( "--Ice.Config=", 0 ) != string::npos )
9             hasConfig = true;
10    }
11
12    if ( hasConfig )
13        return app.main( argc, argv );
14    else
15        return app.main(argc, argv, "config");
16 }

```

cpuI.h

The `CPUI` class is the class which will act as the component interface, and so it must implement all the interface methods:

```

1 #ifndef CPUI_H
2 #define CPUI_H
3
4 #include <Ice/Ice.h>
5 #include <CPU.h>
6 #include <Speech.h>
7 #include "cpuworker.h"

```

```

8
9 class CPUI : public virtual RobolabModCPU::CPU {
10 public:
11     CPUI(CPUWorker *_worker);
12     ~CPUI() {}
13     Ice::Int getCPUFrequency(const Ice::Current&);
14     Ice::Int getCacheSize(const Ice::Current&);
15 private:
16     CPUWorker *worker;
17 };
18
19 #endif

```

cpuI.cpp

In cpuI.cpp it will be stored the implementation of the CPUI class:

```

1 #include "cpuI.h"
2
3 CPUI::CPUI(CPUWorker *_worker) {
4     worker = _worker;
5 }
6
7 Ice::Int CPUI::getCPUFrequency(const Ice::Current&) {
8     return worker->getCPUFrequency();
9 }
10
11 Ice::Int CPUI::getCacheSize(const Ice::Current&) {
12     return worker->getCacheSize();
13 }

```

cpuworker.h

In a real world component, the component would do something more than providing the interface. The CPUI class do not make the real job, it just calls some methods of another class, and CPUWorker is that class. The code of CPUWorker could be inside the CPUI in this example due to its extremely low complexity. However in real world components, the worker code will be complex enough to have its own class. We followed this approach because it is considered a good practice habit.

This is the definition of the CPUWorker class:

```

1 #ifndef CPUWORKER_H
2 #define CPUWORKER_H
3
4 #include "Speech.h"
5 #include "CPU.h"
6
7 using namespace RobolabModCPU;
8 using namespace RobolabModSpeech;
9
10 class CPUWorker {
11 public:
12     CPUWorker(SpeechPrx &speech_param);
13     ~CPUWorker() {}
14     Ice::Int getCPUFrequency();
15     Ice::Int getCacheSize();
16 private:
17     SpeechPrx speech;
18     int lastFrequency;
19 };

```

```
20
21 #endif
```

cpuworker.cpp

This is the implementaiton of the CPUWorker class. Note that, as its constructor receives a proxy to the speech component, you can use its interface inside CPUWorker whenever you want:

```
1 #include "cpuworker.h"
2
3 CPUWorker::CPUWorker(SpeechPrx &speech_param) {
4     speech = speech_param;
5     speech->say("CPU comp is up.");
6     lastFrequency = -1;
7 }
8
9 Ice::Int CPUWorker::getCPUFrequency() {
10    int frequency = 1000; // This is what makes the example just a
11                          // skeleton. If implemented, it would be
12                          // actually useful.
13    if (frequency != lastFrequency and lastFrequency != -1)
14        speech->say("Frequency changed!");
15    lastFrequency = frequency;
16    return frequency;
17 }
18
19 Ice::Int CPUWorker::getCacheSize() {
20    return 512;
21 }
```

3.4 Compilation and usage

Compilation

```
1 luisj@oliva:~/cpuComp$ g++ -c -O2 -Wall -W -I. -o cpuI.o cpuI.cpp
2 luisj@oliva:~/cpuComp$ g++ -c -O2 -Wall -W -I. -o cpuworker.o cpuworker.cpp
3 luisj@oliva:~/cpuComp$ g++ -c -O2 -Wall -W -I. -o CPU.o CPU.cpp
4 luisj@oliva:~/cpuComp$ g++ -c -O2 -Wall -W -I. -o Speech.o Speech.cpp
5 luisj@oliva:~/cpuComp$ g++ -c -O2 -Wall -W -I. -o main.o main.cpp
6 main.cpp:17: warning: unused parameter 'argc'
7 main.cpp:17: warning: unused parameter 'argv'
8 luisj@oliva:~/cpuComp$ OBJS="main.o cpuI.o cpuworker.o CPU.o Speech.o"
9 luisj@oliva:~/cpuComp$ g++ -Wl -o cpuComp $OBJS -lIce -lIceUtil -lpthread
```

Usage

This the following would be the config file of our brand new component, cpu.conf. If you want to use a speech component that you are running in other computer, just replace “localhost” to the host name where it is running. Recall that the component properties are optional.

```
1 CPUComp.Endpoints=tcp -p 10099
2 SpeechProxy=speech:tcp -p 10021 -h localhost
3
4 #
5 # Component properties
6 #
7 Ice.Warn.Connections=0
8 Ice.Trace.Network=0
```

```
9 Ice.Trace.Protocol=0
10 Ice.ACM.Client=10
11 Ice.ACM.Server=10
```

In order to make it run:

```
1 luisj@oliva:~/cpuComp$ ./cpuComp --Ice.Config=cpu.conf
2 SpeechProxy initialized Ok!
3 CpuComp started
4 beep
5 beep
6 beep
```

Chapter 4

Introduction to RoboComp

4.1 Motivation

introduction to robocomp

4.2 Component list

4.3 Fast component development with componentGenerator

When a new programming paradigm is developed, it always takes some time to have a full featured programming language providing support for the paradigm. Despite the theoretical foundations of component oriented programming dates back to 1968[2], only a few programming languages support it[3], and none of them have been widely spread.

The lack of component oriented programming features within current mainstream programming languages may be the major problem when developing component oriented software. Since creating the skeleton of a new component is a boring tedious process, a component skeleton code generator has been developed within the RoboComp project: *componentGenerator*. The code generator requests the name of the new component and the name of the components it will connect to. As its output, it generates a ready-to-use source tree for the new component, with all the most commonly needed classes.

Example of use

Generated source tree

4.4 Component manager

Components are independent programs wich are independently executed. They can be run manually, but when a software systems exceeds a reasonably small number of components it becomes hard to monitor or manage. In order to make those process an easy task, a component manager, *managerComp* has been developed.

use

features

images

Chapter 5

RoboComp Reference

This chapter contains reference documentation over the complete set of RoboComp components. Each section describes one component in detail with examples of connection to other components forming graphs of processes.

5.1 AcpiComp

basic laptop state (battery, temperature, wifi signal) monitoring using the acpi interface

5.2 BaseComp

Motor control of a differential-drive robot

5.3 CamaraComp

Image acquisition from firewire, v4l2 and mplayer

5.4 CamMotionComp

Motor control of a 3 DOFs common tilt binocular head

5.5 JoystickComp

Joystick control of robot movements

5.6 LaserComp

URG-04LX laser acquisition and preprocessing

5.7 LinevisionComp

Detection of linear segments

5.8 LocalNavigatorComp

Local navigation with VHF+

5.9 RoimantComp

Tracking of ROIs in image space

5.10 SpeechComp

Speech synthesis of text using the Festival speech synthesis software

5.11 VDescriptorsComp

RIFT descriptors computation from ROIs

5.12 VisionComp

Real time (Harris-Laplace, Hess-Laplace) regions of interest (ROIs) in a pyramidal or prismatic (foveated) multiscale setup

Bibliography

- [1] D. Abberley, S. Renals, G. Cook, and T. Robinson. The 1997 THISL spoken document retrieval system. In *Proc. Sixth Text Retrieval Conference (TREC-6)*, pages 747–752, 1998.
- [2] M. D. McIlroy. Mass produced software components. pages 138–155, 1968.
- [3] Per Brand Peter Van Roy, Seif Haridi and Raphael Collet. Distributed programming in moztart - a tutorial introduction.
- [4] Robolab. RoboComp Wiki. <http://robocomp.wiki.sourceforge.net/>.