GNUstep Distributed Objects

Nicola Pero n.pero@mi.flashnet.it

January 2002 AD

1 What are Distributed Objects (DO)

In this tutorial we introduce the reader to the world of *GNUstep Distributed Objects*. GNUstep Distributed Objects are a collection of facilities offered by the GNUstep base library which allow communication between different processes running on the same machine or on different machines on the same network. Distributed Objects might also be used for communicating between different threads in the same application.

The basic idea in GNUstep Distributed Objects is that of extending normal object oriented programming by allowing objects in different processes to transparently call methods of each other. The low level details of how this interprocess messaging is done are normally hidden inside the GNU Objective-C runtime and the GNUstep base library; this allows you to concentrate on a higher level of abstraction, designing a distributed application as composed of many objects running in different processes, and intercommunicating between them using messaging. GNUstep Distributed Objects provide you support for implementing this type of designs easily, cleanly, and very quickly. The resulting code is natural and readable, which reduces the effort required to maintain and extend the distributed application.

In practice, because the Objective-C language has built in support for remote messaging, you do not need to use any special syntax or any additional programming tools to send a remote message – you can just do it with the same code you use to send a normal message. As a consequence, GNUstep Distributed Objects are very simple and natural to use, and they fit very easily and very elegantly in an object oriented framework.

The name "GNUstep Distributed Objects" is often abbreviated *GNUstep DO*. In this tutorial we will be even more brief, and refer to them as *DO*.

2 A basic program to show files

We start our tutorial with a very basic non-DO program – a command line tool which accepts a filename as an argument, reads this file from disk, and prints it out to `stdout`. For example, typing

```
Example Client.m
```

should display the contents of the file `Client.m`. Once we have this basic example in place, in the next sections we'll use DO to extend its capabilities.

Our basic tool is the following one:

```
#include <Foundation/Foundation.h>

/* This object does the job of fetching a file from
   the hard disk */

@interface FileReader : NSObject
- (NSString *)getFile: (NSString *)fileName;
@end

@implementation FileReader
- (NSString *)getFile: (NSString *)fileName
{
  return [NSString stringWithContentsOfFile: fileName];
}
@end
```

```
int
main (void)
{
  NSAutoreleasePool *pool;
  NSArray *args;
  int count;
  FileReader *reader;
  NSString *filename;
  NSString *file;

  pool = [NSAutoreleasePool new];

  /* Create our FileReader object */
  reader = [FileReader new];

  /* Get program arguments */
  args = [[NSProcessInfo processInfo] arguments];

  /* the first string in args is the program name;
     get the second one if any */
  if ([args count] == 1)
    {
      NSLog (@"Error: you should specify a filename");
      exit (1);
    }

  filename = [args objectAtIndex: 1];

  /* Ask the reader object to get the file */
  file = [reader getFile: filename];

  /* If the reader object could get the file, show it */
  if (file != nil)
    {
      printf ("%s\n", [file lossyCString]);
    }
  else
    {
      NSLog (@"Error: could not read file `%@'", filename);
      exit (1);
    }

  return 0;
}
```

To compile the program, you need a `GNUmakefile`, such as the following one:

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = Example
Example_OBJC_FILES = example.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

3 Extending the program to deal with files on the network

Now we want to modify our little program to retrieve files from other machines on our network, and display them on the local machine.

Looking again at the source code of our tool, we see that we have a `main` function which parses the command line arguments, and then calls the `getFile:` method of a `FileReader` object to read the file; then it displays it. With the help of a bit of magic, you could simply run the `FileReader` object on the remote machine, and the `main` function on the local machine, without changing your code. The `main` function would parse the command line arguments, and then call the `getFile:` method of the `FileReader` object (running on the remote machine), and the `FileReader` object would read the file and return it to the `main` function (which is running on the local machine) as the return value of the `getFile:` method; the `main` function would print it.

GNUstep Distributed Objects allow you to do this kind of magic: calling methods of remote objects precisely as if they were normal local objects.

The design of our modified example will be a server/client design: we have a server, and a client. The server will contain the `FileReader` object; the client will connect to the server, and send the `getFile:` message to the `FileReader` object in the server, getting in return a string containing the file contents.

4 The server

We put the source code for the server in the `Server.m` file. Basically, the server consists of the `FileReader` class, plus a new `main` function, which creates an instance of `FileReader` and *vends* it to the network, that is, it exposes it to the network, allowing other remote processes to call its methods via GNUstep Distributed Objects. Then, the server enters into a run-loop waiting for something to happen.

The code to vend the object to the network is quite simple: you get the `defaultConnection` object:

```
NSConnection *conn = [NSConnection defaultConnection];
```

then you tell the connection which object you want to vend:

```
[conn setRootObject: reader];
```

and finally, you register it on the network with a certain name:

```
if (![conn registerName:@"FileReader"])
  {
    NSLog (@"Could not register us as FileReader");
    exit (1);
  }
```

the name is quite important – the client needs to know the name of the server to establish a connection with it and access the vended object (which is the FileReader object in this case).

So, here is the full code for the server:

```
#include <Foundation/Foundation.h>

/* This object does the job of fetching a file from
   the hard disk */

@interface FileReader : NSObject
- (NSString *)getFile: (NSString *)fileName;
@end

@implementation FileReader
- (NSString *)getFile: (NSString *)fileName
{
  return [NSString stringWithContentsOfFile: fileName];
}
@end
```

```
int
main (void)
{
  NSAutoreleasePool *pool;
  FileReader *reader;
  NSConnection *conn;

  pool = [NSAutoreleasePool new];

  /* Create our FileReader object */
  reader = [FileReader new];

  /* Get the default connection */
  conn = [NSConnection defaultConnection];

  /* Make the reader available to other processes */
  [conn setRootObject: reader];

  /* Register it with name ‘FileReader’ */
  if (![conn registerName:@"FileReader"])
    {
      NSLog (@"Could not register us as FileReader");
      exit (1);
    }

  NSLog (@"Server registered - waiting for connections...");

  /* Now enter the run loop waiting forever for clients */
  [[NSRunLoop currentRunLoop] run];

  return 0;
}
```

5 The client

The client is implemented in the `Client.m` file; it is composed by the `main` function of the original program, with some simple yet very interesting changes.

5.1 The FileReader protocol

Because the `FileReader` class is not compiled into the client (as we moved it into the server), we can't refer to the `FileReader` class in the client. And still, if we need to access the remote `FileReader` object, we need a way to declare which methods it supports.

To manage this situation, we use a `protocol`. If you know Java, this is similar to an `interface` in Java. A `protocol` declares some methods, leaving the implementation unspecified. The language then allows you to declare that a certain object *conforms* to a certain protocol; this means that the object implements the methods listed in the protocol. In our example, this allows us to declare that we can send the `getFile:` method to the `reader` object, without actually knowing the implementation of the method nor actually knowing the class of the `reader` object.

The declaration of the protocol is as follows:

```
@protocol FileReader
- (NSString *) getFile: (NSString *)fileName;
@end
```

This declares the protocol `FileReader` to have a single method, `getFile:`. Objects conform to this protocol if and only if they have a `getFile:` method taking a `NSString *` argument, and returning an `NSString *`.

The `reader` object, which we used to declare to be of class `FileReader`,

```
FileReader *reader;
```

is now declared more generically to conform to the `FileReader` protocol:

```
id <FileReader> reader;
```

`id` means a generic object; `<FileReader>` means that it must conform to the `FileReader` protocol; in this case this simply means that `reader` is an object and you can send the message `getFile:` to it.

5.2 Accessing the remote FileReader object

To access the `reader` object, where we used to create it directly,

```
reader = [FileReader new];
```

we now ask the gnustep-base library to give us the object registered with the name `FileReader` on a remote machine:

```
reader = (id <FileReader>)[NSConnection
               rootProxyForConnectionWithRegisteredName: @"FileReader"
               host: @"*"];
```

strictly speaking, `reader` is a local proxy to the remote object – but the whole thing is made so that you can forget about this distinction, and think of `reader` simply as the remote object. Using `*` for the host argument means that gnustep-base will look for an object registered with name `FileReader` anywhere on the network; if you know the host on which you want to access the `FileReader` object, you should better use your specific host name, such as `localhost` or `192.14.29.1`.

We need a cast to `id <FileReader>` because the call to `NSConnection` returns a generic object, while we know the `FileReader` object implements `getFile:`. A more robust application could check at execution time that the remote object in the server actually can respond to `getFile:` messages before doing the cast (for example by using the method `respondsToSelector:`); we skip this little complication in this first example.

But we need to check that we have a real `reader` object – if it is `nil`, it is because for some reason the gnustep-base library couldn't connect to an object registered as `FileReader` on the network. Usually this is because the server is not running; there is nothing we can do in the client in these cases, so we simply print an error message and exit.

As promised, the rest of the function is unchanged; in particular, when we send the `getFile:` method to the remote object, that starts a network connection to the server, and returns the result – but the nice thing is that we don't need to do anything special to perform this remote call: we just call the method normally, as if the object were our old friendly local object.

Here is the source code:

```
#include <Foundation/Foundation.h>

/* This tells us how the reader object behaves */

@protocol FileReader
- (NSString *)getFile: (NSString *)fileName;
@end


int
main (void)
{
  NSAutoreleasePool *pool;
  NSArray *args;
  int count;
  id <FileReader> reader;
  NSString *filename;
  NSString *file;

  pool = [NSAutoreleasePool new];
```

```
  /* Create our FileReader object */
  reader = (id <FileReader>)[NSConnection
             rootProxyForConnectionWithRegisteredName:
                   @"FileReader"
             host: @"*"];

  if (reader == nil)
    {
      NSLog (@"Error: could not connect to server");
      exit (1);
    }

  /* From now on the code is the same, whether reader is
     in the local process or in a remote one */

  /* Get program arguments */
  args = [[NSProcessInfo processInfo] arguments];

  /* the first string in args is the program name;
     get the second one if any */
  if ([args count] == 1)
    {
      NSLog (@"Error: you should specify a filename");
      exit (1);
    }

  filename = [args objectAtIndex: 1];

  /* Ask the reader object to get the file */
  file = [reader getFile: filename];

  /* If the reader object could get the file, show it */
  if (file != nil)
    {
      printf ("%s\n", [file lossyCString]);
    }
  else
    {
      NSLog (@"Error: could not read file '%@'", filename);
      exit (1);
     }

  return 0;
}
```

6 Putting them together

For the sake of completeness, here is the `GNUmakefile`:

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = Server Client

Server_OBJC_FILES = Server.m
Client_OBJC_FILES = Client.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

After you compile the two tools, you have definitely to play with them because it's real fun. Start the server:

```
./obj/Server
```

Now open another shell (on another machine on the same network if you are so lucky that you can play on the network), and start the client from there:

```
./obj/Client Client.m
```

this should display you the `Client.m` file, as fetched by the `Server` program.

On my machine it works so simply, immediately and nicely that it is even difficult to realize that the client has actually opened a network connection to the server, asked for the file, and the server has sent it back through the connection! To get more feeling of what is happening,you might want to modify the `FileReader` class in the server to display a log each time it sends a file:

```
@implementation FileReader
- (NSString *)getFile: (NSString *)fileName
{
  NSLog (@"A client asked for file %@", fileName);
  return [NSString stringWithContentsOfFile: fileName];
}
@end
```

7 A modified client which accesses both local and remote files

The previous sections introduced you to the black magic of DO: we replaced a local object with a remote one, and we still called a method of the object in the usual way. We didn't need to use a special syntax to send messages to the remote object: we just did as if it still were our old little object in the local process; the language and the libraries managed silently all the rest. Now we want to push it even further: we do the replacement at execution time.

We want to extend the client so that it can access both local and remote files; it will decide what to do depending on the command line arguments which are used to call it. When called with a single argument, for example `README`,

```
Client README
```

it will display the file `README` from the local machine; when called with a filename and a host name, for example `README` and `didone.gnustep.it`,

```
Client README didone.gnustep.it
```

it will display the file `README` fetched from the server running on the host called `didone.gnustep.it`.

The server is the same; we only change the client. We declare the usual `FileReader` protocol because we now want to have a `reader` object which might be local or remote, and the only thing we know is that we can send the `getFile:` message to it, which is expressed in Objective-C by saying that the `reader` object conforms to the `FileReader` protocol.

Then, we implement a `LocalFileReader` class, which reads a file (from the local machine). I called it `LocalFileReader` rather than `FileReader` to avoid confusion, but technically there is nothing preventing you from calling it `FileReader`. It is the usual class, which implements the facility of reading a local file. But the class declaration has an interesting modification:

```
@interface LocalFileReader : NSObject <FileReader>
- (NSString *)getFile: (NSString *)fileName;
@end
```

in this declaration, `LocalFileReader` inherits from `NSObject`, and conforms to the `FileReader` protocol. We declare that the class implements the protocol because that allows us to cast any `LocalFileReader` object to `id <FileReader>`.

When the client is run, it examines its arguments to decide whether it needs to create a local or a remote `reader` object:

```
/* Get program arguments */
args = [[NSProcessInfo processInfo] arguments];

/* If there is a second argument, read it as a hostname
   to fetch files from */
if ([args count] > 2)
  {
    NSString *host = [args objectAtIndex: 2];

    /* Create our remote FileReader object */
    reader = (id <FileReader>)
      [NSConnection
        rootProxyForConnectionWithRegisteredName: @"FileReader"
        host: host];

    if (reader == nil)
      {
        NSLog (@"Error: could not connect to server on host %@",
               host);
        exit (1);
      }
  }
else /* No second argument - read local file */
  {
    reader = [LocalFileReader new];
  }
```

and that's it; the code which follows is the usual one. Notice how both the remote and the local object conform to the `FileReader` protocol, which makes it possible for the code to contact the local and the remote object in an opaque way.

The full client source code is:

```
#include <Foundation/Foundation.h>


/* This tells us how the reader object behaves */

@protocol FileReader
- (NSString *)getFile: (NSString *)fileName;
@end

/* A local file reader conforms to the FileReader protocol
   and reads files locally */
@interface LocalFileReader : NSObject <FileReader>
- (NSString *)getFile: (NSString *)fileName;
@end

@implementation LocalFileReader
- (NSString *)getFile: (NSString *)fileName
{
  return [NSString stringWithContentsOfFile: fileName];
}
@end


int
main (void)
{
```

```
NSAutoreleasePool *pool;
NSArray *args;
int count;
id <FileReader> reader;
NSString *filename;
NSString *file;

pool = [NSAutoreleasePool new];

/* Get program arguments */
args = [[NSProcessInfo processInfo] arguments];

/* If there is a second argument, read it as a hostname
   to fetch files from */
if ([args count] > 2)
  {
    NSString *host = [args objectAtIndex: 2];

    /* Create our remote FileReader object */
    reader = (id <FileReader>)
      [NSConnection
        rootProxyForConnectionWithRegisteredName:
              @"FileReader"
        host: host];

    if (reader == nil)
      {
        NSLog
          (@"Error: could not connect to server on host %@",
           host);
        exit (1);
      }
  }
else /* Local file */
  {
    reader = [LocalFileReader new];
  }

/* From now on the code is the same, whether reader is
   in the local process or in a remote one */

/* the first string in args is the program name;
   get the second one if any */
if ([args count] == 1)
  {
    NSLog (@"Error: you should specify a filename");
    exit (1);
  }

filename = [args objectAtIndex: 1];

/* Ask the reader object to get the file */
file = [reader getFile: filename];

/* If the reader object could get the file, show it */
if (file != nil)
  {
```

```
      printf ("%s\n", [file lossyCString]);
    }
  else
    {
      NSLog (@"Error: could not read file ‘%@’", filename);
      exit (1);
     }

  return 0;
}
```

NB: If you play with this client, and if you want to pass * as host name to have it look in all the network, make sure you escape the * to prevent it being expanded by the shell:

```
./obj/Client Client.m \*
```

At this point we probably need to say a few words about why and how this magic is possible. In other words, it's time for a bit of sane religious praise of our beloved Objective-C language `:-)`.

The whole magic is in the method invocation

```
file = [reader getFile: filename];
```

which invokes the method `getFile:` of the object `reader`, no matter if `reader` is a local or a remote object; the type of the `reader` object is determined only at execution time, depending on the command line arguments which were passed to the tool. The example is impressive because it shows that the language allows you to replace *any* local object in your application with a remote object at execution time, without recompiling or restarting your code, and everything will still work, assuming of course that the remote object can respond to the methods the local object could. This is possible because Objective-C provides `dynamic binding` of method invocations, which means that the method invocation is bound to the method implementation only at runtime. This allows many powerful object oriented designs which wouldn't be possible otherwise (Distributed Objects are an example of such a design); typically designs in which objects are dynamically and cleanly replaced with other objects at runtime, or designs in which an object encapsulates some functionality, but the actual class or implementation of the object is not known till execution time.

Objective-C is the fastest language available which supports this kind of advanced object oriented designs; Objective-C code is normally as fast as C because it *is* C code, except in method invocations (which C doesn't have) which on average take three times more than the time required by a function invocation. It's practically impossible for a fully object oriented language to go faster than that; Objective-C performs nearly as fast as C++, but yet provides you with dynamic binding and many other very advanced and flexible object oriented features (such as categories, or access to the runtime internals) which C++ doesn't provide. Some of these features are missing even in Java.

8 Error checking - timeout exceptions

Up to now, this tutorial has ignored error checking; but to build robust applications, your code needs to be able to manage problems in the network connections, both in the server and in the client.

In this section we examine the simplest problem: when a problem occurs during invocation of a remote method, typically a time-out on the network connection, and an exception is raised. If your client code is to use remote objects in a robust way, you need to be prepared to catch and manage these exceptions in the relevant sections of code.

A timeout means that your client sent the method invocation to the server, but it didn't get a response in a reasonable time. After waiting for 15 seconds (FIXME - this was the old default for gnustep-base, it's probably changed now), the gnustep base library raises a `NSPortTimeOutException`, which you need to catch if you want to write robust code.

In our example, we need to catch exceptions thrown when getting the file from the `FileReader`:

```
NS_DURING
  {
    file = [reader getFile: filename];
  }
```

```
NS_HANDLER
  {
    NSLog (@"Got exception while reading file: %@",
           localException];
    exit (1);
  }
NS_ENDHANDLER
```

this example is not particularly brilliant because the only thing we do when we catch the exception is printing out the description of the exception and quitting the program – which is the same thing which the gnustep-base library does for us if we don't catch the exception... but it should give you a clear example of how to catch and manage timeouts.

9 Error checking - connection died

The other typical error which can occur is that a connection dies. As an example, suppose that we had a modified client, in which we get a remote `reader` object, and then loop waiting for the user to input a filename. When the user inputs a filename, we ask to the remote object to get that file, and we print it out, then we go back into the loop waiting for the user to input another filename.

In that situation, it might happen that the remote server is up and running, but at a certain point, while we are in our loop waiting for the user to input a filename, the remote server becomes unavailable (for example, because of a hardware crash, or because the network connection goes down, or because the server crashed with a segmentation fault during some operation). If the gnustep base library detects this problem, it invalidates the proxy to the remote object (which in plain words means that the `reader` object is no longer useful, and you should no longer use it), and posts the `NSConnectionDidDieNotification` notification. If we are observing that notification, we can be informed immediately that this problem has occurred, and we might manage it – we might want to exit the application immediately, or try to connect to another server, or inform the user that because the remote server is down, we will only fetch files locally till we can connect to the remote server again, or something else. In any case, the notification gives us an opportunity to release our `reader` object – which is no longer useful because the connection has died – and, if we want and if we can, to take some corrective measures. Here is the simple code needed to observe the notification:

```
 [[NSNotificationCenter defaultCenter]
    addObserver: myObserver
       selector: @selector(methodToExecuteWhenTheConnectionDies:)
          name: NSConnectionDidDieNotification
        object: [(NSDistantObject *)reader connectionForProxy]];
```

Calling this code after the `reader` object is created will cause the `methodToExecuteWhenTheConnectionDies:` of the `myObserver` object to be called when the connection to the server goes down.

Here is an example `Client.m` code which you can use as a starting point for playing with dying connections –

```
#include <Foundation/Foundation.h>


@protocol FileReader
- (NSString *)getFile: (NSString *)fileName;
@end

@interface Observer : NSObject
- (void)connectionDied: (NSNotification *)not;
@end

@implementation Observer
- (void)connectionDied: (NSNotification *)not
{
  NSLog (@"Connection to server died! - exiting");
  exit (1);
```

```
}
@end

int
main (void)
{
  NSAutoreleasePool *pool;
  NSArray *args;
  int count;
  id <FileReader> reader;
  NSString *filename;
  NSString *file;

  pool = [NSAutoreleasePool new];

  /* Get program arguments */
  args = [[NSProcessInfo processInfo] arguments];

  /* Create our remote FileReader object */
  reader = (id <FileReader>)
    [NSConnection
      rootProxyForConnectionWithRegisteredName: @"FileReader"
      host: @"*"];

  if (reader == nil)
    {
      NSLog (@"Error - could not connect to FileReader Server");
      exit (1);
    }
  else
    {
      NSLog (@"Connected");
    }


  /* Register Observer -connectionDied: to be informed if the connection
     to the server dies.  */
  [[NSNotificationCenter defaultCenter]
    addObserver: [Observer new]
    selector: @selector(connectionDied:)
    name: NSConnectionDidDieNotification
    object: [(NSDistantObject *)reader connectionForProxy]];

  /* Ok - now we enter the main run loop.  In this example, we just do
     nothing in the main run loop as I'm too lazy to code it to do
     something -- but you should imagine that this is a server doing
     a lot of stuff in the run loop, and using the 'reader' to fetch
     files from the remote server when its activity in the run loop
     requires it.  */
  [[NSRunLoop currentRunLoop] run];

  return 0;
}
```

You can play by running the server in an xterm, then running the client in another xterm, then killing the server in the first xterm (for example by pressing `Control-C`): the client in the second xterm should be immediately notified that the connection to the server is dead!

10 For further information

This tutorial introduced you to the basics of Distributed Objects, but there are many more interesting things to explore. A good starting point for more fun with distributed objects is the `NSConnection` documentation. A very complete (but maybe quite complex) example is the `gdnc` tool in the gnustep base library `Tools` directory.