Taming the daemon: Writing cross-platform service applications in FPC/Lazarus

Michaël Van Canneyt

February 4, 2007

Abstract

Programs that do not interact directly with the user and simply run in the background are commonly called services on Windows, or daemons on Unix systems. In this article a set of components is presented which allow to create daemons which work both on Unix and Windows systems. These components can be used in the Visual Designer of Lazarus, but can be used in plain FPC programming just as well.

1 Introduction

On Unix, Daemon applications exist since a long time. They're simply programs which are launched at system boot time, and which run till they are explicitly stopped. They don't interact directly with the user, but provide some local or remote service such as FTP or HTTP or logging messages. They have no special status: they can be programmed like any other unix command-line program, just they don't communicate directly with the user. (in fact, the standard file descriptors are usually closed right after program startup.

On Windows, services appeared in Windows NT. Windows introduced a special API for daemons (called services) and a program which manages the services: the service manager, which is in itself a service in the broad sense of the word.

In an earlier contribution by the author (**Jörg, can you insert a reference here ?, I wrote it in June 2003**), it was shown how services can be controlled and written using Delphi. In this article, it will be shown how to do the same using Lazarus/FPC, but in a cross-platform manner: Not all features will be available on Unix (it has no service manager), but the daemon will run and act in the same manner. The Windows-specific service manager component was also ported to FPC, and can be used in Lazarus.

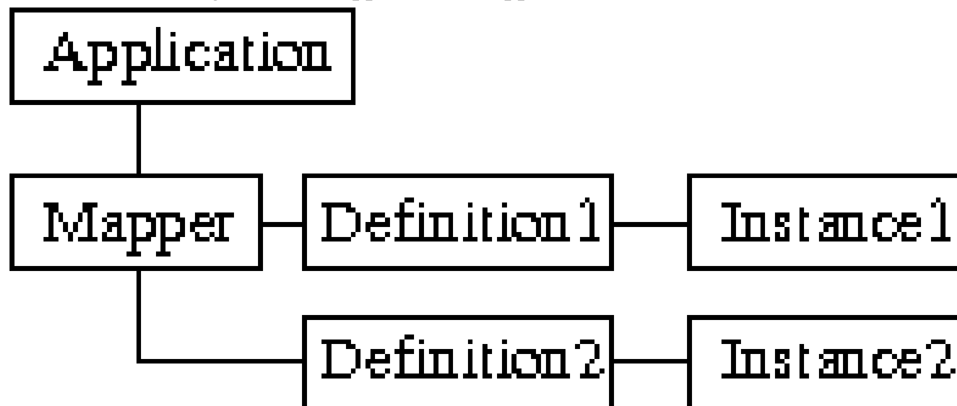
The term Daemon was chosen so as not to conflict with the Delphi use of the term: the implementation in Free Pascal is not compatible to Delphi, although it works rather similar. A compatibility layer is planned which will allow to port Services written in Delphi, to Free Pascal. To avoid overlap, the term Daemon was used to implement the components.

2 Architecture

When creating a daemon application in FPC, 3 classes must be used:

TCustomDaemon This is a `TDataModule` descendent which will contain the actual code for the daemon: it will do all the work. It has several methods, which must be overridden by descendents, and these methods should do the work.

Figure 1: The application, mapper and daemon instances



TCustomDaemonApplication This `TCustomApplication` descendent creates the necessary `TCustomDaemon` instances and then runs till the application gets a message to stop: In windows this is when the service manager is stopping the services, and in Unix, this is when a TERM signal arrives. It also provides a logging object which can be used to write diagnostic or error messages to the system log, both on Unix and Windows.

Normally, it is not necessary to manipulate this class.

TDaemonMapper This class tells the application object which daemons (or services) should be installed and run. It is possible to have a single `TCustomDaemon` class which is instantiated several times, each time doing the work of a single service.

A possible use of this is a webserver daemon: several services can be registered with windows, but they all do the same work: they listen to HTTP requests, but simply at different ports. A single `TCustomDaemon` descendent is enough to implement this. But it can also be two different services, running in the same binary: a FTP and HTTP server. Not a very common situation in Unix, but rather common under Windows.

For more Visual programming (in the Lazarus IDE), a fourth class is created: `TDaemon`. This is a simple `TCustomDaemon` descendent, which provides events for all major functions in `TCustomDaemon`.

The daemon mapper class contains one or more daemon definitions, symbolized in it's `DaemonDefs` property: a collection of daemon definitions (services) which (in the case of Windows) should be registered in the system. Each definition contains some properties to describe the service to windows, and a reference to a `TCustomDaemon` class which will be instantiated to handle the work of that service. This is shown in figure 1 on page 2.

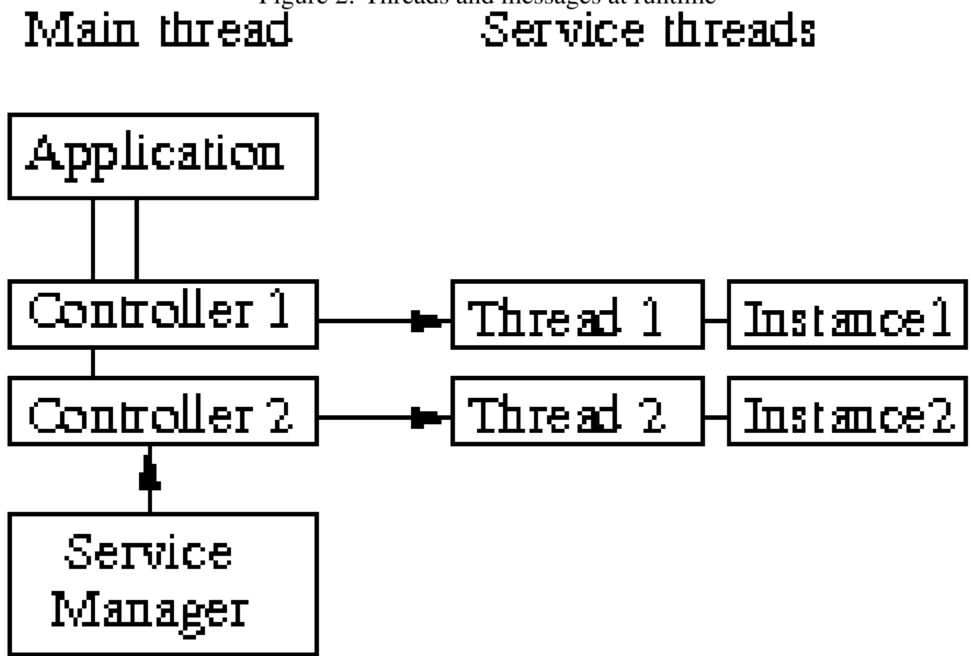
At runtime, the application works as follows:

- First, the command-line is analyzed, to see what needs to be done. There are 3 command-line options which are recognized:

  -i or `-install`, this will register the service definitions in Windows. On unix, this has (currently) no effect. For Delphi compatibility, the `/install` option can be used.

  -u or `-uninstall`, this will de-register the service definitions in Windows. On unix, this has (currently) no effect. For Delphi compatibility, the `/uninstall` option can be used.

Figure 2: Threads and messages at runtime



-r or `-run`, this will actually run the daemons. However, on Windows, this option should never be used: the service manager will provide it automatically when it needs to start a daemon.

- A daemon mapper is created. Only 1 daemon mapper should exist in the system. The Lazarus IDE support ensures this is so.
- For each daemon definition present in the daemon mapper, the appropriate `TCustomDaemon` descendent instance is created.
- What happens then depends on the command-line options: Either each of the instances is installed or uninstalled, or the daemons are started by the service manager (on windows).
- If any daemons were started, the application will wait for them to finish, otherwise it will exit.

When running the daemons, the daemons are started in a separate thread: all code of a daemon is run inside a separate thread, one thread per `TCustomDaemon` instance. The main application thread runs a loop till all threads finish, and then terminates.

Communication with the Windows Service manager happens through an extra class: `TDaemonController`. Under normal circumstances, it should never be needed to access the methods and properties of this class. However, people who want to extend the daemon support in Free Pascal, can make descendents of this class which have customized behaviour. When the application creates all daemon instances, it creates an instance of `TDaemonController` for each daemon. The windows service manager will communicate with this instance (this communication happens in the main thread of the application), and the controller instance delivers the message to the appropriate thread, so the daemon's reaction on the message runs inside the context of the thread. figure 2 on page 3 Shows this. The arrows show how messages are sent. The simple lines are ownership relations.

The functionality of the daemon must be implemented in a TCustomDaemon descendent. The following methods exist:

Start called when the daemon should start it's work. This method should return as soon as possible, and return `True` if the work was started succesfully.

Stop called when the daemon should stop it's work. This method should immediatly return, and return `True` if the work was stopped succesfully.

Shutdown called when the daemon should stop absolutely it's work. This method should immediatly return, and must return `True` if the work was stopped succesfully. This is called if the system is shutting down, and all services must be stopped.

Pause called when the daemon should temporarily suspends it's activity. This method should immediatly return, and return `True` if the work was suspended succesfully.

Continue called when the daemon should resume it's activity after it was paused. This method should immediatly return, and return `True` if the work was resumed succesfully.

Install Called when the daemon must be registered as a (Windows) service.

UnInstall Called when the daemon must be unregistered as a (Windows) service.

AfterUnInstall Called after the daemon was unregistered as a Windows service.

HandleCustomCode is called when a the service manager sends a non-standard control code (the code is passed in the `ACode` parameter to the `HandleCustomCode` call)

In rare cases, the `Execute` method can be overridden: use this if no separate thread should be started to run the daemon in.

3 Creating a daemon application in code

To create a daemon application without using the visual support for it in Lazarus, the following steps should be made:

1. Create a `TCustomDaemon` descendent, and override any of the `Start`,`Stop` etc. methods.
2. Register the descendent with `RegisterDaemonClass`, so the daemon mapper knows which class to instantiate.
3. Create a descendent of `TDaemonMapper` which initializes the `DaemonDefs` collection with correct daemon definitions.
4. Register the descendent with `RegisterDaemonMapper`, so the application class knows which mapper to instantiate.
5. Run the application.

To demonstrate this, a small daemon application is created. It does nothing useful, it sends a tick message every second to the system logger, and sends a message about every action to the system logger.

This action is achieved in a thread, which will be controlled by the daemon:

```
Type
TTestThread = Class(TThread)
  Procedure Execute; override;
```

```
end;

procedure TTestThread.Execute;

Var
  C : Integer;

begin
  C:=0;
  Repeat
    Sleep(1000);
    inc(c);
    Application.Logger.Info(Format('Tick : %d',[C]));
  Until Terminated;
end;
```

A most simple thread.

Then, the daemon which controls this thread is created:

```
Type
TTestDaemon = Class(TCustomDaemon)
  Private
    FThread : TTestThread;
    Procedure ThreadStopped (Sender : TObject);
  public
    Function Start : Boolean; override;
    Function Stop : Boolean; override;
    Function Pause : Boolean; override;
    Function Continue : Boolean; override;
    Function Execute : Boolean; override;
    Function ShutDown : Boolean; override;
    Function Install : Boolean; override;
    Function UnInstall: boolean; override;
  end;
```

The start method looks like this:

```
function TTestDaemon.Start: Boolean;
begin
  Result:=inherited Start;
  AWriteln('Daemon Start ',Result);
  FThread:=TTestThread.Create(True);
  FThread.OnTerminate:=@ThreadStopped;
  FThread.FreeOnTerminate:=False;
  FThread.Resume;
end;
```

As can be seen, it simply creates the TTestThread instance (with some housekeeping), and returns `True`. The `AWriteln` simply writes a message to the system logger.

The stop method is quite simple, the thread is told to suspend it's action:

```
function TTestDaemon.Stop: Boolean;
begin
```

```
  Result:=inherited Stop;
  AWriteln('Daemon Stop: ',Result);
  FThread.Terminate;
end;
```

The pause and continue functions hold no surprises:

```
function TTestDaemon.Pause: Boolean;
begin
  Result:=inherited Pause;
  AWriteln('Daemon pause: ',Result);
  FThread.Suspend;
end;

function TTestDaemon.Continue: Boolean;
begin
  Result:=inherited Continue;
  AWriteln('Daemon continue: ',Result);
  FThread.Resume;
end;
```

The various install methods simply write a message to the system log:

```
function TTestDaemon.Install: Boolean;
begin
  Result:=inherited Install;
  AWriteln('Daemon Install: ',Result);
end;
```

The other methods are similar, they are not essential in this context.

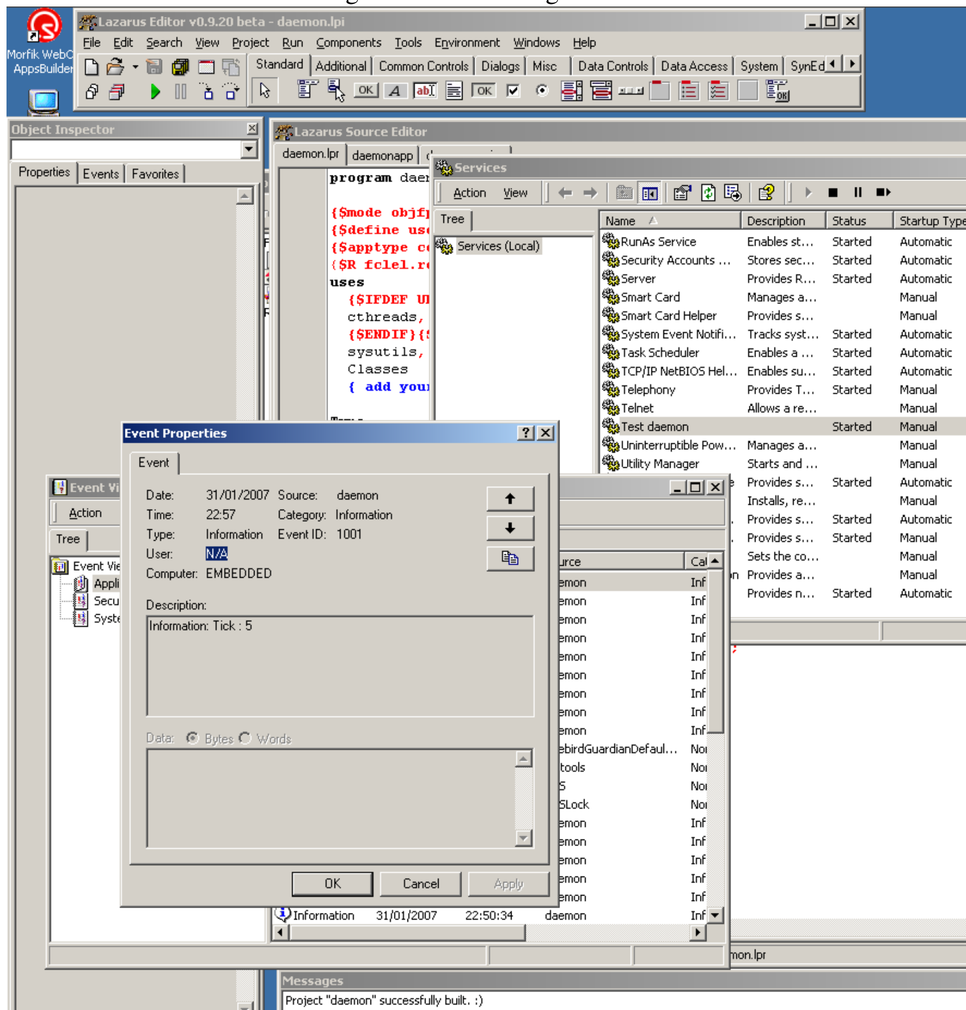
The mapper class is equally simple:

```
Type
  TTestDaemonMapper = Class(TCustomDaemonMapper)
    Constructor Create(AOwner : TComponent); override;
  end;

constructor TTestDaemonMapper.Create(AOwner: TComponent);

Var
  D : TDaemonDef;

begin
  inherited Create(AOwner);
  D:=DaemonDefs.Add as TDaemonDef;
  D.DisplayName:='Test daemon';
  D.Name:='TestDaemon';
  D.DaemonClassName:='TTestDaemon';
  D.WinBindings.ServiceType:=stWin32;
end;
```

It simply fills the definitions with a single daemon, and refers to the test class that was created above.

Remains to create the main program code:

Figure 3: The running daemon




```
begin
  RegisterDaemonClass(TTestDaemon);
  RegisterDaemonMapper(TTestDaemonMapper);
  Application.Title:='Daemon test application';
  Application.Run;
end.
```

And that's it. To test this program on windows, first run it from the command-line (or from within Lazarus) with the command-line option `-install`. It will then appear in the list of services (which can be consulted from the 'Services' menu entry under the Control panel, option 'Administrative Tools'.)

The service manager can then be used to start the service: use a right-click on the 'Test daemon' service to call the context menu in the service manager, and select 'Start'. To see the effect of the daemon, the 'Event viewer' application should be used (it can also be found among the Administrative tools). In the application log, the 'Tick' messages sent by the daemon can be found. This is shown in figure 3 on page 7.

Under unix, the daemon can be run straight away with the `-r` option.

4 Creating a daemon visually

The latest Lazarus versions in SubVersion have support for designing daemons in the IDE. To enable this support, the LazDaemon package must be installed in the IDE. If the package is supported, 3 items appear in the File - New dialog, under the heading "Daemon (service) applications":

Daemon (service) application this creates a new daemon application. It automatically creates one `TDaemon` instance and a `TDaemonMapper` instance. A registration procedure for both the daemon and daemon mapper classes are created automatically.

Daemon Module Creates a new `TDaemon` instance.

Daemon Mapper Creates a new TDaemonMapper instance. Normally this should not be used, as only one mapper should be created per application. Registering a second mapper will result in an error.

The daemon mapper can be used to define a daemon mapping: The `DaemonDefs` property can be edited completely in the object inspector.

The `TDaemon` module has events for all of the calls that exist in `TCustomDaemon`: `OnStart`, `OnStop`, `OnShutdown`, `OnPause`, `OnContinue`, `OnbeforeInstall`, `OnAfterInstall`, `OnBeforeUnInstall`, `OnAfterUnInstall`, `OnControlCode`.

To recreate the same daemon application with the Lazarus IDE is very simple. The `Daemon (service) application` option should be chosen, the daemon can be renamed to 'TestDaemon' and the mapper to 'TestMapper'. The `TestDaemon` can be added to the `DaemonDefs` property of the 'TestMapper'. (note that if the testdaemon module is renamed, the `DaemonClassname` property of the corresponding `DaemonDef` item is not updated) After that, the daemon can be coded. The `TTestThread` can be added to the TestDaemon unit, and the events can be coded. The following code shows the `OnStart` and `OnStop` events:

```
procedure TTestDaemon.TestDaemonStart(Sender: TCustomDaemon; var OK: Boolean);
begin
  OK:=True;
  FThread:=TTestThread.Create(False);
  FThread.OnTerminate:=@ThreadStopped;
  FThread.FreeOnTerminate:=False;
  FThread.Resume;
end;

procedure TTestDaemon.TestDaemonStop(Sender: TCustomDaemon; var OK: Boolean);
begin
  FThread.Terminate;
end;
```

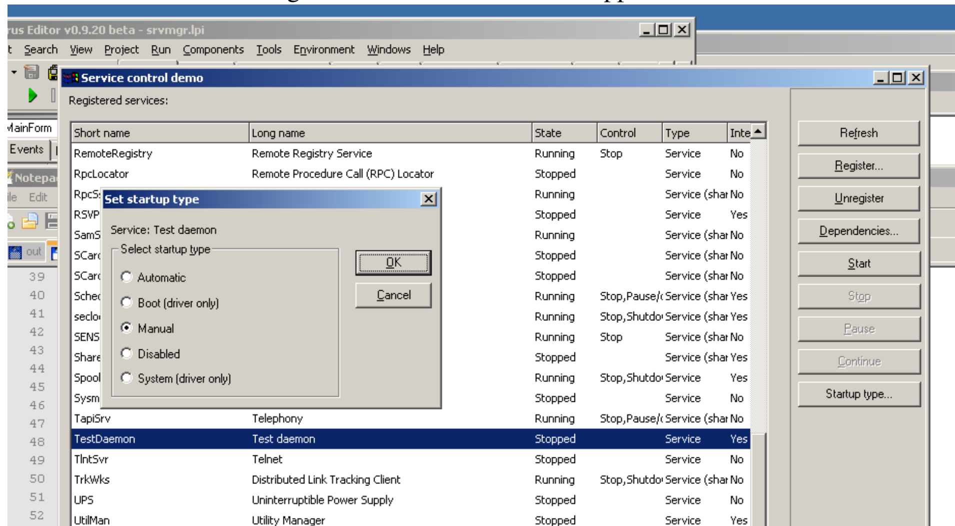
As can be seen, the code is similar to the code of the non-visual coded daemon. The rest of the code can be found on the CD accompagnying this issue.

Running the daemon on Linux with the following command:

```
./daemon --run
```

will produce an output like this in the /var/log/messages log file:

Figure 4: The Service Control application




```
Feb  1 22:27:48 home daemon: [Info] Daemon Test daemon current status: Start Pend
Feb  1 22:27:48 home daemon: [Info] Daemon Test daemon current status: Running
Feb  1 22:27:49 home daemon: [Info] Tick : 1
Feb  1 22:27:50 home daemon: [Info] Tick : 2
Feb  1 22:27:51 home daemon: [Info] Tick : 3
Feb  1 22:27:52 home daemon: [Info] Tick : 4
Feb  1 22:27:53 home daemon: [Info] Tick : 5
```

Which is the equivalent of what could be observed under Windows.

5 The Windows Service Manager component

In a previous article on using Delphi to write service applications for Windows, the `TServiceManager` component was introduced. This component was ported to Free Pascal, and added to the Free Pascal distribution.

The `TServiceManager` component was a component offering access to the Windows Service Manager API, using an Object Oriented approach: it allowed to retrieve the definitions and status of all services, register new services, plus offered the functionality which the Service Manager application of Windows offers: starting and stopping services.

The sample application which was presented then (a copy of the Services control panel applet), has been ported to Lazarus: This basically meant importing it via the Lazarus import functionality under the "Tools" menu, and recompiling it. The interested reader can check the sources on the CD-ROM accompagnying this issue. In figure 4 on page 9, the sample application can be seen, after compilation using Lazarus.

6 Conclusion

Creating cross-platform services is made easy using the `daemonapp` unit. The architecture is quite extendible (a more Delphi-compatible extension is under way). It is not entirely finished: Although the daemon can be paused, continued and stopped on Unix by sending it the classical Unix signals STOP, CONT and TERM, it does not have quite the

same semantics as the Windows services: the onpause and oncontinue events will not be triggered, as the kernel effectively pauses the application. Work is being done to create a small control layer which provides the same possibilities as the windows version. However, for most applications, the current functionality is sufficient to create services that can run and do their work on both Windows and Unix platforms.