

Re-Inventing the NullStream

Jonathan Shapiro, Ph.D.
The EROS Group, LLC

December 6, 2007

Abstract

Building your first Coyotos application may seem a bit daunting. This tutorial describes how to build a first, simple, application that implements an existing interface specification. Subsequent tutorials address more complicated aspects of application construction, including specifying new interfaces and handling multiple interfaces in a single application.

1 Objectives

In this tutorial, we will construct a duplicate of the NullStream program. While NullStream doesn't do very much in and of itself, it does illustrate how to implement the skeleton of a basic Coyotos service process: one that accepts requests on an interface, dispatches those requests to handler procedures that implement the methods of the interface, and sends replies to client(s). The tutorial also shows how to package an application in a constructor, allowing multiple copies of the application to be instantiated.

When you have completed this tutorial, you will understand:

- The role of interfaces in the Coyotos system.
- How to use `capidl` to generate a template of a service application.
- How endpoint IDs and protected payloads are used.
- How to fill in the missing parts of the template to produce an application that actually does something.

2 Role of Interfaces

Most processes in a Coyotos system exist to implement one or more interfaces. An **interface** is a collection of types and methods that are implemented by some capability. Every capability in Coyotos implements exactly one interface. Every **entry capability** names (via an Endpoint) some process that implements the behavior of that capability's interface.

When an entry capability is invoked, the receiving process (the "server") receives the **endpoint identifier** (`endpointID`) from the endpoint and the **protected payload** field from the capability. The server interprets these two values to determine what "object" is being invoked, what interface is being used to invoke that object, and what permissions the invoking party has. By convention, the endpoint identifier is used to select (name) some object implemented by the server, and the protected payload is used to specify permissions. This practice is not universal, but it is often a good starting point.

A single process may implement the behavior of multiple entry capabilities. These capabilities may have identical interfaces (e.g. many capabilities implemented by the file system object implement the `coyotos.file` interface) or different interfaces (e.g. the constructor implements a builder's interface `coyotos.Builder`, a requestor's interface `coyotos.Constructor`, and a verifier's interface `coyotos.Verifier`).

3 Generating a Template

If your server implements a new interface, the first step would be to create an interface specification (which is the subject of the next tutorial). In this tutorial, we will be building a process that implements the methods of an existing, standard interface: `coyotos.InputStream`. The `coyotos.InputStream` interface defines a character-based stream protocol. The specification for this interface can be found in the *Coyotos Core Domain Interfaces* document under `coyotos.InputStream`.

Once you have determined what interfaces your program must provide, the next step is to generate a template for your server. This is done using the `-t` (template) option to `capidl`, giving the IDL file name for each interface that you want your server to be able to implement:

```
capidl -a i386 -t -I ../coyotos/usr/include/IDL \
      ../coyotos/usr/include/IDL/coyotos/InputStream.idl
```

The `-I directory` adds an import search path. Each interface that is imported (transitively) from one of your input files will be resolved by searching the import search path directories from left to right. The first file found will be used. The `-a i386` indicates that we want to generate output suitable for the `i386` target architecture.

The template generator will produce a file `template.c` that contains:

- A "grand server union" declaration. This declares the input structure that is used by the server's `ProcessRequest()` loop. It contains one element for each interface that your server will support, plus additional elements for low-level manipulation by the CapIDL-generated server code.
- A *declaration* for the `choose_if()` function. This is a function that you must implement. It accepts as input an endpoint ID and a protected payload value. It should return the unique interface (the IKT value) for the interface corresponding to that endpoint ID and protected payload value.
- A minimal *definition* of an IDL server environment structure. If your server must carry information down into the handler functions that cannot be obtained from global variables, that information should be added to the server environment structure.
- A `ProcessRequests()` procedure. This procedure sits in a loop that waits for a new request, processes it, and returns a reply. This procedure is automatically generated for you by `capidl`. The main body of this loop calls your `choose_if()` function to determine what interface should be served for this request. Given the interface, it then determines the method invoked by examining the incoming message. It demarshalls the incoming request, calls a handler procedure, marshalls the result of the handler procedure, and returns a response.

Much of *your* effort in a typical server will lie in implementing the required handler procedures.

- A minimal initialization procedure. You will almost certainly need to modify this procedure.
- A `main()` procedure.

We strongly recommend that you rename the template file.

4 Creating an MKI File

Before you can compile your template file, you will need a `mkimage` description file (**MKI file**). The `mkimage` utility builds Coyotos system images. It's role is similar to that of an object file linker. Just as an object file linker (e.g. `ld`) links your object files together and resolves references between them, `mkimage` loads program images, creates processes, and links them together by inserting the capabilities that will be needed for one process to invoke the next.

Your source file describes what your program will do when it is run. Your MKI file describes how your program should be incorporated into a Coyotos system image, and how it should be connected to other programs in the image.

If you are building a new program, you will probably have a test case as well. Each of these will have its own MKI file.

4.1 An Example

Coyotos does not have any system call that is analogous to UNIX *fork* or *exec*. The mechanism for instantiating processes in Coyotos is to use a **constructor**. The main purpose of an application's MKI file is to fabricate the constructor for the application, and to export a capability to that constructor.

Most of the MKI files that you will see for applications look pretty similar. Here is a typical example. The parts you will typically need to change are shown in bold:

```
1 module coyotos.stream.Null {
2   import rt = coyotos.RunTime;
3   import image = coyotos.Image;
4   import ctor = coyotos.Constructor;
5
6   header capreg APP {
7     Log = rt.REG.APP0,
8     MyEntry
9   };
10
11  header enum TOOL {
12    Log = rt.TOOL.APP0
13  };
14
15  def bank = new Bank(PrimeBank);
16
17  def space = image.load_elf(bank, "coyotos/stream/Null");
18
19  def tools = ctor.fresh_tools(bank);
20  tools[TOOL.Log] = KernLog();
21
22  export def constructor =
23    ctor.make(bank, space, tools, NullCap());
24 }
```

4.2 Line by Line

Line 1 gives the **module name** for this MKI file. This is simply a unique module name that can be used to import this module file. To avoid name collisions, it is good practice to use a module name in the Java style, beginning by reversing your domain name and then appending the name of the program. It is okay to extend beneath your corporate domain name, for example `com.YourDomain.test.program` or `com.YourDomain.Product.program`, but think ahead! These names tend to become carved in stone very quickly as various parts of your build environment come to depend on them. Modules that are part of the Coyotos core distribution do not follow this convention, which is why the module name here is `coyotos.stream.Null`.

Line 17 gives the name of your application's binary file. This expresses where to find the binary in the install tree. In most cases, the binary file name will relate closely to the module name.

Most programs require capabilities in order to run usefully. In this example, the program needs to be able to add messages to the kernel log. In order to do this, it is necessary to add this capability to the "tools" that are

supplied to the program. Certain tools are commonly provided by the constructor logic (see the enumeration in `coyotos/RunTime.mki`), so a tools page is mandatory and must be allocated (line 19). The first available “slot” in the tools page is defined by `coyotos.RunTime.TOOL.APP0`, which is why we start the tools enumeration at that value. In order to obtain access to that value, we import the MKI file defining the slots used by the runtime at line 2. Having determined what the slot numbers are, we populate the relevant slots of the tools page at line 20.

Finally, your program will typically require some number of working capability registers. Because some registers are reserved for use by the runtime, it is helpful to define register number constants in the MKI file (line 6). Once again, we start the constants at a starting point defined in the MKI file for the runtime system.

Two last comments. The `header` keyword indicates that the corresponding definitions will be emitted to a header file when you invoke `mkimage` with the `-H` option. This allows your C program to use the constants defined here, which helps ensure that they remain in sync. The `export` keyword indicates that the corresponding definition will be available for other MKI modules to reference. In this case, the only symbol we are exporting is for the constructor of the program.

In order to run, your process will require various capabilities, and we will need to fabricate a constructor for it. A **constructor** is a utility program that knows how to start new instances of some program. Each application has a constructor that creates instances of that application. The `mkimage (.mki)` file describes how the pieces of your application should go together.

It is typical for each constructor to run from a distinct space bank. This allows the constructor to be destroyed later by destroying the space bank. This is why almost every applicaion MKI file will include a line allocating a new bank (line 15) and use that bank for all subsequent operations that allocate storage.

At line 17, we load the binary image of the program and create an address space containing that program. More precisely, we create a memory fault handler that knows how to populate an address space by loading it on demand from an ELF-based binary file.

Finally, at line 22, we create the constructor process that knows how to instantiate our new program.

5 Creating the Makefile

To compile your new application, you will need to create a script for the make program:

```
1 default: package
2 COYOTOS_SRC=../../../../../
3 CROSS_BUILD=yes
4
5 INC=-I. -I$(COYOTOS_SRC)/../usr/include -I$(BUILDDIR)
6 SOURCES=$(wildcard *.c)
7 OBJECTS=$(patsubst %.c,$(BUILDDIR)/%.o,$(wildcard *.c))
8 TARGETS=$(BUILDDIR)/Null
9
10 include $(COYOTOS_SRC)/build/make/makerules.mk
11
12 ENUM_MODULES=coyotos.stream.Null coyotos.TargetInfo
13 ENUM_HDRS=$(ENUM_MODULES:%=$(BUILDDIR)/%.h)
14
15 $(ENUM_HDRS): $(MKIMAGE) | $(BUILDDIR)
16     $(RUN_MKIMAGE) --MD -H $(BUILDDIR) $(ENUM_MODULES)
17
18 $(OBJECTS): $(ENUM_HDRS)
19
20 install all: $(TARGETS)
21
22 install: all
```

```

23     $(INSTALL) -d $(COYOTOS_ROOT)/usr/domain/coyotos/stream
24     $(INSTALL) -m 0755 $(TARGETS) $(COYOTOS_ROOT)/usr/domain/coyotos/stream
25
26     $(BUILDDIR)/Null: $(OBJECTS)
27     $(GCC) $(GPLUSFLAGS) $(OBJECTS) $(LIBS) $(STDLIBDIRS) -o $@
28
29     -include $(BUILDDIR)/.*.m

```

Almost all of this file is “boiler plate”, and it is likely that we will be trying to simplify it in the future. We will discuss here only the lines that are specialized for your program.

Line 8 defines the **TARGETS** variable, which gives the binary name of your program. By convention, all Coyotos makefile scripts place output in a subdirectory, which is why the program name is prefixed by \$(BUILDDIR).

Our program requires constant definitions to be supplied by various modules. We need to generate a header file containing those constants so that we can include it. Line 12 specifies the MKI module names for each of these modules. The header names are derived automatically from the module names by line 13, and line 18 states that our object files require these headers in order to be compiled.

Lines 23 and 24 ensure perform installation. We first ensure that the target directory exists, and then install our program into that directory. Note that the “reverse domain name” convention applies to installed locations as well.

Line 26 says how to build our program. While we could have re-used the **TARGETS** variable here, it is occasionally useful to build more than one program in a single makefile script, so our practice tends to be to specify the program explicitly.

Line 29 is *very important*. Many steps of the build process generate automatic dependency tracking files into the build directory. These are used by make to ensure that all needed items get build when something changes. If you delete this line, your builds are unlikely to remain up to date.