Building Infrastructure for DROPS (BID) Tutorial

Jork Löser Ronald Aigner
l4-hackers@os.inf.tu-dresden.de

April 11, 2012

Abstract

BID is a collection of conventions and files that help in the build (compile) process of applications for DROPS. Among the features are: compiler and linker selection, automatically generated rules for linking a binary and archiving a library, dependency generation, installation, documentation compilation.

BID ensures a defined behavior of packages that is needed for the automatic building process of the project.

This document describes how to write a package.

1 Assumptions

This tutorial is to help the reader (you) to write applications using BID. We assume you to know the basic Unix commands and concepts of its file system. For understanding the example programs, you should be familiar with C.

Compiling packages for DROPS requires a specific compilation environment to be set up. This environment contains IDL compilers, special versions of a standard C library, various Linux kernel sources, compiled DROPS packages and others. A tutorial on how to set up this environment can be found in [2]. We assume the environment being set up correctly.

We assume that you checked out the L4Env [1] module from the DROPS download page [3] and build the environment. To test your package, you will also need the Fiasco module. If you are at the Operating Systems Group here at TU Dresden with access to the file server *os.inf.tu-dresden.de*, most of these files are installed at /home/drops.

2 Finding around

Lets assume your base directory where your DROPS-sources are is `$(HOME)/src`. After checking out the files using CVS, you end up with a couple of directories:

l4/	the L4 directory, later referenced as $(L4DIR).
l4linux-2.6/	L4Linux 2.6

There might be others as well, likely Linux kernels, which are needed by some DROPS packages. Within the L4 directory, you will find

l4/kernel	sources of L4 micro kernel implementations, primarily FIASCO.
l4/pkg	base directory of all DROPS packages.
l4/tool	tools for compilation, among them the IDL compiler DICE.
l4/mk	BID macros to control compilation and installation.
l4/doc	base directory for documentation, will also contain compiled packet documentation later.
l4/include	base directory for include files exported by packages.
l4/lib	base directory for compiled libraries.
l4/bin	base directory for compiled binaries (might be a symbolic link).

3 Your first hello world

As DROPS is a multi-server system, programs are seen as servers, offering some services to other programs. Our infrastructure reflects this by using packages for managing different programs. A package typically contains a server, libraries and include files to be exported.

Your first hello world program does not offer any service to clients, nonetheless we say it is a server (without any clients, though). Let's name it hiworld. You create the package by creating a directory called `hiworld` in `l4/pkg`:

```
~/src/l4/pkg> mkdir hiworld
```

This directory is called *package dir* and will be referenced as `$(PKGDIR)`. Switch to the directory and use the template provided by BID to create the basic directory structure together with some Makefiles:

```
~/src/l4/pkg/hiworld> ../../mk/tmpl/inst
```

This results in a couple of directories and a makefile to be created. The makefile is ready to use and builds your whole package when doing `make`. A first look into it shows the basic scheme of all BID-makefiles: After setting `$(PKGDIR)` and `$(L4DIR)` you add your commands, and finally a macro-file is included. We refer to the macro-file as *role file*, as this file determines a distinct role of the current directory. The purpose of `$(PKGDIR)` is just to propagate various make commands, which is achieved by the subdir role. Fortunately, its default behavior meets our needs, and we need no additional commands.

As you want to create a server, switch into the `server/` directory. There is already another makefile, very similar to the one you just looked at. Note the modified `$(PKGDIR)` assignment, as the package dir is one level up now. Go ahead into the `src/` directory, where you finally place your source code.

The makefile prepared in this directory creates executable binaries. Optionally, it uploads them to your file-server (tftp at TUD), so you can boot it right afterwards over the network. If you did not do it already, this is a good time to fix the paths for later uploading the binaries to your file server (see Section 5 for details).

The makefile in `server/src` expects your code in the file `main.c` (which we could change, of course), so edit this file.

```
~/src/l4/pkg/hiworld/server/src> xemacs main.c
```

The file prepared already has an empty `main()` function. Add a `printf("Hi world")`:

hiworld/server/src/main.c

```
int main(int argc, char**argv){
  printf("Hi world\n");
  return 0;
}
```

Compile the program:

```
~/src/l4/pkg/hiworld/server/src> make O=/path/to/builddir
```

This will create a directory (`OBJ-x86_586-l4v2/`) in the build directory where all compiled files will go. You will find the compiled binary in this directory, its name is the name of the package: `hiworld`. The binary is also installed into `$(L4DIR)/bin`. If you did the setup for installing the binaries (see Section 5), it should be installed on your file server this way, and hence is ready for booting over the network already. To execute it, boot L4 together with the essential servers. Use the following `menu.lst file` after adapting the paths to the binaries.

menu.lst

```
title Hi world
  kernel (nd)/tftpboot/yourname/bin/x86_586/l4v2/bootstrap
  modaddr 0x02000000
  module (nd)/tftpboot/yourname/bin/fiasco -nokdb -nowait
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/sigma0
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/roottask
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/log
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/names
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/dm_phys
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/hiworld
```

If all went right (you installed an L4 version 2 micro-kernel named *fiasco* into `$(L4DIR)/bin`, and installed all the other binaries as well; your boot loader found all the files; your test machine executed the micro kernel and the applications), you should see the 'Hi world' near the bottom of the screen attached to your test node.

You may also run your application using Fiasco-UX (refer to Section 6 for instructions on building and using). The respective start script is:

hello

```
#! /bin/bash

. ${0%/*}/generic.inc

fiasco \
        -l names                \
        -l log                  \
        -l dm_phys              \
        -l hiworld
```

In this section we

- created a new package
- wrote code printing "hi world"
- compiled and installed the program, so the file-server could find it
- booted L4 to execute our native hello world program

4 Hello world server

In this section, we create a server that reacts to commands of a client. For generating the code for communication between server and client we use an IDL compiler.

Our server should offer two services, one is to output the well-known “Hi world” string, and another is to return the number of calls to the former function.

4.1 IDL file

We start with writing a corresponding IDL file. Therefore, switch into `$(PKGDIR)/idl/` and create the file `hiworld.idl`:

hiworld/idl/hiworld.idl

```
interface hi {
        void print(void);
        int  count(void);
};
```

Modify the prepared makefile to assign the IDL file to the variable IDL, and to ask the IDL compiler to generate server skeletons. The skeletons will help us later to insert our code:

hiworld/idl/Makefile

```
...
IDL       = hiworld.idl
IDL_FLAGS = -t
...
```

Generate the C sources by calling make:

```
~/src/l4/pkg/hiworld/idl> make O=/path/to/build
```

If your IDL compiler was installed correctly, you end up with the directory `OBJ-x86_586-l4v2` in the build directory containing the C sources for IPC communication between client and server. For details on the IDL file syntax, the meaning of the command line switches or the generated source code, please see the documentation of the IDL compiler *dice*, which you can build yourself in `$(L4DIR)/tool/dice/doc/` or find it in [4].

4.2 Server side

Next we implement the server-side functionality. Copy the generated skeleton file into the already known `server/src/` directory.

```
~/build/pkg/hiworld/idl/OBJ-x86_586-l4v2> cp hiworld-template.c \
    ~/src/l4/pkg/hiworld/server/src/server.c
```

Modify the functions so they do what we want, write code that registers at the DROPS name server and calls the main server loop:

hiworld/server/src/server.c

```
#include <stdio.h>
#include <l4/names/libnames.h>
#include "hiworld-server.h"

char LOG_tag[9]="hiserver";
static int count;

void
hi_print_component(CORBA_Object _dice_corba_obj,
                   CORBA_Environment *_dice_corba_env){
    printf("Hi world\n");
    count++;
}

int
hi_count_component(CORBA_Object _dice_corba_obj,
                   CORBA_Environment *_dice_corba_env){
    return count;
}

int main(void){
    if(names_register("hiworld")==0){
      printf("Error registering at nameserver\n");
      return 1;
    }
    hi_server_loop(0);
}
```

We still have to tell BID that we want to compile another binary, containing the IDL server code and our implementation. Modify the makefile the following way:

hiworld/server/src/Makefile

```
PKGDIR          ?= ../..
L4DIR           ?= $(PKGDIR)/../..

TARGET          = hiserver
DEFAULT_RELOC   = 0x01800000

SRC_C           = server.c
SERVERIDL       = hiworld.idl

include $(L4DIR)/mk/prog.mk
```

You are ready to compile the server. BID knows where to find the files generated by the IDL compiler and arranges the include paths and additional object files for you.

```
~/src/l4/pkg/hiworld/server/src> make O=/path/to/build
```

If there are any problems, they are likely caused by typos, as your installation has already been proved correct in the previous steps.

4.3 Client helper library

Let us build a client library that encapsulates calls to the hello world server. The library will be made available for other packages by BID, so any package can use the library for easy communicating with the hello world server.

The functions exported by the library will be declared as prototypes in an include file. The include file will also be made available for other packages by BID. Let us create the include file first. Switch to the `$(PKGDIR)/include/` directory and create the new file `hiworld.h`:

hiworld/include/hiworld.h

```
#ifndef __HIWORLD_INCLUDE_HIWORLD_H_
#define __HIWORLD_INCLUDE_HIWORLD_H_
#include <l4/hiworld/hiworld-client.h>

#define hiworld_name "hiworld"
int hiworld_print(void);
int hiworld_count(void);

#endif
```

The prepared makefile installs all include files in the directory tree, so that other packages can use them. Our newly created file will be accessed with `#include <l4/hiworld/hiworld.h>` later:

```
~/src/l4/pkg/hiworld/include> make O=/path/to/build
```

Client libraries are typically build in the `$(PKGDIR)/lib/` directory of a package. Like in the server directory, there is a `src/` subdirectory. Switch into it and create the new file `encap.c` containing the encapsulation code:

hiworld/lib/src/encap.c

```
#include <l4/names/libnames.h>
#include <l4/env/errno.h>
#include <l4/sys/consts.h>
#include <l4/hiworld/hiworld.h>

static l4_threadid_t server_id = L4_INVALID_ID;
static CORBA_Object server = &server_id;

//! Request netserver id at nameserver
static int check_server(void){
    if (l4_is_invalid_id(server_id)){
        if (!names_waitfor_name("hiworld",&server_id,10000)) return 1;
    }
    return 0;
}

//! print the string
int hiworld_print(void){
    CORBA_Environment env = dice_default_environment;

    if (check_server()) return -L4_EINVAL;
    hi_print_call(server, &env);
    return -env._p.ipc_error;
```

```
}
//! get the string
int hiworld_count(void){
    CORBA_Environment env = dice_default_environment;
    int count;

    if (check_server()) return -L4_EINVAL;
    count=hi_count_call(server, &env);
    if(!env._p.ipc_error) return count;
    return -env._p.ipc_error;
}
```

The makefile in this directory is already prepared to create libraries. We still have to specify which files go into it. BID also knows about generated IDL client files, the according make variable is `$(CLIENTIDL)`:

hiworld/lib/src/Makefile

```
PKGDIR          ?= ../..
L4DIR           ?= $(PKGDIR)/../..

TARGET          = lib$(PKGNAME).a
SRC_C           = encap.c
CLIENTIDL       = hiworld.idl

include $(L4DIR)/mk/lib.mk
```

Finally, build the library and install it, so others can use it:

```
~/src/l4/pkg/hiworld/lib/src> make O=/path/to/build
```

4.4 Client side

The last piece is the client program actually triggering the actions at the server. We see it as an example on how to use our hello world server, and consequently place it into the `$(PKGDIR)/examples/` directory tree. Create a new subdirectory there:

```
~/src/l4/pkg/hiworld/examples> mkdir client
```

Switch into it and create the source file `main.c` for sending requests to our server:

hiworld/examples/client/main.c

```
#include <stdio.h>
#include <l4/hiworld/hiworld.h>

char LOG_tag[9]="hiclient";

int main(void){
  int count;

  hiworld_print();
  count = hiworld_count();
  if(count>=0){
      printf("Hello world server returned %d.\n", count);
```

```
  } else {
      printf("Some problem with the hello world server occurred\n");
  }
  return 0;
}
```

Create a makefile that compiles a binary, and tell BID to link the client library created in Section 4.4.

hiworld/examples/client/Makefile

```
PKGDIR          ?= ../..
L4DIR           ?= $(PKGDIR)/../..

TARGET          = hiclient
SRC_C           = main.c
LIBS            = -lhiworld
DEFAULT_RELOC   = 0x01100000

include $(L4DIR)/mk/prog.mk
```

Compile the program:

```
~/src/l4/pkg/hiworld/examples/client> make O=/path/to/build
```

4.5 Execution

To execute the code, you need to boot L4 together with the essential servers, your hello world server and the hello world client. Adapt and add the following text to the `menu.lst` file you already created in Section 3:

menu.lst

```
title Hi world, server version
  kernel (nd)/tftpboot/yourname/bin/x86_586/l4v2/bootstrap
  modaddr 0x02000000
  module (nd)/tftpboot/yourname/bin/fiasco -nokdb -nowait
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/sigma0
  module (nd)/tftpboot/yourname/bin/x86_586/l4vs/roottask
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/log
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/names
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/dm_phys
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/hiserver
  module (nd)/tftpboot/yourname/bin/x86_586/l4v2/hiclient
```

Now boot the new configuration on your test machine. If all went right you should see the “Hi world” issued by the server and a “Hello world server returned 1.” issued by the client near the bottom of the screen attached to your test machine.

4.6 Closing remarks

With the current version of the makefile in the `examples/` directory, our example is only built upon explicit make request in `examples/client/`. You should modify it to propagate a `make` in the package directory, as it is done for all the other directories automatically. Thus it becomes:

hiworld/examples/Makefile

```
...
TARGET = client
...
```

Now, you could do a `make` in `hiworld/` to rebuild your whole package (probably nothing actually gets built, as everything should be up to date). Or, you could do a `make clean` in `hiworld/` to delete all generated intermediate files in your package and only keep the generated library and the binaries. A `make cleanall` removes these too.

Note, that BID takes care of build dependencies very carefully. If a source-file is changed, a `make` in this directory rebuilds everything in this directory that depends on this file. BID also notices modified makefiles and client libraries being linked to binaries.

If you trigger a make at the root of a directory tree, BID generally brings the tree up to date, but never builds anything outside this tree. For example, you build a package by doing a `make` in its package directory. However, if it requires include files of libraries from other packages, you must either build them before, or go into the `$(L4DIR)/pkg` directory and trigger the make there.

In this section we

- used an IDL compiler to generate IPC communication code
- used an include file to propagate prototypes and defines to users of our package
- created a server implementing the server side of the IDL
- created a library allowing clients to communicate with our server
- created a client program using that library to communicate with the server
- learned about dependencies and recursive make invocation

5 Installing binaries for booting

BID can automatically install your compiled binaries so you can boot them later using the network. Therefore, after compilation binaries are stripped and copied into `$(L4DIR)/bin` per default. Make this a symbolic link to your actual file server directory, and you are set.

At TUD, this requires two steps: Ensure the tftp server finds the binaries within your home directory. Then, ensure the binaries are installed into this directory. Thus at *os.inf.tu-dresden.de*:

```
os:/home/yourname> mkdir -p boot
os:/home/yourname> ln -s boot /tftpboot/yourname
```

(The latter step may require administrator rights, ask the administrator in this case.) At your *workstation*, where os-home:/home is mounted (we assume /home to be the mountpoint):

```
.../build> ln -s bin /home/YOURNAME/boot/bin
```

6 Using Fiasco-UX to run L4 applications

Fiasco can be build to run as Linux user application – Fiasco-UX. The steps to build Fiasco-UX you may obtain from the Fiasco-UX homepage [5].

To simplify running application we provide a set of scripts that wrap the typical options of Fiasco-UX. Please check out `l4/tool/runux`. Change into that directory and edit the file hello.

hello

```
#! /bin/bash

. ${0%/*}/generic.inc

fiasco \
        -l names                    \
        -l log                      \
        -l dm_phys                  \
        -l hiworld
```

This script will start Fiasco-UX with the basic servers required by the hello binary. These servers you have already seen in the Grub boot list. To start the hiserver/hiclient example simply replace the line containing hiworld with respective lines for hiserver and hiclient.

7 FAQ

Q: We might want to specify the exact CPU types we want to build a target for. example: watchdog example of l4util. However, this must be linked against the according libraries, which may be not available. Do we have CPU-dependent libraries, that should be linked against other libraries? Or: Do we want to provide libraries that should be used with different CPU types?

A: No, we have one library for every CPU type.

Q: We want to build several packages with *and* without l4api support, such as the l4util package. The parse_cmdline stuff is independent of the binding, l4_sleep is not.

A: This must be split into two libraries, one with l4api support and one without. Use the system-specific `$(TARGET_system)` variables to define which library should be built for which system. Then, use the target-specific `$(SRC_C_target)` variables to define which source-files go into which library.

Q: I want to use an own CRT0, and use ”CRT0=mycrt0.o”. But mycrt0.o is not built automatically.

A: This is right. The CRT0 variable references an external object already built. You probably want to use ”SRC_S=mycrt0.S” (if its assembler). Set ”CRT0=” to prevent the standard crt0's to be linked.

Q: I need my own linker to link a specific binary. But BID uses $(CC).

A: Set the CC variable target-dependent. If your target is mytarget and your favorite linker is mylinker, use ”mytarget: CC=mylinker”.

Q: Why is mconfig not supported?

A: `mconfig` is another Linux configuration tool. It has no help, and provides only one menu-like user-interface, which is not very handy.

8 References

[1] L4Env. Documentation at `http://os.inf.tu-dresden.de/l4env/`.

[2] *DROPS Building HowTo*, 2006. Available at `http://os.inf.tu-dresden.de/l4env/doc/html/drops-building/building.pdf`.

[3] *DROPS Download Page*, 2006. Available at `http://os.inf.tu-dresden.de/drops/download.html`.

[4] DICE *User's Manual*, 2006. Available at `http://os.inf.tu-dresden.de/dice/manual.pdf`.

[5] FIASCO-*UX Homepage*, 2006. Available at `http://os.inf.tu-dresden.de/fiasco/ux/`.