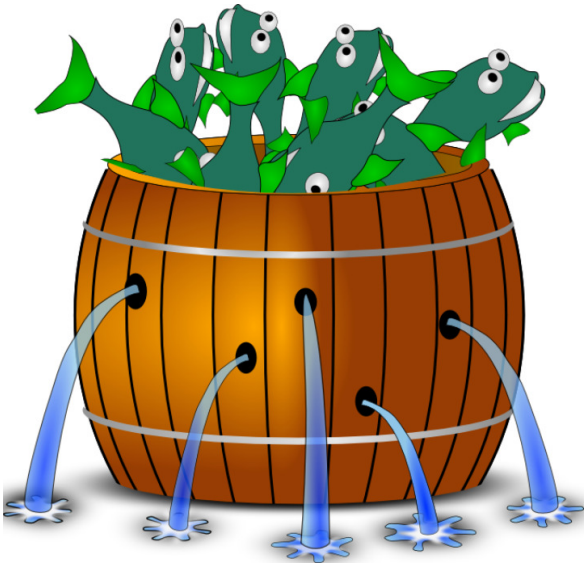
Barrelfish Project
ETH Zurich



Barrelfish Practical Guide

Barrelfish Technical Note 018

Team Barrelfish

29.07.2021

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
http://www.barrelfish.org/

Revision History

Revision	Date	Author(s)	Description
1.0	06.12.2013	The Barrelfish Team	Initial version
2.0	29.07.2021	LH	Updated to recent version

Contents

Chapter 1

Introduction

This note describes how to build and boot Barrelfish on 64-bit PC hardware, which is the default configuration for Barrelfish. It then goes on to work through a simple "hello, world"-style application which illustrates client-server programming in C on Barrelfish, and the use of the message-passing subsystem.

Information on how to compile Barrelfish for other platforms, in particular ARM, are found in other documents. However, the application programming section in this guide is still relevant.

Please refer to the README in the source code root for a list of supported hardware, build dependencies and how to compile the initial Barrelfish version.

Chapter 2

Writing Hello World application

2.1 Writing A Sample Barrelfish Application

Here we show how to write a simple program on Barrelfish. Refer to `usr/examples/...` for the hello world we present here as well as further examples.

2.1.1 A Simple Hello World Program

The source for user domains are stored under

`/usr`

and its subdirectories.

A simple hello world domain consists of a source file, and a Hakefile. e.g.,

```
usr/examples/xmpl-hello/
                        hello.c
                        Hakefile
```

For a simple hello world app, the source is trivial: a simple printf in main will suffice (note that for output printf either uses a debug syscall to print to the serial output, or a proper serial server if one is available).

```
#include <stdio.h>
int main(void) {
    printf("Hello World\n");
    return 0;
}
```

The Hakefile will describe how our example application is being built. For the details on how to write Hakefiles, refer to the Barrelfish Technical Note TN-003. For now, it suffices to understand that the C files for our application are hello.c and the built binary will be stored in `examples/xmpl-hello` in the build tree.

```
[ build application { target = "examples/xmpl-hello",
                      cFiles = [ "hello.c" ]
                    }
]
```

To inform the build system of our newly created Hakefile, we must call hake.sh in the build directory (for example using `mkdir build && ../hake/hake.sh -s ..  -a x86_64`). The build system will discover changed Hakefiles automatically, but it wont notice *new* ones. If we have added new Hakefiles

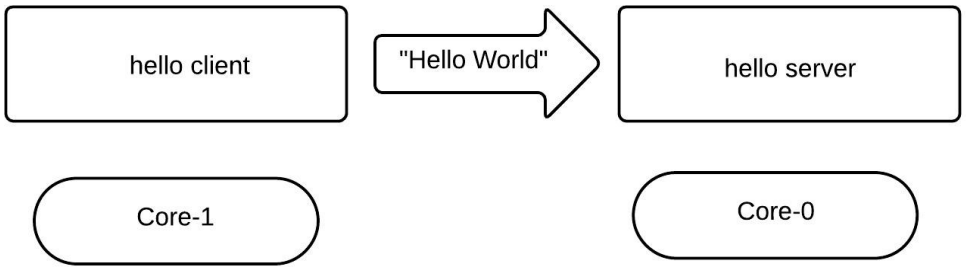


Figure 2.1: Hello world application overview

we have to re-run hake.sh (or use the shortcut `make rehake`). The hake run will create Config.hs and a Makefile.

At this point, we can explicitly build our sample application by running `make x86_64/sbin/examples/xmpl-hello`. The next step is to add a dependency from a simulator target to our example application, such that it gets built and bundled automatically. The platforms are described itself in a Hakefile, which is located in `platforms/Hakefile`. To add a dependency, we can for example add our target to `modules_x86_64` by adding the line `"examples/xmpl-hello"` to the list of modules. Now our application is built automatically when invoking any X86 platform, but it is not yet bundled into the image and it wont be available at runtime. For the QEMU simulator target, the build system uses the `hake/menu.lst.x86_64` to describe which domains are included. Thus we add `module /x86_64/sbin/examples/xmpl-hello` to the menu.lst. Then we run `make qemu_x86_64` and once it has booted into our shell, we can use the command `./x86_64/sbin/examples/xmpl-hello` to run our application and hopefully be greeted with a ”hello world”.

2.2 A Simple Hello World Program Using IDC

In this section, we talk how we can write an application that uses Barrelfish IDC mechanism to communicate with other applications. For purpose of this section, we will develop simple **Hello World** client and server application. The client running on core-0 will send an IDC message to a server running on core-1. This message will contain a string with contents `"Hello World"`.

Adding IDC (inter-dispatcher communication) is a bit more complicated. It requires defining an interface in Flounder IDL, updating the interfaces Hakefile, updating the domain's (myapp) Hakefile, and then adding initialization code as well as appropriate callbacks to the source file.

2.2.1 Interface

In this subsection, we will create a new interface for our use. We do this by creating a new interface file called `hello.if` in the `if`
directory. Following is simple interface that will suffice for our requirements of sending single string as a message.

```
interface hello "Hello World interface" {
    message hello_msg(string s);
};
```

You can find out more about how to write such interfaces by referring IDC-Technote.

You also need to modify the `if/Hakefile` to add this newly created interface into the compilation process.

```
[ flounderGenDefs (options arch) f
      | f <- [ "ahci_mgmt",
```

```
                ...
                ...
         "hello"],
             arch <- allArchitectures
] ++
```

Adding the name of your interface here will make sure that the communication stubs will be automatically generated for by the compilation process. These can be found in following location in your build directory:

```
BUILD/ARCH/include/if/hello_defs.h
BUILD/ARCH/include/if/hello_lmp_defs.h
BUILD/ARCH/include/if/hello_ump_defs.h
BUILD/ARCH/include/if/hello_multihop_defs.h
```

There will be one for each type of communication mechanism supported by the Barrelfish. You don't need to worry about these files as build mechanism will take care of these files. Only things developers need to do is write the interface file and include appropriate headers in the application code (discussed in following section)

2.2.2 Application

In this section, we discuss how would we write the application code, and how to compile it. For sake of simplicity, we will write both server and client code in same application. Following is the code for the `main` function, which runs the client code or server code based on the command line argument.

```
int main(int argc, char *argv[]) {
    errval_t err;
    if ((argc >= 2) && (strcmp(argv[1], "client") == 0)) {
        start_client();
    } else if ((argc >= 2) && (strcmp(argv[1], "server") == 0)) {
        start_server();
    } else {
        return EXIT_FAILURE;
    }
    /* The dispatch loop */
    struct waitset *ws = get_default_waitset();
    while (1) {
        err = event_dispatch(ws); /* get and handle next event */
        if (err_is_fail(err)) {
            DEBUG_ERR(err, "in event_dispatch");
            break;
        }
    }
    return EXIT_FAILURE;
}
```

The important part of the above is in the dispatch loop where application will wait for events on the default wait-set and handle those events in infinite while loop.

2.2.3 Server side

In this section, we will develop the server code in steps of ”exporting service”, ”registering the service” and ”handling actual requests”.

Exporting service

As a first step, server needs to export the service it wants to provide by calling export function on the service interface. In this case, server will call `hello_export` as we are providing *Hello World* service.

```
static void start_server(void)
{
    errval_t err;
    err = hello_export(NULL /* state for callbacks */,
            export_cb, /* Callback for export */
            connect_cb,  /* Callback for client connects */
            get_default_waitset(), /* waitset where events will be sent */
            IDC_EXPORT_FLAGS_DEFAULT);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "export failed");
    }
}
```

This call also registers two callback handles. Once the service is successfully exported, then the export callback provided. When any client connects with the service then the connect callback will be called. We will see the use of these callbacks in next few steps.

Registering service

Server also need to register the service so that other applications can find it. Following code registers the `"hello_service"` on the callback from successful completion of export:

```
const static char *service_name = "hello_service";
static void export_cb(void *st, errval_t err, iref_t iref)
{
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "export failed");
    }
    err = nameservice_register(service_name, iref);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "nameservice_register failed");
    }
}
```

The `nameservice_register` is a blocking call which will connect to the global nameserver and publish the service name.

Connect and handling messages

This section describes how exactly the connection is established and requests are handled.

```
static void rx_hello_msg(struct hello_binding *b, char *str)
{
    printf("server: received hello_msg:\n\t%s\n", str);
    free(str);
}

static struct hello_rx_vtbl rx_vtbl = {
    .hello_msg = rx_hello_msg,
};
```

```
static errval_t connect_cb(void *st, struct hello_binding *b)
{
    b->rx_vtbl = rx_vtbl;
    return SYS_ERR_OK;
}
```

The above code defines a function `rx_hello_msg` which will be responsible to actually handle the requests. In our case, we are just printing out the string we received as part of the message.

We need to create a list of function pointers where one function pointer is assigned for every possible incoming message. So we are creating an instance `rx_vtbl` of type `hello_rx_vtbl`.

On arrival of new connection, we provide this list of function pointers to register which functions to call for handling particular type of message.

2.2.4 Client side

Client needs to find the service and connect to it. Once the connection is successfully established then it can send the actual requests. In this section, we show how it can be done.

Find and Bind

Following code finds the desired service with given name and binds with the service.

```
static void start_client(void)
{
    errval_t err;
    iref_t iref;
    err = nameservice_blocking_lookup(service_name,
            &iref);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err,
                "nameservice_blocking_lookup failed");
    }
    err = hello_bind(iref, bind_cb, /* Callback function */
            NULL /* State for the callback */,
            get_default_waitset(),
            IDC_BIND_FLAGS_DEFAULT);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "bind failed");
    }
}
```

As the name suggests `nameservice_blocking_lookup` is a blocking call which will connect with the nameserver and query for the given service name. The `iref` handle returned by this call can be used for binding with the service.

The `hello_bind` is part of the code generated from the interface file and will try and connect with the server on appropriate channel.

Following code is callback function which will be called when client successfully connects with the server. This code also creates a client state which stores pointer to the communication channel binding. This code then calls a function `run_client` with the client state.

```
struct client_state {
    struct hello_binding *binding;
    int count;
};
```

```
static void bind_cb(void *st, errval_t err, struct hello_binding *b)
{
    struct client_state *myst = malloc(sizeof(struct client_state));
    assert(myst != NULL);
    myst->binding = b;
    myst->count = 0;
    run_client(myst);   /* calling run_client for first time */
}
```

Sending messages

Most of the actual work of sending message is done from the `run_client` function which is described bellow:

```
static void run_client(void *arg)
{
    errval_t err;
    struct client_state *myst = arg;
    struct hello_binding *b = myst->binding;

    /* Creating a continuation which will call run_client */
    struct event_closure txcont = MKCONT(run_client, myst);

    err = b->tx_vtbl.hello_msg(b, txcont, "Hello World");
    if (err_is_fail(err)) {
        DEBUG_ERR(err, "error sending message %d\n", myst->count);
    }
}
```

This function first creates a continuation which calls itself and will be used as a callback function. The next step is to actually send the message by calling an appropriate function on the send side of interface binding (`tx_vtbl.hello_msg`) with actual message and above created continuation. This call will register a message to be sent and a callback that will be triggered when message is successfully.

As you can notice, callback function is calling `run_client` again, leading an infinite loop of sending messages.

Include files

Following is a typical set of include files that you will need to add in your code.

```
#include <stdio.h>
#include <barrelfish/barrelfish.h>
#include <barrelfish/nameservice_client.h>
#include <if/hello_defs.h>
```

The `barrelfish.h` file contains declaration of functionalities related to Barrelfish OS (eg: system calls, capability system related functionalities, etc.) `nameservice_client.h` file provides declarations for functions related to registering and looking up services. The `hello_defs.h` file includes the declarations for automatically generated code to support `if/hello.if` interface.

2.2.5 Building and running the application

In this section, we talk about how build your application. For this, you will need to create a `Hakefile` in the code directory which should look something like bellow:

```
[ build application { target = "hello-cs",
                      cFiles = [ "hello.c" ],
                      flounderBindings = [ "hello" ]
                    }
]
```

This `Hakefile` specifies that you want to build an application with name `hellocs` from the `hello.c` file, and also it marks the dependency on the communication interface `hello`.

The next step is to modify the `barrelfish/hake/symbolic_targets.mk` file to include your newly created application into the build process. Here you can choose an architecture for which your application should be compiled, or you can also add your application in `MODULES_COMMON` list so that your application will be compiled for all architectures.

```
MODULES_COMMON= \
sbin/init_null \
        ...
        ...
sbin/xcorecapbench \
sbin/hello-cs \
```

After this, rebuild the Barrelfish to include newly added application. On successful compilation the binary will be created in sbin directory of desired architecture.

Next step is to edit one of the `barrelfish/hake/menu.lst.*` file based on which architecture you are targeting. Bellow is the minimal `menu.lst.x86_64` file to be able to run this newly created application.

```
title Barrelfish
root (nd)
kernel /x86_64/sbin/elver loglevel=3
module /x86_64/sbin/cpu loglevel=3
module /x86_64/sbin/init

# Domains spawned by init
module /x86_64/sbin/mem_serv
module /x86_64/sbin/monitor

# Special boot time domains spawned by monitor
module  /x86_64/sbin/ramfsd boot
module  /x86_64/sbin/skb boot
modulenounzip /skb_ramfs.cpio.gz nospawn
module  /x86_64/sbin/kaluga boot
module  /x86_64/sbin/acpi boot
module  /x86_64/sbin/spawnd boot
#bootapic-x86_64=1-15
module  /x86_64/sbin/startd boot
module /x86_64/sbin/routing_setup boot

# Drivers
module /x86_64/sbin/pci auto
module /x86_64/sbin/ahcid auto

# General user domains
```

```
module /x86_64/sbin/serial

# Your hello world application (both server and client)
module /x86_64/sbin/hello-cs core=0 server
module /x86_64/sbin/hello-cs core=1 client
```

Now, you can build your application and verify that binary is created as follows:

```
$ make
$ ls x86_64/sbin/hello-cs
```

If everything goes fine, you can run the application in `qemu` simulator with following command:

```
$ make sim
```

At this moment, you should be able see Barrelfish booting and starting your applications which are then able to communicate by sending "Hello World" message in infinite loop. Following is a sample output that you may expect while running this setup.

```
...
...
...
startd.0: starting app /x86_64/sbin/hello-cs on core 0
spawnd.0: spawning /x86_64/sbin/hello-cs on core 0
startd.0: starting app /x86_64/sbin/hello-cs on core 1
skb.0: waiting for: spawn.1
skb.0: waiting for: pci
spawnd.0: spawning /x86_64/sbin/ahcid on core 0
kernel: 0: installing handler for IRQ 1
kernel: 0: installing handler for IRQ 2
pci_client.c: got vector 2
ahcid: registered device 8086:2922
spawnd.1: spawning /x86_64/sbin/hello-cs on core 1
No bootscript
server: received hello_msg:
        Hello World
server: received hello_msg:
        Hello World
server: received hello_msg:
        Hello World
server: received hello_msg:
        Hello World
server: received hello_msg:
        Hello World
server: received hello_msg:
        Hello World
...
...
...
```