Security Research

PASR

Preparatory Action on the enhancement of the European industrial potential in the field of Security research

Grant Agreement no. SEC5-PR-104600
ROBIN

Open Robust Infrastructures

Project

Deliverable D.10

Implementation of ROBIN TCB Kit

Due date of deliverable: 31/04/2008
Actual submission date: 15/05/2008

Start date of Activity: 31/01/2006 Duration: 31/04/2008

Organisation name of lead beneficiary for this deliverable: Technische Universität Dresden

Revision 1

CLASSIFICATION: Public

15/05/2008

A guided walk through Bastei

Norman Feske, Björn Döbel

April 11, 2008

Contents

This guide is meant to provide you a painless start with the Bastei source tree, with the default demonstration scenario, and with creating your first Bastei application.

1 Introduction to Bastei's development environment

1.1 Quick start to build Bastei for Linux

The best starting point for exploring Bastei is to run it on Linux. Make sure that your system satisfies the following requirements:

- GNU `Make` version 3.81 or newer installed
- GNU `gcc-4.*` installed
- `libSDL-dev` installed

In the following, we assume that you downloaded, extracted, and entered the root directory of the Bastei sources.

The Bastei build system never touches the source tree but generates object files, libraries, and targets inside a build directory. We do not have a build directory yet. So let us create one:

1. Create a directory at the same level as the `base/` directory:

   ```
   mkdir build
   cd build
   ```

2. Create a symbolic link for the Makefile that manages the build process:

   ```
   ln -s ../tool/builddir/build.mk Makefile
   ```

Now let us build the targets by simply issuing

```
make
```

After the build process is successfully finished, you find the newly created binaries in the `bin/` subdirectory. By default however, you won't find any exciting applications to play with. The Bastei base system comes with only one real program called *Core*. Anyway, to see a lifesign of Bastei, you may give Core a shot:

```
cd bin
./core
```

As indicated by the debug output, `Core` tries to spawn the `Init` process, which does not exist yet.

1.1.1 Adding additional source-code repositories

To keep the Bastei base system tidy and neat, the build system provides a way to implement non-basic Bastei components outside the Bastei source tree by incorporating additional source code repositories into the build process. Example of such additional source-code repository are the `os` and `demo` repositories. The `os` repository contains device driver, fundamental services, and the `Init` process. The `demo` repository supplements the `os` repository with some components to graphically demonstrate the Bastei concepts. You can tell the build system to incorporate these repositories into the build process by creating an `etc/` subdirectory in your build directory and adding a configuration file called `etc/build.conf` with the following entry:

```
REPOSITORIES = ../base ../os ../demo
```

If you now issue `make` again, you will see some additional targets such as `init`, `nitpicker`, `timer`, and `fb_sdl` to be created.

When starting Core now, the debug output indicates that Core successfully spawns `init`, which fails to access its configuration file. You can cancel the execution of `Core` by pressing `Control-C`. As a starting point, you can use the configuration-file template supplied with the `os` repository:

```
cp ../../os/config/linux_demo config
```

If you start Core again, you will get a more satisfying result. At least, you should see something colorful popping up.

1.1.2 Building during development

When using multiple source-code repositories, the overall build process may revisit a lot of targets and thus, give you some extra seconds to look bored on your screen. To improve the workflow during development, the Bastei build system supports building subtrees of the sources. For example, by issuing

```
make core server/nitpicker
```

The build system builds all targets found in the `core` and `nitpicker` source directories. As indicated by the build output, the build system revisits each library that is used by the target. This is very handy for developing libraries because instead of re-building your library and then your library-using program, you just build your program and that's it. This concept even works recursively, which means that libraries may depend on other libraries.

In practice, you won't ever need to build the *whole tree* but only the targets that you are interested in.

1.1.3 Controlling the verbosity of the build process

If you are venturesome enough to try understanding the inner workings of the build system in more detail, you can tell the build system to display each directory change by specifying

```
make VERBOSE_DIR=
```

If you are interested in the arguments that are passed to each invocation of `make`, you can make them visible via

```
make VERBOSE_MK=
```

Furthermore, you can observe each single shell-command invocation by specifying

```
make VERBOSE=
```

Of course, you can combine these verboseness toggles for maximizing the noise.

1.1.4 Customizing your work flow

The `etc/build.conf` file in your build directory is not only useful for declaring the repositories to be included into the build process but it also enables you to customize your work flow. For example, if you happen to specify `VERBOSE=` to the build process at a regular basis, you can create a syntactical shortcut for this argument by adding the following snippet into your `etc/build.conf` file:

```
ifeq ($(V),1)
VERBOSE =
endif
```

With this addition, you can use

```
make V=1
```

to achieve the same effect as by using

```
make VERBOSE=
```

During development, your may often restrict the build process to revisit only the targets that you are working on. For example, if only the targets within `core`, `init`, and `drivers/framebuffer` are of interest to your work:

```
make core init drivers/framebuffer
```

As this list of targets grows, the manual specification of the targets becomes sufficiently inconvenient. You can alternatively specify the these targets by overriding the `MAKECMDGOALS` make variable in your `etc/build.conf` file:

```
MAKECMDGOALS = init core drivers/framebuffer
```

1.2 Source tree directory layout

The Bastei source tree has the following layout:

Directory	Description
`doc/`	Documentation, specific for the repository
`etc/`	Default configuration of the build process
`mk/`	The build system
`include/`	Globally visible header files
`src`	Source codes and target build descriptions
`lib/mk/`	Library build descriptions

1.3 Creating targets and libraries

1.3.1 Target descriptions

A good starting point is to look at the Init target. The source code of Init is located at `os/src/init/`. In this directory, you will find a target description file named `target.mk`. This file contains the building instructions and it usually is very simple. The build process is controlled by defining the following variables. Additionally, a `target.mk` file may contain custom `make` rules. For an example, take a look at `demo/src/nitpicker/bastei/target.mk`.

Build variables to be defined by you

`TARGET` is the name of the binary to be created. This is the only **mandatory variable** to be defined in a `target.mk` file.

`REQUIRES` expresses the requirements that must be satisfied in order to build the target. You find more details about the underlying mechanism in Section 1.3.3.

`LIBS` is the list of libraries that are used by the target.

`SRC_CC` contains the list of `.cc` source files. The default search location for source codes is the directory, where the `target.mk` file resides.

`SRC_C` contains the list of `.c` source files.

`SRC_S` contains the list of assembly `.s` source files.

`SRC_BIN` contains binary data files to be linked to the target.

`INC_DIR` is the list of include search locations. Directories should always be appended by using +=. Never use an assignment!

`EXT_OBJECTS` is a list of Bastei-external objects or libraries. This variable is mostly used for interfacing Bastei with legacy software components.

Rarely used variables

`CC_OPT` contains additional compiler options to be used for `.c` as well as for `.cc` files.

`CXX_OPT` contains additional compiler options to be used for the C++ compiler only.

`LD_OPT` contains additional linker options.

Specifying search locations When specifying search locations for header files via the `INC_DIR` variable or for source file via `vpath`, relative pathnames are illegal to use. Instead, you can use the following variables to reference locations within the source-code repository, where your target lives:

`REP_DIR` is the base directory of the current source-code repository. Normally, specifying locations relative to the base of the repository is never used by `target.mk` files but needed by library descriptions.

`PRG_DIR` is the directory, where your `target.mk` file resides. This variable is always to be used when specifying a relative path.

1.3.2 Library descriptions

In contrast to target descriptions that are scattered across the whole source tree, library descriptions are located at the central place `lib/mk`. Each library corresponds to a <`libname`>`.mk` file. The base of the description file is the name of the library. Therefore, there is no `TARGET` variable to be set. The source code locations are expressed as `$(REP_DIR)`-relative `vpath` commands.

1.3.3 Specializations

Building components for different platforms likely implicates portions of code that are tied to certain aspects of the target platform. For example, a target platform may be characterized by

- A kernel API such as L4v2, Linux, L4.sec,
- A hardware architecture such as x86, ARM, Coldfire,
- A certain hardware facility such as a custom device, or even
- Other properties such as software license requirements.

All these attributes express a specialization of the build process. The Bastei build system provides a generic mechanism to handle such specializations.

The *programmer* of a software component knows the properties on which his software relies and thus, specifies these requirements in his build description file.

The *user/customer/builder* decides to build software for a specific platform and defines the platform specifics via the `SPECS` variable per build directory in `etc/specs.conf`. The default platform specification that you just used in Section 1.1 is located at `base/etc/specs.conf`.

Each <`specname`> in the `SPECS` variable instructs the build system to

- Include the `make`-rules of a corresponding `base/mk/spec-`<`specname`>`.mk` file. This enables us to customize the build process for each platform. For example, the `spec-linux.mk` file tells the build system to add Linux-specific include search directories and link the POSIX thread library to the target.
- Search for <`libname`>`.mk` files in the `lib/mk/`<`specname`>`/` subdirectory. This way, we can provide alternative implementations of one and the same library interface for different platforms.

Before a target or library gets built, the build system checks if the `REQUIRES` entries of the build description file are satisfied by entries of the `SPECS` variable. The compilation is executed only if each entry in the `REQUIRES` variable is present in the `SPECS` variable as supplied by the build directory configuration.

1.3.4 Using a secondary source-code repository for your developments

A source-code repository is a directory tree with a layout that corresponds to the Bastei source tree. The following directories are mandatory:

```
lib/mk/
src/
include/
```

2 Exploring the provided demonstration setup

This section provides a step-by-step guide through the demo scenario provided with Bastei. The scenario highlights the following features:

- Creation and destruction of single processes as well as arbitrarily complex sub systems
- Trusted-path facility of the Nitpicker secure GUI
- Assignment of resource quotas to sub systems
- Multiple instantiation of services
- Usage of run-time adaptable policy for routing client requests to different services

2.1 Bootstrapping the system

The Bastei system is structured as a tree of processes with the Core process as the root of the tree. Core provides the most fundamental services that are needed to start processes and manage physical resources. These services are the following:

RAM is the memory manager. From this service, processes can obtain memory.

ROM is the file provider for files that are present at boot time. Depending on the platform, these files are stored inside a ROM chip or loaded by a boot loader.

CPU is the thread manager, which enables processes to create, manage, and destroy threads.

TASK is a facility to create address spaces. Each Bastei process lives in a separate address space (called task) that is by default completely isolated from all other processes except from its parent process. You can think of a task as a virtual world in which the process exists. At creation time, a task is completely empty and it is the job of the task's parent to define the layout and the content of the address space before starting the task's first thread.

RM is the manager for address-space layouts (called region map). A region map is used by a parent to define the execution environment of a new child process at creation time. During its lifetime, a process may also use its own region map to manipulate its address-space layout directly.

CAP is a capability manager, which is used for establishing communication channels between processes.

In addition, Core provides the following services to user-level device drivers:

I/O is the input/output manager, which makes memory-mapped I/O registers and I/O ports accessible to user-level device drivers.

IRQ is the interrupt manager. This service is used by user-level device drivers to handle device interrupts.

The fundamental services described above are complemented by an additional service called LOG that enables processes to print debug output.

Core only provides these services as raw mechanisms and it is completely free from policy. That means there exist no means of boot-time configuration. You can think of Core as the user-level portion of the underlying kernel. The only built-in policy of Core is starting the Init process and transferring all physical resources such as all the available memory to Init. In contrast to Core, Init is driven by policy. This policy defines the static bootstrapping of further processes and it is expressed via XML syntax in Init's `config` file. Init obtains this file from Core's ROM service as a ROM module. For the demo scenario, the config file looks as follows:

```
<config>
  <start>
    <filename>fb_sdl</filename>
    <ram_quota>2M</ram_quota>
  </start>
  <start>
    <filename>timer</filename>
    <ram_quota>0x10000</ram_quota>
  </start>
  <start>
    <filename>nitpicker</filename>
    <ram_quota>1M</ram_quota>
  </start>
  <start>
    <filename>launchpad</filename>
    <ram_quota>2G</ram_quota>
  </start>
</config>
```

Init's configuration is a list of processes to be started as children of Init, whereas the order of the `start` entries determines the starting order of the processes. Each `start` entry can further describe policy to be applied to the process. For the demo scenario, we have assigned a different amount of memory via the `ram_quota` definition to each process. If the specified value exceeds the available memory (see the launchpad entry), Init assigns all of the remaining available memory to the process.

The processes started by this configuration are the following:

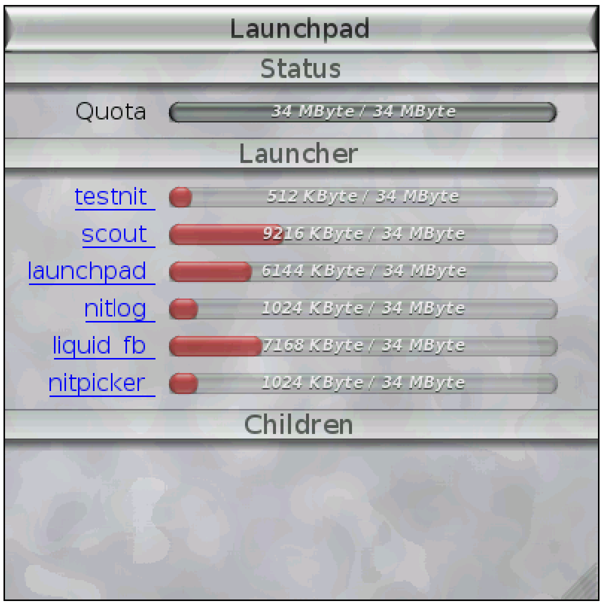

Figure 1: Main window of the launchpad application starter.

Fb_sdl is a process that provides services for accessing the frame buffer and for requesting user input. The interfaces for these services are platform independent but `fb_sdl` is a Linux-specific implementation of these services that relies on `libSDL`.

Timer is a process that enables other processes to wait for a specified amount of time.

Nitpicker is a low-complexity GUI server that allows multiple graphical applications to share the graphics device and user input in a secure fashion. More information about its concept are provided by the following paper:

Norman Feske and Christian Helmuth: "A Nitpicker's guide to a minimal-complexity secure GUI", *In proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005), Tucson, Arizona, USA, December 2005.*

`http://os.inf.tu-dresden.de/papers_ps/feske-nitpicker.pdf`

Launchpad is a graphical application for starting and killing further processes. This is the program that you see right after starting the demo scenario and which is assigned all remaining memory by Init.

2.2 The launchpad application starter

Figure 1 shows the main window of the launchpad application. It consists of three areas. The upper area contains status information about launchpad itself. The available memory quota is presented by a grey-colored bar. The middle area of the window contains the list of available applications that can be started by clicking on the application's name. Before starting an application, the user can define the amount of memory quota to donate to the new application by adjusting the red bar using the mouse. For a first test, you may set the memory quota of the program named scout to 10MB and then click its name. In turn, the

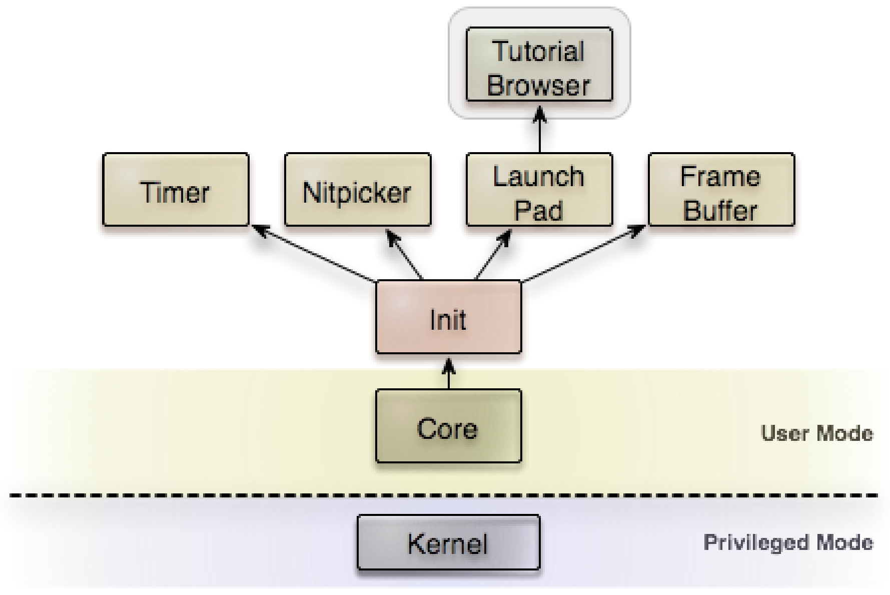

Figure 2: Illustration of the system setup after having started the scout tutorial browser.

scout tutorial browser will be started and the lower area of launchpad becomes populated the status information about launchpad's children. Currently, launchpad has scout as its only child. For each child, its name, its memory quota, and a kill button are presented. After having started scout, you will further notice a change of launchpad's own status information as the memory quota spent for scout is not directly available to launchpad anymore.

Inside the scout window, you see an illustration of the current setup (figure 2). At the very bottom, there are the kernel, Core, and Init. Init has started framebuffer, timer, input, nitpicker, and launchpad as it children (note: on Linux, input and framebuffer are both contained in the `fb_sdl` process). Launchpad has started scout as its only child. You can get a further idea about the relationship between the applications visible on screen, by pressing the `ScrLk` key, which gets especially handled by the nitpicker GUI server. We call this key the X-ray key because it makes the identity of each window on screen visible to the user. Each screen region gets labeled by its chain of parents and their grandparents respectively. For example, the scout window is labeled by "Launchpad → Scout", which tells us that this program was started by launchpad. During the walk through the demo scenario, you may press the X-ray key at any time to make the parent-child relationships visible on screen.

By pressing the kill button of the scout child in launchpad's window, scout will disappear and launchpad regains its original memory quota. Although killing a process may sound like a simple thing to do, it is worthwhile to mention that scout was using a number of services, for example Core's LOG service, the nitpicker GUI service, and the timer service. While using these services, scout made portions of its own memory quota available to them. When scout was killed by launchpad, all those relationships were gracefully reverted such that there is no resource leakage.

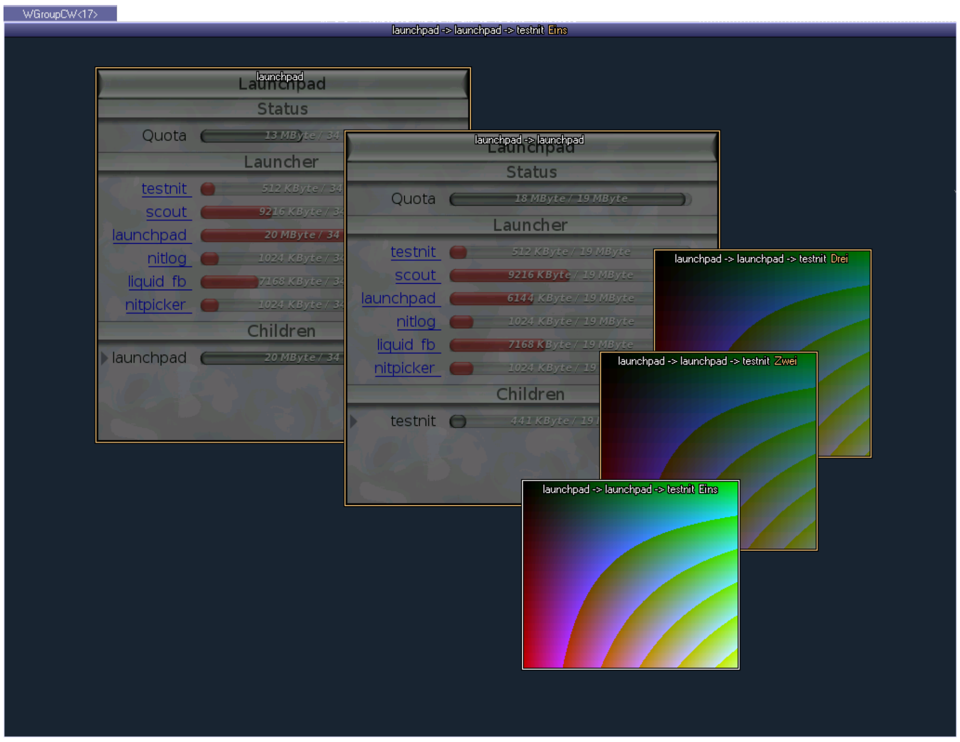

Figure 3: A second instance of launchad is used to start the `testnit` program, which manages three colored windows. The identity of each screen regions is unveiled by the X-ray mode of the nitpicker GUI server.

2.3 Recursive system structure

Thanks to the recursive structure of Bastei, the mechanisms that function for a single application are also applicable to whole sub systems. As a test, you may configure the launchpad application entry within the launchpad window to 20MB and start another instance of launchpad. A new launchpad window will appear. Apart from the status information at the upper part of its window, it looks completely identical to the first instance. With the new instance, you may start further applications, for example by clicking on `testnit`. To distinguish the different instances of the applications on screen, the X-ray key becomes handy again. Figure 3 shows a screenshot of the described setup in X-ray mode. Now, after creating a whole hierarchy of applications, you can try killing the whole tree at once by clicking the kill button of the launchpad entry in the original launchpad window. You will notice that whole sub system gets properly destructed and the original system state is regained.

2.4 The flexibility of nested policies

Beside providing the ability to construct and destruct hierarchically structured sub systems, the recursive system structure allows for an extremely flexible definition and management of system policies that can be implanted into each parent. As an example, launchpad has a simple built-in policy of how children are connected to services.

If a child requests a service, launchpad looks if such a service is provided by any of the other children and, if so, a connection gets established. If the service is not offered by any child, launchpad delegates the request to its parent. For example, a request for the `LOG` service will always end up at Core, which implements the service by the means of terminal (or kernel debug) output. By starting a child that offers the same service interface, however, we can shadow Core's `LOG` service by an alternative implementation. You can try this out by first starting `testnit` and observing its log output at the terminal window. When started, `testnit` tells us some status information. By further starting the program called `nitlog`, we create a new `LOG` service as child of launchpad. On screen, this application appears just as a black window that can be dragged to any screen position with the mouse. When now starting a new instance of `testnit`, launchpad will resolve the request for the `LOG` service by establishing a connection to `nitlog` instead of propagating the request to its parent. Consequently, we can now observe the status output of the second `testnit` instance inside the `nitlog` window.

The same methodology can be applied to arbitrarily complex services. For example, you can create a new instance of the framebuffer service by starting the `liquid_fb` application. This application provides the framebuffer service and, in turn, it uses nitpicker to get displayed on screen. Because any new requests for a framebuffer will now be served by the `liquid_fb` application, we can start another instance of nitpicker. This instance uses `liquid_fb` as its graphics back end and, in turn, provides the GUI service. Now, when starting another instance of scout, the new scout window will appear within `liquid_fb` too (Figure 4).

The extremely simple example policy implemented in launchpad in combination with the recursive system structure of Bastei already provide a wealth of flexibility without the need to recompile or reconfigure any application. The policy implemented and enforced by a parent may also deny services for its children or impose other restrictions. For example, the window labels presented in X-ray mode are successively defined by all parents and grandparents that mediate the request of an application to the GUI service. For example, the launchpad as the

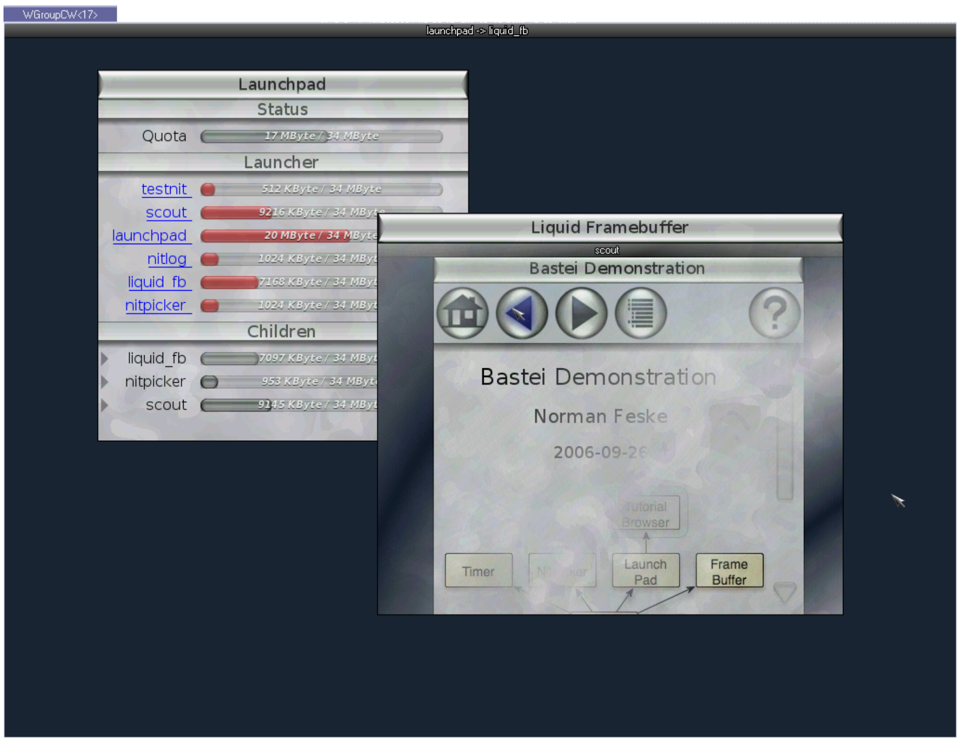

Figure 4: Executing multiple instances of the nitpicker GUI server in a nested way.

parent of scout imposes its policy of labeling the GUI session with the label “Scout”. Init as the parent of launchpad again overrides this label by the name of its immediate child from which the GUI request comes from. Hence the label becomes “Launchpad → Scout”.

3 Creating your first Bastei application

This section will give you a step-by-step introduction for writing your first little client-server application using the Bastei OS Framework. We will create a server that provides two functions to its clients and a client that uses these functions. The code samples in this section are not necessarily complete. You can download the complete tutorial source code from the link at the bottom of this page.

3.1 Prerequisites

We assume that you know how to write code and have read:

> Norman Feske and Christian Helmuth: “Design of the Bastei OS Architecture”, *TU Dresden technical report TUD-FI06-07, Dresden, Germany, December 2006.*
>
> `http://os.inf.tu-dresden.de/papers_ps/bastei_design.pdf`

so that you have a basic understanding of what Bastei is and how things work. Of course, you will also need to check out Bastei before going any further.

3.2 Setting up the build environment

The Bastei build system enables developers to create software in different repositories that don't need to interfer with the rest of the Bastei tree. We will do this for our example now. In the Bastei root directory, we create the following subdirectory structure:

```
hello_tutorial
hello_tutorial/include
hello_tutorial/include/hello_session
hello_tutorial/src
hello_tutorial/src/hello
hello_tutorial/src/hello/server
hello_tutorial/src/hello/client
```

In the remaining document when referring to non-absolute directories, these are local to `hello_tutorial`. Now we tell the Bastei build system, that there is a new repository. Therefore we add the path to our new repository to `build/etc/build.conf`:

```
REPOSITORIES = ../base ../os ../hello_tutorial
```

Later we will place build description files into the tutorial subdirectories so that the build system can figure out what is needed to build your applications. You can then build these apps from the `build` directory using one of the following commands:

```
make hello
make hello/server
make hello/client
```

The first command builds both the client and the server whereas the latter two commands build only the specific target respectively.

3.3 Defining an interface

In our example we are going to implement a server providing two functions:

`void say_hello()` makes the server print "Hello world."

`int add(int a, int b)` adds two integers and returns the result.

The interface of a Bastei service is called a *session*. We will define it as a C++ class in `include/hello_session/hello_session.h`

```
namespace Hello {

  class Session
  {
    protected:

      enum Opcode { SAY_HELLO = 23, ADD = 42 };

    public:

      virtual ~Session() { }
      virtual void say_hello() = 0;
      virtual int add(int a, int b) = 0;
  };
}
```

As a good practice, we place the Hello Service into a dedicated namespace. The *Hello::Session* class defines the public interface for our service as well as a protected enumeration defining the opcodes for the RPCs we are going to implement.

3.4 Writing server code

Now let's write a server providing the interface defined by *Hello::Session*.

3.4.1 Implementing the server interface

First of all, we are going to implement a server-side communication stub that dispatches the `Hello::Session` interface and is derived from this abstract class as well as from the `Server_object` class defined by the Bastei framework. Therefore, we need to include `hello_session.h` and the `base/server.h` that includes the definition of `Server_object`. A server object contains one important method:

`dispatch(int opcode, Ipc_istream is, Ipc_ostream os)`

This method is called whenever a client calls our server. From the `opcode` parameter it can decide, which actions are to be performed. The second parameter, `Ipc_istream is`, is a stream of input arguments, the third one is the stream to which output parameters are written.

With this knowledge, we can implement our server-side `Hello::Session_server` class in `include/hello_session/server.h`:

```
#include <hello_session/hello_session.h>
#include <base/server.h>

namespace Hello {

  class Session_server : public Bastei::Server_object,
                         public Hello::Session
  {
    public:

      Session_server() {
        PDBG("Creating session component."); }

      int dispatch(int op, Bastei::Ipc_istream is,
                   Bastei::Ipc_ostream os)
      {
        int a = 0;
        int b = 0;

        PDBG("dispatch op %d", op);

        switch(op) {
```

```
        case SAY_HELLO:
          say_hello();
          break;

        case ADD:
          is >> a;
          is >> b;
          PDBG("%d + %d ?", a, b);
          os << add(a,b);
          break;

        default:
          PWRN("Invalid opcode.");
        }

        return 0;
      }
  };
}
```

We learn some new things here:

- The dispatch method is basically a switch statement concerning the Opcodes we defined in `Hello::Session`.
- For the `ADD` opcode we see how the server makes use of the `Ipc_istream` and `Ipc_ostream` parameters. Before calling the `add` method, we read our parameters from the `Ipc_istream`. Afterwards we write the result to the `Ipc_ostream`. Bastei's IPC framework takes care of marshalling and unmarshalling the messages passed between client and server. **Note, it is important that clients write their arguments to the IPC streams in the order the server expects them to do so.**

For a detailed discussion of the RPC framework and comparison to traditional approaches refer to:

> Norman Feske: "A Case Study on the Cost and Benefit of Dynamic RPC Marshalling for Low-Level System Components", *SIGOPS OSR Special Issue on Secure Small-Kernel Systems, 2007.*

Now that we have created the server-side dispatch code, the only thing that is missing is the implementation of the actual functionality provided via the RPC interface. We place the implementation of the virtual functions as declared in session interface into the `Session_component` class derived from `Session_server`. By placing the implementation of these functions into a separate *component* class, the implementation of the RPC interface remains independent from the communication stub code such that one and the same RPC interface (`Session`, `Session_client`, `Session_server`) could have multiple implementations. Because the `Session_component` class is not part of the RPC interface but part of the server implementation, we place this class into the file `src/hello/server/component.h` local to the source code of the server implementation:

```
#include <hello_session/server.h>
```

```
namespace Hello {

  class Session_component : public Hello::Session_server
  {
    void say_hello() {
      PDBG("I am here... Hello."); }

    int add(int a, int b) {
      return a + b; }
  };
}
```

3.4.2 Getting ready to start

The server object won't help us much as long as we don't use it in a server application. Starting a service with Bastei works as follows:

- Open a CAP session to our parent, so that we are able to create capabilities.
- Create and announce a root capability to our parent.
- When a client requests our service, the parent invokes the root capability to create session objects and session capabilities. These are then used by the client to communicate with the server.

We did not yet define a root object, so we do it now in `src/hello/server/main.cc`. The class `Hello::Root_component` derives from Bastei's `Root_component` class. This class defines a `_create_session` method which is called when a client wants to establish a connection to the server. This function is responsible for parsing the parameter string the client hands over to the server and create a `Hello::Session_component` object from these parameters.

```
#include <base/printf.h>
#include <root/component.h>
#include "component.h"

namespace Hello {

  class Root_component : public Bastei::Root_component<Hello::Session_component>
  {
    protected:

      Hello::Session_component *_create_session(const char *args)
      {
        PDBG("creating hello session.");
        return new Hello::Session_component();
      }

    public:

      Root_component(Bastei::Server_entrypoint *ep,
                     Bastei::Allocator         *allocator)
      : Bastei::Root_component<Hello::Session_component>(ep, allocator)
      {
        PDBG("Creating root component.");
      }
  };
}
```

Now we only need a main method that announces the service to our parent:

```
#include <base/env.h>
#include <base/sleep.h>
#include <cap_session/client.h>

using namespace Bastei;

int main(void)
{
  /*
   * Get a session for the parent's capability service, so that we are able
   * to create capabilities.
   */
  Capability cap_session_cap = env()->parent()->session("CAP", "ram_quota=4K");
  Cap_session_client csc(cap_session_cap);

  /*
   * A sliced heap is used for allocating session objects - thereby we can
   * release objects separately.
   */
  static Sliced_heap sliced_heap(env()->ram_session(), env()->rm_session());

  /*
   * Create objects for use by the framework.
   *
   * Msgbufs are used for sending and receiving messages using a
   * Server_activation. A Server_entrypoint is created to announce
   * our service's root capability to our parent and manage incoming
   * session creation requests.
   */
  static Msgbuf<256>              snd_msg, rcv_msg;
  static Server_activation<4096> act(snd_msg, rcv_msg);
  static Server_entrypoint       entry(csc, act);

  static Hello::Root_component hello_root(entry, sliced_heap);
  env()->parent()->announce("Hello", entry.manage(hello_root));

  /* We are done with this and only act upon client requests now. */
  sleep_forever();

  return 0;
}
```

3.4.3 Making it fly

In order to run our application, we need to perform two more steps:

Tell the Bastei build system that we want to build `hello_server`. Therefore we create a `target.mk` file in `src/hello/server`:

```
 TARGET   = hello_server
 REQUIRES = bastei
 SRC_CC   = main.cc
 LIBS     = cxx env server
```

To tell Init to start the new program, we have to add the following entry to Init's `config` file, which is located at `build/bin/config`.

```
<config>
  <start>
    <filename>hello_server</filename>
    <ram_quota>256K</ram_quota>
  </start>
</config>
```

Now rebuild `hello/server`, go to `build/bin`, run `./core`, and get excited.

3.5 Writing client code

In the next part we are going to have a look at the client-side implementation. The most basic steps here are:

- Get a capability for the Hello service from our parent.
- Create a `Bastei::Ipc_client` using this capability.
- Invoke RPCs through the `Ipc_client`.

3.5.1 A client object

We will encapsulate the Bastei IPC interface in a `Hello::Client` class for ease-of-use purposes. This class derives from `Hello:Session` and implements a client-side object. Therefore edit `include/hello_session/client.h`:

```
#include <hello_session/hello_session.h>
#include <base/ipc.h>
#include <base/printf.h>

namespace Hello {

  class Client : public Hello::Session
  {
    protected:

      Bastei::Capability  _session_cap;
      Bastei::Msgbuf<256> _sndbuf, _rcvbuf;
      Bastei::Ipc_client  _ipc_client;

    public:

      Client(Bastei::Capability cap)
      : _session_cap(cap), _ipc_client(cap, _sndbuf, _rcvbuf) { }


      ~Client() { }

      void say_hello()
      {
        PDBG("Saying Hello.");
        _ipc_client << SAY_HELLO << Bastei::IPC_CALL;
      }

      int add(int a, int b)
      {
        int ret = 0;
        _ipc_client << ADD << a << b << Bastei::IPC_CALL >> ret;
        return ret;
      }

      void invalid_op()
      {
        PDBG("Calling server with invalid opcode.");
        _ipc_client << 1234 << Bastei::IPC_CALL;
      }
  };
}
```

Things to note:

- A `Hello::Client` object is created using a `Capability`. From the given capability and two static message buffers we create an `Ipc_client` object, that is used to perform our IPC calls.
- An IPC client is a C++ stream to which we write our RPC. The first argument for an RPC is always the opcode. It is followed by the parameters and finally by `Bastei::IPC_CALL` if you want RPC semantics (which means to sleep until the RPC's result is available).
- For demonstration purposes, there is another function in the `Hello::Client` class called `invalid_op`. This function is used to show what happens if an invalid opcode is delivered to the server.

3.5.2 Client implementation

The client-side implementation using the `Hello::Client` object is pretty straightforward. We request a capability for the Hello service from our parent. This call blocks as long as the service has not been registered at the parent. Afterwards, we create a `Hello::Client` object with it and invoke calls. In addition, we use the timer service that comes with Bastei. This server enables us to sleep for a certain amount of milliseconds.

```
#include <base/env.h>
#include <base/printf.h>
#include <hello_session/client.h>
#include <timer_session/client.h>

using namespace Bastei;

int main(void)
{
  Capability hello_session_cap =
             env()->parent()->session("Hello", "foo, ram_quota=4K");
  Capability timer_session_cap =
             env()->parent()->session("Timer", "ram_quota=4K");

  Timer::Session_client timer_client(timer_session_cap);
  Hello::Client h(hello_session_cap);

  while (1) {
    h.say_hello();
    timer_client.msleep(1000);

    int foo = h.add(2,5);
    PDBG("Added 2 + 5 = %d", foo);
    timer_client.msleep(1000);

    h.invalid_op();
    timer_client.msleep(1000);
  }

  return 0;
}
```

3.5.3 Ready, set, go...

Add a `target.mk` file with the following content to `src/hello/client/`:

```
TARGET    = hello_client
REQUIRES  = bastei
SRC_CC    = main.cc
LIBS      = cxx env
```

Add the following entries to your `config` file:

```
<start>
  <filename>timer</filename>
  <ram_quota>256K</ram_quota>
</start>
<start>
  <filename>hello_client</filename>
  <ram_quota>256K</ram_quota>
</start>
```

Build `drivers/timer`, and `hello/client`, go to `build/bin`, and run `./core` again. You have now successfully implemented your first Bastei application.