Protobufs for kernel/user interface

Taylor ‘Riastradh’ Campbell
campbell@mumble.net
riastradh@NetBSD.org

EuroBSDcon 2015
Stockholm, Sweden
October 3, 2015

Kernel/user interface

- Userland does computation, talks to kernel for I/O
- Kernel serves as kind of RPC server for remote procedure calls
- System calls, ioctl commands for devices
- (For elaboration on kernel as RPC server, see rump_server.)

Kernel/user interface: syscalls

- Main RPC entry points: syscalls
- $\sim$ 450 syscalls in NetBSD
- Lots of attention
- Seldom changes

Kernel/user interface: ioctl

- Secondary entry point: `ioctl` syscall
- Hundreds or thousands in-tree
- Added/changed without much scrutiny
- Any driver module can add more
- Traditionally inputs and outputs are stored in simple C structs:
- `#define VNDIOCGET _IOWR('F', 3, struct vnd_user)`

```
struct vnd_user {
        int             vnu_unit;
        dev_t           vnu_dev;
        ino_t           vnu_ino;
};
```

Compatibility

- NetBSD kernel always supports previous version's userland
- (also older userlands, with extra libraries)
- Changes to syscalls, ioctls require compatibility code

Compatibility example

```
/* time_t is now int64_t */
struct timeval {
        time_t          tv_sec;
        suseconds_t     tv_usec;
};

/* time_t used to be int32_t */
struct timespec50 {
        int32_t tv_sec;
        long    tv_nsec;
};
```

Compatibility example

```
struct clockctl_clock_settime {
        clockid_t clock_id;
        const struct timespec *tp;
};

#define CLOCKCTL_CLOCK_SETTIME \
    _IOW('C', 0x7, struct clockctl_clock_settime)

struct clockctl50_clock_settime {
        clockid_t clock_id;
        const struct timespec50 *tp;
};

#define CLOCKCTL_OCLOCK_SETTIME \
    _IOW('C', 0x3, struct clockctl50_clock_settime)
```

Compatibility example

```
int
compat50_clockctlioctl(dev_t dev, u_long cmd, void *data, int flags,
    struct lwp *l)
{
        ...
        case CLOCKCTL_OCLOCK_SETTIME: {
                struct timespec50 tp50;
                struct timespec tp;
                struct clockctl50_clock_settime *args = data;

                error = copyin(args->tp, &tp50, sizeof(tp50));
                if (error)
                        return (error);
                timespec50_to_timespec(&tp50, &tp);
                error = clock_settime1(l->l_proc, args->clock_id,
                    &tp, true);
                break;
        }
        ...
}
```

Problems

- Need copy pasta for integer size change
- Need extra code for 32-bit userlands, 64-bit kernels
- Need extra code for new arguments
- Need extra code for new extra answers
- Extra code: seldom exercised, often buggy

Proplib

- Apple-style XML property lists
  - (Not exactly compatible with Apple proplists.)
- Added to NetBSD 4
- Used for some newer ioctls
- `prop_dictionary_sendrecv_ioctl`
- Yes: parsing (almost) XML in kernel

Proplib: advantages

- Can add fields, remove fields, without extra compat code
- Can easily handle nested structures, lists, etc.
- . . . and that's about it.

Proplib: disadvantages

- No type-checking: `void *` everywhere
- No *typo*-checking:

```
if (!prop_dictionary_get_uint32(dict, "wieght",
        &weight))
        weight = 0;
```

- XML parser in kernel
- No schema: compiler does not detect using field in wrong place
- Schemas seldom even informally documented beyond exact use in source code

Protobufs

- Binary RPC message format
- Used by Google internally, released in 2008
- Simple, compact wire format
- Simple message schema
- Designed to make compatibility easy
- Google `protoc` compiles `.proto` message schema into C++ library

Protobuf schema example from Google[1]

```
// person.proto
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

- Message fields are identified with numbers for wire format
- Names do not appear on wire — only in API
- Integers compactly encoded with as few bytes as necessary
- Fields may be required, optional, or repeated

[1] https://developers.google.com/protocol-buffers/, retrieved 2015-10-03

Protobuf sender example from Google[2]

```
// sender.cc
#include "person.pb.h"

Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```

[2] https://developers.google.com/protocol-buffers/, retrieved 2015-10-03

Protobuf receiver example from Google[3]

```
// receiver.cc
#include "person.pb.h"

Person john;
fstream input(argv[1], ios::in | ios::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();
```

[3] `https://developers.google.com/protocol-buffers/`, retrieved 2015-10-03

Protobuf compatibility

- Never change message tag numbers
- Add only *optional* or *repeated* fields
- Never add *required* fields
- Standard integers, e.g. `int32`, have same wire format for every size so if you change `int32` to `int64` then new readers can still handle old messages

Protobuf RPC

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}

message SearchResponse {
  repeated Result result = 1;
}

message Result {
  required string url = 1;
  optional string title = 2;
  repeated string snippets = 3;
}

service SearchService {
  rpc Search (SearchRequest)
    returns (SearchResponse);
}
```

Protobuf RPC

- Used internally by Google for a long time
- Released this year as gRPC[4]: `http://www.grpc.io/`

[4] `http://googledevelopers.blogspot.com/2015/02/introducing-grpc-new-open-source-http2.html`

Protobufs for ioctl

- ioctl is basically a channel for RPC from userland to kernel
- So why not use protobuf for the ioctl RPC?

```
message timespec {
        required int64_t sec;
        required uint32_t nsec;
}

message clockctl_settime_request {
        required int32 clock_id;
        required timespec tp;
}

message clockctl_settime_response {
}

service clockctl {
        rpc SETTIME(clockctl_settime_request)
            returns (clockctl_settime_response);
}
```

Protobuf implementations: Google `protoc`

- Google's original protobuf implementation
- Generates C++ code
- Resulting API very complex
- Resulting `.pb.cc`, `.pb.h` files very large
- No good for NetBSD kernel (C only)
- $\sim$ 100k lines of C++

Protobuf implementations: picopb

- New protobuf implementation
- Generates C code
- Resulting API very simple
  - . . . but still type-safe: no `void *`
- Resulting `.pb.c`, `.pb.h` files very compact
- Same schema format (not all features supported)
- Same wire format
- $\sim$ 10k lines of C
- `picopbc` detects and warns about cycles in messages
- `picopbc` computes maximum parser stack depth for non-cyclic messages
- (Name is a riff on nanopb, which wasn't small enough for me!)

picopb example: trivial case

```
#include "clockctl.pb.h"

int
clockctl_settime(int fd, int clock_id, const struct timespec *ts)
{
        struct clockctl_settime_request req;
        struct clockctl_settime_response resp;
        int ret;

        pb_init(clockctl_settime_request(&req));
        pb_init(clockctl_settime_response(&resp));
        req.clock_id = clock_id;
        req.tp.sec = ts->tv_sec;
        req.tp.nsec = ts->tv_nsec;
        ret = ioctl_pb(fd, CLOCKCTL_CLOCK_SETTIME,
            clockctl_settime_request(&req),
            clockctl_settime_response(&resp));
        pb_destroy(clockctl_settime_request(&req));
        pb_destroy(clockctl_settime_response(&resp));
        return ret;
}
```

picopb example: nested structures, lists

```
message hdaudio_fgrp_pin_config {
        repeated pin pins = 1 [(picopb).max = 128];

        message pin {
                required int32 nid = 1;
                required uint32 config = 2;
        }
}

message hdaudio_fgrp_info_request {
}

message hdaudio_fgrp_info_response {
        required hdaudio_fgrp_info fgrp_info = 1
            [(picopb).proplib.name = "function-group-info"];
}
```

picopb example: nested structures, lists

```
static int
hdaudioctl_list(int fd)
{
        struct hdaudio_fgrp_info_request request;
        struct hdaudio_fgrp_info_response response;
        const struct hdaudio_fgrp_info *info;
        const struct hdaudio_fgrp_info__fgrp *fgrp;

        pb_init(hdaudio_fgrp_info_request(&request));
        pb_init(hdaudio_fgrp_info_response(&response));
        if (ioctl_pb(fd, HDAUDIO_FGRP_INFO,
                hdaudio_fgrp_info_request(&request),
                hdaudio_fgrp_info_response(&response)) == -1)
                err(1, "ioctl(HDAUDIO_FGRP_INFO)");
```

picopb example: nested structures, lists

```
        info = &response.fgrp_info;
        for (i = 0;
             i < pb_repeated_count(&info->fgrps.repeated);
             i++) {
                fgrp = &info->fgrps.item[i];
                printf("codecid 0x%02"PRIX16" nid 0x%02"PRIX16
                    " vendor 0x%04"PRIX16" product 0x%04"PRIX16
                    " subsystem 0x%08"PRIX16" device %s\n",
                    fgrp->codecid, fgrp->nid,
                    fgrp->vendor, fgrp->product, fgrp->subsystem,
                    (fgrp->device.present
                        ? pb_string_ptr(fgrp->device.value)
                        : "<default>"));
        }

        pb_destroy(hdaudio_fgrp_info_response(&response));
        pb_destroy(hdaudio_fgrp_info_request(&request));
}
```

picopb proplib compatibility

- Normally, `libpicopb` encodes/decodes protobuf in standard wire format
- `libpicopbprop` encodes/decodes protobuf in XML property list wire format

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC
    "-//Apple Computer//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
        <dict>
                <key>config</key>
                <integer>0x90a60140</integer>
                <key>nid</key>
                <integer>18</integer>
        </dict>
        ...
```

picopb proplib compatibility

- Sometimes mapping between protobufs and XML property lists is not straightforward
- No support yet in `picopbc`, but. . .
- Annotate protobuf schema:

```
enum hdaudio_fgrp_type {
        HDAUDIO_FGRP_TYPE_UNKNOWN = 0
            [(picopb).proplib.value = "unknown"];
        HDAUDIO_FGRP_TYPE_AFG = 1
            [(picopb).proplib.value = "afg"];
        HDAUDIO_FGRP_TYPE_VSM_FG = 2
            [(picopb).proplib.value = "vsmfg"];
}

message hdaudio_fgrp_info_response {
        required hdaudio_fgrp_info fgrp_info = 1
            [(picopb).proplib.name = "function-group-info"];
}
```

Future

- Auto-generate ioctl stub code from

  `service DEVICE { rpc IOCCMD(...)  ...  }`
- Finish proplib compatibility support in `picopbc`
- Integrate into NetBSD system
- Convert existing proplib uses to picopb
  - ...grovel through code to discover schemas

Questions?

Questions?

Questions?[5]

Source code: `http://mumble.net/~campbell/hg/picopb/`

[5] Questions?