Ubus Brief

The ubus is designed for providing communication between various daemons and applications.
The architecture as below

Objects and Object paths
The Object paths are bindings can name object instances, and allow applications to refer to them.
In OpenWRT, the object path is namespace like network.interface.lan

Methods and Notifications
Methods are operations that can be invoked on an object, with optional input parameters and output.
Notifications are broadcasts from the object to any interested observers of the object.
The notifications may contain a data payload

Calling a method
A method call in ubus consists of two messages; A call messages from process A to process B and the reply messages from process B to process A.
The send message and reply messages are both routed through the ubus daemon.

The call message contains the method arguments. The reply messages may be error messages, or may contain method returned data.

Call Process

1.The call method messages contains the ubus connection context, the destination object id, the method name, the method arguments.
2. The method call message is send to the ubus daemon
3. The ubus daemon lookup the destination object id, if a process owns the object instance, then the daemon will forward the method call to the find process. Otherwise the ubus daemon creates an error messages and sends the error message back to the message call as reply.
4. The receiving process will parse the ubus object messages, and find the call method and arguments belong to the method. Then match the object methods in object instance, if find matched method, will invoke the method and then send the reply messages.
5. Ubus daemon receive the reply message and forward the reply message to the process that made the method call.
6. The reply messages is transferred as ubus blob messages structure which is TLV (Type-Length-Value) based binary messages type.
7. The process received the reply message should parse the message and format to human-nice message type as JSON or XML.

Notify Notifications

A notification in ubus consists of a single messages, send by one process to any number of other processes, which means the notification is a unidirectional broadcast, no need expected reply message.
The notification sender do not know the notifications recipients, it just send the notification onto bus The interest recipients should subscribe the sender object with the bus daemon.

Notification Process

1. Add notification object onto ubus daemon
2. The notification message contains ubus connection context, the notification sender object ID, the notification type and optional arguments with the type.
3. Any process on the ubus can subscribe the notification object. The bus may has a list of subscribers, which will match the observers when daemon handle the notification message.
4. The ubus daemon check the notification and determines which processes are interested in it. Then send the notification to all of the interested processes.
5. Each subscriber process receiving the notification decides what to do with the notification message.

Blob_buf structure on ubus

Blob_attr

Blob_msg

Blob_buf

How to use ubus

Server Main process

M1. Define a object with some abstract methods

M2. Connect the server process to ubus daemon and get a ubus_context, the context will contained the connected fd, registered fd callback and an AVL tree to manage all objects information with this connection

M3. Using uloop utilities to add the ubus_context, which is to register the connected fd into epoll set

M4. Add the defined object into ubusd

M5. Forever loop to epoll the fd set

What to do in method handler

H1. Parse the blob_attr msg into a blob_attr table, which can easy using by index the table by msg ID

H2. Get the method arguments according to msg id, the handler maybe call method in another objects or invoke a shell script to do some service, etc

H3. Prepare the response msg into blob_buff and send the response to ubus daemon, which will forward the response to request client if not specify “no_reply” or ”deferred” flag

H4. If specify “deferred” flag in req context in the method handler, which means the server process will not expect the response in this request handler and just complete this request.

```
#include <libubox/blobmsg_json.h>

#include "libubus.h"

static struct ubus_context *ctx;

static int test_hello(struct ubus_context *ctx, struct ubus_object *obj,

                                  struct ubus_request_data *req, const char *method,

                                  struct blob_attr *msg)

{

          struct hello_request *hreq;

          struct blob_attr *tb[__HELLO_MAX];

          const char *format = "%s received a message: %s";

          const char *msgstr = "(unknown)";



    // H1. Parse the blob_attr msg(blob_data(msg)) into a blob_attr

    //table (tb), which can easily use by msg ID to index the table

          blobmsg_parse(hello_policy, ARRAY_SIZE(hello_policy), tb, blob_data(msg),
blob_len(msg));


     // H2.  Get method arguments by msg ID

          if (tb[HELLO_MSG])

                    msgstr = blobmsg_data(tb[HELLO_MSG]);
```

```
            hreq = calloc(1, sizeof(*hreq) + strlen(format) + strlen(obj->name) + strlen(msgstr) + 1);
            sprintf(hreq->data, format, obj->name, msgstr);
```
// H4. Defer the reply for the request

// The reply will be making in timer callback
```
            ubus_defer_request(ctx, req, &hreq->req);
            hreq->timeout.cb = test_hello_reply;
            uloop_timeout_set(&hreq->timeout, 1000);
            return 0;
}

            // Define hello method with test_hello handle
            //hello policy tell ubusd  the object method parameters type
            static const struct ubus_method test_methods[] = {
                        UBUS_METHOD("hello", test_hello, hello_policy),
};
```
// M1. Define test_object
```
static struct ubus_object test_object = {
            .name = "test",
            .type = &test_object_type,
            .methods = test_methods,
            .n_methods = ARRAY_SIZE(test_methods),
};
static void server_main(void)
{
            int ret;
           // M4.  Add the defined object into ubusd
            ret = ubus_add_object(ctx, &test_object);
            if (ret)
                fprintf(stderr, "Failed to add object: %s\n", ubus_strerror(ret));
            // M5.  Forever loop to epoll the fd set and handle the available fd
            uloop_run();
}
int main(int argc, char **argv)
{
            const char *ubus_socket = NULL;
            int ch;
            uloop_init();
            signal(SIGPIPE, SIG_IGN);
             // M2.  Connect to ubusd, will get the ubus_context
            ctx = ubus_connect(ubus_socket);
            if (!ctx) {
                    fprintf(stderr, "Failed to connect to ubus\n");
                    return -1;
            }
              // M3.  Add the ubus connection into epoll set
            ubus_add_uloop(ctx);
            server_main();
            ubus_free(ctx);
            uloop_done();
```

```
        return 0;
}
```

Client Main Process

M1. Connect the client process to ubus daemon, will get the ubus context, the context will contained the connected fd, registered fd callback and an AVL tree to manage all objects information with this connection

M2.Using uloop utilities to add the ubus_context, which is to register the connected fd into epoll set

M3. Look up the target object id by the object path in ubus context

M4.   Arrange the ubus call method and method arguments into blob_buff.

M5. Invoke ubus high level API to invoke a method on a specific object,   and wait for the reply .

```
        /* invoke a method on a specific object */

        int ubus_invoke(struct ubus_context *ctx, uint32_t obj, const char *method, struct blob_attr *msg, ubus_data_handler_t cb, void *priv,

int timeout);
```

Specify a callback to handle the response blob_msg to human-nice message format like JSON or XML

Or

M4.   For some case, we may not need to wait for the response, should call asynchronous version invoke

```
/* asynchronous version of ubus_invoke() */

int ubus_invoke_async(struct ubus_context *ctx, uint32_t obj, const char *method, struct blob_attr *msg, struct ubus_request *req);
```

```
static int ubus_cli_call(struct ubus_context *ctx, int argc, char **argv)

{

        uint32_t id;
        int ret;

        if (argc < 2 || argc > 3)

                return -2;

        //M4. Arrange the ubus call method and method arguments into blob_buff

        blob_buf_init(&b, 0);

        if (argc == 3 && !blobmsg_add_json_from_string(&b, argv[2])) {

                if (!simple_output)

                        fprintf(stderr, "Failed to parse message data\n");

                return -1;

        }

        //M3. Look up the target object id by the object path

        ret = ubus_lookup_id(ctx, argv[0], &id);

        if (ret)

                return ret;

        //M5. Invoke the method and wait for the reply

    // receive_call_result_data callback will convert blob_attr data to JSON format

        return ubus_invoke(ctx, id, argv[1], b.head, receive_call_result_data,      NULL, timeout *
1000);

}

int main(int argc, char **argv)

{

        const char *ubus_socket = NULL;

        int ch;

        while ((ch = getopt(argc, argv, "cs:")) != -1) {

                switch (ch) {

                case 's':
```

```
                ubus_socket = optarg;
                break;
            default:
                break;
            }
    }
    argc -= optind;
    argv += optind;

    uloop_init();
```
//M1. Connect to ubus daemon and get the connected ubus context
```
    ctx = ubus_connect(ubus_socket);
    if (!ctx) {
            fprintf(stderr, "Failed to connect to ubus\n");
            return -1;
    }
```
//M2. Add the connected fd into epoll fd set
```
    ubus_add_uloop(ctx);
    // call specific ubus method
ubus_cli_call(ctx, argc, argv);

    //When request done, just free the resource, and return
    ubus_free(ctx);
    uloop_done();
    return 0;
```

}

How to use notification

Subscriber

S1. Connect the process to ubus daemon, will get the ubus context, the context will contained the connected fd, registered fd callback and an AVL tree to manage all objects information with this connection

S2. Using uloop utilities to add the ubus_context, which is to register the connected fd into epoll set

S3. Define a subscriber object, which contain a ubus object and a callback to handle received subscribe notification

S4. Add ubus object onto ubus daemon

S5. Specify callback handler to handle notification

S6. Subscribe interested object(notify object)

```
static struct ubus_subscriber test_event;

static void subscriber_main(void)

{

    int ret;

     uint32_t id;

    // S4. Add subscriber object onto bus

    ret = ubus_register_subscriber(ctx, &test_event);

        if (ret)

            fprintf(stderr, "Failed to add watch handler: %s\n", ubus_strerror(ret));

   // S5. Specify callback handler to handle notification

    test_event.remove_cb = test_handle_remove;

    test_event.cb = test_notify;
    // Lookup the notify object

    ret = ubus_lookup_id(ctx, "network.interface", &id);


    // S6.  Subscribe interested object

    ret = ubus_subscribe(ctx, &test_event, id);

    uloop_run();

}

int main(int argc, char **argv)

{

    const char *ubus_socket = NULL;

    int ch;

    while ((ch = getopt(argc, argv, "cs:")) != -1) {

    switch (ch) {

            case 's':

                ubus_socket = optarg;

                break;

            default:

                break;

            }

    }

    argc -= optind;

    argv += optind;

    uloop_init();

    signal(SIGPIPE, SIG_IGN);
    //S1.  Connect the process to ubus daemon

    ctx = ubus_connect(ubus_socket);

    if (!ctx) {

            fprintf(stderr, "Failed to connect to ubus\n");

            return -1;

    }

    //S2.  Add connected fd into epoll fd set.

  ubus_add_uloop(ctx);

 // Subscriber main process

    subscriber_main();


    ubus_free(ctx);

    uloop_done();

    return 0;

}
```

Notification Sender

N1. Connect the process to ubus daemon, will get the ubus context, the context will contained the connected fd, registered fd callback and an AVL tree to manage all objects information with this connection

N2. Using uloop utilities to add the ubus_context, which is to register the connected fd into epoll set

N3. Define a notify object

N4. Add notify object onto bus

N5. Prepare notify type and arguments when actually an event happens

N6. Broadcast the event notification to bus

```
//N3.  Define a notify object

static struct ubus_object test_object ;

static void event_broadcast(char *event)

{

        //prepare event argument if necessary

        // N6.  Broadcast the event notification to bus

        ubus_notify(ctx,  &test_object, event, NULL, -1);

}

int main(int argc, char **argv)

{

        const char *ubus_socket = NULL;

        int ch;

        while ((ch = getopt(argc, argv, "cs:")) != -1) {

                switch (ch) {

                case 's':

                        ubus_socket = optarg;

                        break;

                default:

                        break;

                }

        }

        argc -= optind;

        argv += optind;
        uloop_init();

        //N1.  Connect the process to ubus daemon

        ctx = ubus_connect(ubus_socket);

        if (!ctx) {

                fprintf(stderr, "Failed to connect to ubus\n");

                return -1;

        }

        //N2.  Add connected fd into epoll fd set

        ubus_add_uloop(ctx);

        //N4.  Add notify object onto bus

        ubus_add_object(ctx, & test_object);

       //N5.  Prepare notify type and arguments when actually an  event happens

                ……

        event_ broadcast(event);


        ubus_free(ctx);

        uloop_done();

        return 0;

}
```

The example code can refer to ubus\examples\