Developer's Guide to the Circumference Project

December 2010 — Circumference 1.5

Codename Circumference is an implementation of an extensible Diameter server with (as of this writing) WebAuth module, a WebAuth client for Apache 2, and base libraries.

Contents

I libcircum – A3 API **2**

1 Initialization **2**

2 ID mapping **2**

3 Packet manipulation **3**

4 Connection handling **7**

II libcircum – Circumference API **9**

5 Low-level network I/O **9**

6 Debugging **10**

III libcircumplus **11**

7 Packet manipulation **11**

8 Connection handling **13**

IV circumd – Diameter Server **15**

Part I
libcircum – A3 API

libcircum provides the core operations related to handling Diameter elements like messages building and extraction, name translation, and network I/O. All functions that belong to this API are prefixed with `a3_`[1].

1 Initialization

Explicit library initialization is required to set up a bit of internal state and to read the AVP definition XML dictionaries. Do this at the start of your program using:

```
bool a3_initialize(void);
```

The corresponding exit call is:

```
void a3_deinitialize(void);
```

Both functions together implement proper nesting with refcounting. All users of libcircum should make use of them, including secondary libraries and master programs that directly use libcircum's functions.

2 ID mapping

One key goal of the Codename Circumference project was that it is possible to extend or change the list of known AVP mapping tuples (*names*, *id*) without having to recompile any file, or without causing any file to recompile as part of build dependencies. That rules out providing the definitions as enum constants in `.h` files, or by providing it in a static data array in a `.c` file. As such, Codename Circumference grabs the definitions from standalone XML files. This has the positive side-effect of not cluttering up the namespace in C code (in case of a `.h` file), or unnecessarily bloating up the program (in case of `.c`), and being able to selectively pick XML dictionaries depending on the needs. To make use of the XML dictionaries, a number of mapping functions exist that translate from names to IDs. The reverse direction is not needed all that often, which is why functions might be absent[2].

The function declarations for ID mapping are available in `<libcircum/idmap.h>`.

```
bool a3_dict_register(const char *path);
```

`a3_dict_register` will read the given XML file located at `path`, or if `path` refers to a directory, will read all XML files in that directory (without descending into further subdirectories), and will indicate either success or failure with return values of `true`/`false`, respectively. libcircum will automatically read all XML dictionaries from its shared data directory when the library is initialized with `a3_initialize`. The following resolver functions exist, which translate application names, command names, AVP names, and enum names for AVPs, to their numeric IDs:

[1] We felt this is superior to the camel-casing "`AAAFooBar`" in other specs.

[2] As usual, something is not perfect when there is nothing to add, but when there is nothing left to take away. And currently there are no known users that require ID-to-name translation expect the poor developer seeking for debugging aid.

```
unsigned int a3_idmap_app(const char *application_name);
unsigned int a3_idmap_cmd(const char *command_name);
unsigned int a3_idmap_avp(const char *avp_name);
unsigned int a3_idmap_enum(const char *avp_name, const char *enum_name);
```

In case a mapping is not found, a sensible default value is returned that is recognized as an "unknown value" in accordance with the RFC — usually this is `0` for Application IDs, Command codes and AVP codes, and `-1` for enums[3].

3 Packet manipulation

Packets are represented in memory in a `struct a3_packet`. The structure contains the packet fields as defined in the RFC, the list of AVPs and possibly any other necessary data management parts. To obtain the declarations for packet manipulation, include `<libcircum/packet.h>`.

```
struct a3_packet {
        uint8_t cmdflags;
        uint32_t command;
        uint32_t appl_id;
        uint32_t hbh_id;
        uint32_t ete_id;
        struct HXclist_head avp_list;
};
```

All data in a packet is stored in host byteorder. The AVP list is kept in a linked list so that AVPs can be ordered at will — and remain ordered after insertion.

3.1 Construction

```
struct a3_packet *a3_pkt_new(void);
```

`a3_pkt_new` will create a new empty packet structure. Fields are initialized to sensible defaults (often zero), but no hard assumption should be made. Usually, any fields like command code, application id, hop-by-hop identifier and end-to-end identifier are directly set by the (library) user afterwards. Make sure that you check for `NULL`, which `a3_pkt_new` can return in case of error, and possibly inspect `errno`.

```
struct a3_packet *pkt;
pkt = a3_pkt_new();
if (pkt == NULL)
        abort();
pkt->command  = a3_idmap_cmd("Capabilities-Exchange");
pkt->cmdflags = A3_CMDFLAG_REQUEST;
```

3.2 Command flags

Each packet has a set of command flags that can be set at your convenience. The following constants map directly onto their RFC specification:

`A3_CMDFLAG_REQUEST` "The message is a request."

[3]This is because XML dictionaries do actually define enum constants with value 0.

A3_CMDFLAG_PROXYABLE "The message may be proxied, relayed or redirected."

A3_CMDFLAG_ERROR "The message contains a protocol error." This is usually set for reply packets only.

A3_CMDFLAG_RETRANS "This flag is set after a link failover procedure, to aid the removal of duplicate requests."

A3_CMDFLAG_RESERVED This constant maps to the bitmask of all reserved flag fields.

```
if (packet->cmdflags & A3_CMDFLAG_RESERVED)
        fprintf(stderr, "Packet has flags I do not understand.\n");
```

3.3 Destruction

```
void a3_pkt_destroy(struct a3_packet *);
```

This function will free all resources associated with the packet, including all AVPs contained therein, and their associated data.

3.4 Adding AVPs

```
struct a3_avp *a3_pkt_add_avp(struct a3_packet *pkt,
const char *name, const void *ptr, unsigned int length);
```

The base function for adding AVPs to a packet is `a3_pkt_add_avp`. It takes the name of the AVP, and a pointer to a memory region and its length which will comprise the AVP's data. A group of convenience functions is provided for other inputs:

```
struct a3_avp *a3_pkt_add_int(struct a3_packet *pkt,
const char *name, long long value);
```

Will allow you to pass an integer. Note however that this has no relation to the actual type the AVP uses. You can also `a3_pkt_add_int` for AVPs that are defined as "OctetString". Conversely, you could use `a3_pkt_add_text` for an "Unsigned32" AVP. This gives a certain level of flexibility should the underlying AVP types suddenly change.

```
struct a3_avp *a3_pkt_add_text(struct a3_packet *pkt,
const char *name, const char *value);
```

For enumerated values there is `a3_pkt_add_enum`, which takes the enum constant's symbolic name as third argument. Only symbolic names that are actually defined for the specific AVP as denoted by the second argument, are allowed. All other combinations will evaluate to zero.

```
struct a3_avp *a3_pkt_add_enum(struct a3_packet *pkt,
const char *name, const char *ename);

a3_pkt_add_enum(pkt, "Auth-Request-Type", "AUTHENTICATE_ONLY");
```

All functions return a pointer to the newly-created AVP so that you can directly continue to manipulate it. The first form is used with a `NULL` pointer and zero length to create a grouped AVP. As far as the libcircum implementation goes, whether an AVP is actually grouped depends upon its definition in the Diameter XML dictionaries.

```
struct a3_avp *avpgroup = a3_pkt_add_avp(pkt,
        "Vendor-Specific-Application-Id", NULL, 0);
a3_avp_add_int(avpgroup, "Auth-Application-Id", 1234);
a3_avp_add_text(avpgroup, "Acct-Application-Id", "1234");
```

The functions for adding AVPs to an existing grouped AVP are named similarly, they are:

```
struct a3_avp *a3_avp_add_avp(struct a3_avp *avp,
        const char *name, const void *ptr, unsigned int length);
struct a3_avp *a3_avp_add_int(struct a3_avp *avp,
        const char *name, long long value);
struct a3_avp *a3_avp_add_text(struct a3_avp *avp,
        const char *name, const char *value);
struct a3_avp *a3_avp_add_enum(struct a3_avp *avp,
        const char *name, const char *ename);
```

3.5 AVP structure

The following lists the members of `struct a3_avp` that you need to be aware of. The first block of them maps directly onto their RFC-specified fields. On packet reception via `a3_conn_recv`, an AVP has its blob (the raw data as received from the network) appended in the `blob` member, and its length is stored in `blob_length`. The blob is always available even if it could not be decoded, for example if the AVP type for the code is unknown. For AVPs that can be decoded and are integral or a grouped AVP, the decoded value is stored in the union `d`. OctetString AVPs will keep their data in `blob`, and that is how you access them. `blob` will always be NUL-terminated, so string operations are safe. The trailing security `'\0'` libcircum will insert is not counted in `blob_length`, but any NULs that have been received from the network are.

```
union a3_avp_morph {
        struct HXclist_head avp_list;
        int32_t v_int32;
        uint32_t v_uint32;
        int64_t v_int64;
        uint64_t v_uint64;
        float v_float32;
        double v_float64;
};

struct a3_avp {
        uint32_t code;
        uint8_t flags;
        enum a3_avp_type type;
        uint32_t vendor_id;

        union a3_avp_morph d;
        unsigned int blob_length;
        char blob[0];
};
```

3.6 AVP types

The RFC specifies a number of fundamental AVP types, but the derived ones are not included, because their representation is the same anyway. The ones that libcircum definitely knows about and which you will find in use are:

A3_AVPTYPE_GROUP This AVP is a grouped AVP, and its children are accessible via `d.avp_list`.

A3_AVPTYPE_INT32 Represents a signed 32-bit integer, accessible via `d.v_int32`.

A3_AVPTYPE_UINT32 Represents an unsigned 32-bit integer, accessible via `d.v_uint32`.

A3_AVPTYPE_INT64 Represents a signed 64-bit integer, accessible via `d.v_int64`.

A3_AVPTYPE_UINT64 Represents an unsigned 64-bit integer, accessible via `d.v_uint64`.

A3_AVPTYPE_FLOAT32 Represents an IEEE-754 32-bit floating point number, accessible via `d.v_float32`.

A3_AVPTYPE_FLOAT64 Represents an IEEE-754 64-bit floating point number, accessible via `d.v_float64`.

A3_AVPTYPE_STRING Represents any sort of similar-looking string, including "UTF8String", "OctetString", "DiameterIdentity" and "DiameterURI", and is available in zero-terminated form via `blob`.

A3_AVPTYPE_ENUM Alias for `A3_AVPTYPE_UINT32`.

3.7 Serialization

```
void *a3_pkt_serialize(struct a3_packet *pkt, unsigned int *length);
```

Serialization is the procedure of flattening the tree structure of AVPs and converting the packet into a form that is set forth in the RFC to transfer it over the network. This function will prepare the packet and ensure that flags are propagated before serializing it. It will return a pointer to a memory location with the linearized data, and it is your job to free that in some way when you are done with it. The length of memory region that was allocated is stored in `*length`.

```
struct a3_packet *a3_pkt_unserialize(const void *buffer,
unsigned int length);
```

Unserialization is the opposite transformation. The packet in said buffer of given length will be interpreted and the AVP tree structure is built and returned.

3.8 Locating AVPs and AVP traversal

AVPs can simply be located by the use of the appropriate find functions:

```
struct a3_avp *a3_pkt_find(const struct a3_packet *, const char *name);
struct a3_avp *a3_avp_find(const struct a3_avp *, const char *name);
```

The return value is `NULL` in case of an error or when no AVP by that name was found. `errno` may be set to `EINVAL` by `a3_avp_find` if the AVP passed in is not a grouped AVP. `errno` may also be set to `ENOENT` if the AVP name could not be resolved.

```
const struct a3_avp *avp;
if ((avp = a3_pkt_find(pkt, "Result-Code")) != NULL)
        printf("We have a result code!\n");
```

When there are multiple AVPs with the same name/code, these functions can return any one of them, but at least they will not descend into grouped AVPs. When you do expect multiple AVPs with the same code, you need manual traversal. `struct a3_packet` and `struct a3_avp` use a HXclist to store the linked list of AVPs. HXclist is an inline linked list implementation similar to that of the Linux kernel. I acknowledge that traversal is rather open-coded, but it was designed for fast traversal without any function calls. You start off by declaring an iterator, called `iter` here, followed by invoking the loop construct. `HXlist_for_each_entry` behaves like a for loop, so you are free to use any syntactic optimizations such allows. The second argument to it is the list head, and the third one is the name of the member in the target structure (`a3_avp` in this case) that holds the other side of the "anchor", which, for `a3_avp`, is always "`list`". Interested parties can look into the details of the `list_for_each_entry` implementation in the Linux kernel or libHX.

```
const struct a3_avp *iter;
HXlist_for_each_entry(iter, &pkt->avp_list, list)
        printf("AVP with code %u\n", iter->code);
```

3.9 AVP list manipulations

List manipulation has not been needed so far in our development, but for completeness, it is presented here. You can detach AVPs from their parent object (either packet or parent AVP), and reattach them elsewhere, by simply using the appropriate list operations from libHX. The important thing to remember here is that you first need to delete it from the list before you can re-add it. Failure to do the removal will leave the lists in a state of having no head/sentinel.

```
struct a3_avp *avp = HXlist_entry(pkt->avp_list.next, typeof(avp), list);
HXclist_del(&avp->list);
HXclist_push(&pkt2->avp_list, &avp);
```

4 Connection handling

Interacting with the network is an integral part of the message exchange. Connection setup and teardown, as well as implicit packet serialization on transmission and unserialization on reception are critical tasks for all applications. The header file for connection-related handling is in `<libcircum/net.h>`.

4.1 Connection setup

```
struct a3_conn *a3_conn_new(void);
```

`a3_conn_new` will allocate a new connection structure and initialize it. The reason this is not merged with `a3_conn_open` is the easier error handling, easier error handling for the C++ layer (libcircumplus), since a potential allocation error is not interspersed with errors due to network failures. It also allows to set any extra parameters without having to pass all the parameters into `a3_conn_new`. After initializing the structure this way, you usually set the application ID and whatever is relevant to the Diameter Capabilities-Exchange process.

```
struct a3_conn *conn = a3_conn_new();
conn->appl_id = a3_idmap_app("WebAuth");
```

When done, you give the final call to start connecting:

```
bool a3_conn_open(struct a3_conn *, const char *uri, unsigned int flags);
```

`a3_conn_open` should be used to connect to a given remote host using the given Diameter URI. The remote specification libcircum accepts is an extension to what the RFC allows.

uri	::=	[a3proto "`://`"] host ["`:`" port] [transport] [protocol]
a3proto	::=	"`aaa`" | "`aaas`"
host	::=	hostname | ipv4address | "`[`" ipv6address "`]`"
transport	::=	"`;transport=`" l4proto
l4proto	::=	"`tcp`" | "`sctp`"
protocol	::=	"`;protocol=diameter`"

Figure 1: EBNF for remote targets accepted by libcircum

As extensions to the RFC-mandated URI scheme, libcircum allows omission of the "`aaa://`" prefix, and use of any hostname (not just FQDN) or IPv4/IPv6 it can connect to is permitted. IPv6 addresses need to be enclosed in square brackets to make them distinguishable from the separator for the port number.

There is a contradiction in the RFC — where it demands a reliable transport on page 8 in section 1, but then mentions UDP as a transport on page 48 section 4.3. Since we believe that reliable transport is a good thing, UDP is not implemented as a transport protocol in libcircum.

Also, any protocol other than "`diameter`" is not supported either.

The `aaa://` prefix will establish a normal connection, whereas `aaas://` a secured one, usually by means of TLS**(author?)** [RFC4346].

Accepted strings are, for example:

```
aaas://[2001:db8::1:35]:3868;protocol=sctp
aaa://192.168.1.15:3868
hostname:3868
hostname
```

This makes it possible to specify all the features, but also just a simplistic target host.

The Capabilities Exchange is done as part of `a3_conn_open`. Be sure to have all necessary IDs set up in the `a3_conn` structure beforehand.

4.2 Connection teardown

```
void a3_conn_close(struct a3_conn *);
```

This will send a "Session-Termination" request to the server and await response before closing the actual connection and freeing all internal data associated with the connection.

4.3 Send and receive

`a3_conn_send` is a convenient wrapper for serialization, sending and freeing the packet. `a3_conn_recv` reads the next message and unserialized it, giving a ready-to-use packet, or `NULL` on failure.

```
struct a3_packet *a3_conn_recv(struct a3_conn *);
ssize_t a3_conn_send(struct a3_conn *, struct a3_packet *);
```

Part II
libcircum – Circumference API

5 Low-level network I/O

<libcircum/net.h> also provides three functions for low-level access that is used by the connection code in libcircum itself.

```
ssize_t circum_net_recv(int fd, void *buf, size_t count);
ssize_t circum_net_send(int fd, const void *buf, size_t count);
void *circum_conn_recv_bpkt(struct a3_conn *, unsigned int *length);
```

`circum_net_recv` and `circum_net_send` will receive from or send to the network, using the given file descriptor. These functions directly map to the system calls `recv`(2) and `send`(2), respectively, which are called in a loop to ensure that the full requested amount is retrieved/ sent, due to the semantics of the system calls.

`circum_net_recv_bpkt` will read the next message from the network, and will store it in an appropriately-sized buffer. The length of the message is stored in `*length`, and the buffer pointer is returned. The function will automatically divert to the TLS network code internally if the connection is secured. There is no equivalent function for sending a raw buffer.

5.1 TLS

Servers can use `circum_conn_starttls` to start TLS on the connection after the Capabilities-Exchange and TLS have been negotiated and the "Open state" (see RFC) has been reached. libcircum will automatically call `circum_conn_starttls` during `a3_conn_open` if TLS has been selected, and conversely, will call `circum_conn_leavetls` during `a3_conn_close`.

```
bool circum_conn_starttls(struct a3_conn *, bool is_client);
void circum_conn_leavetls(struct a3_conn *);
```

5.2 Connection helpers for servers

```
void circum_conn_free(struct a3_conn *);
```

Since servers cannot use `a3_conn_open`, they will usually implement their own construction of the server-side `struct a3_conn`. But the exiting side is quite straight-forward since it is all just memory releases. `circum_conn_free` will close the socket descriptor and free all the data in the structure (names for local and remote peer, etc.).

6 Debugging

Over time, a bit of debugging infrastructure accumulated during development. As these functions output directly to `stdout`, and do so with pretty-printing, i. e. indent and colorize where possible, they should really really only be used during and for debugging. The include file needed is `<libcircum/debug.h>`.

```
void circum_hexdump(const void *addr, unsigned int length);
void circum_treedump_pkt(const struct a3_packet *, unsigned int indent);
void circum_treedump_avp(const struct a3_avp *, unsigned int indent);
```

`circum_hexdump` will print out an offset-prefixed, byte-based combined hex and asciidump to stdout. `circum_treedump_pkt` will dump the packet header data, then walk down the list of attached AVPs and call `circum_treedump_avp` for them. `circum_treedump_avp` in turn will recursively descend into the AVP structure and print the AVP metadata and content. To control the dumping behavior of libcircum, the `circum_debug` function can be used.

```
unsigned int circum_debug(unsigned int flags);
```

Because this is really implementation-dependent, you are advised to look into `debug.h`.

Part III
libcircumplus

libcircumplus is a C++ wrapper for libcircum. It makes the packet, AVP and connection structures available as classes. libcircumplus was only a side project, so this API is not complete and may even look odd, but it is usable, as libcircumplus's "minitest" program shows.

All functios will use the `A3_` prefix (note the uppercase[4] A).

7 Packet manipulation

The `A3_packet` class looks very much like the lower-level `struct a3_packet`. The declarations are obtained via inclusion of `<libcircumplus/packet.hpp>`.

```
class A3_packet {
public:
        uint8_t cmdflags;
        uint32_t command;
        uint32_t appl_id;
        uint32_t hbh_id;
        uint32_t ete_id;
        std::list<A3_avp *> avp_list;
};
```

Be aware that this documentation explains the behavior of the class, and may differ from the implementation. What is guaranteed that the implementation is usable the same way as described here. Specifically, `cmdflags` and the other members are actually implemented using `uint8_t &cmdflags`, with the reference being linked to the lower-level structure's `cmdflags` for our (the developer who wrote libcircumplus, aka me) convenience. But this is an implementation detail — the propagation into the lower-level structure could have also been deferred, e.g. until serialization.

7.1 Instantiation and destruction

```
A3_packet::A3_packet(void);
```

Instantiation is done using either C++'s `new` operator, or by putting it on the stack.

```
A3_packet *p = new A3_packet();
A3_packet q;
A3_packet q();
p->appl_id  = a3_idmap_cmd("WebAuth");
p->cmdflags = A3_CMDFLAG_REQUEST;
```

The constructor may throw in case the lower-level function `a3_pkt_new` returns `NULL`, so you might want to catch it. Destruction, obviously, is done using the `delete` operator, or when the object goes out of scope.

The packet flags are the same as for libcircum (see section 3.2).

[4]This is needed because otherwise it would clash with `struct a3_packet`, since struct and class names share the same namaespace.

7.2 Adding AVPs

```
class A3_packet and class A3_avp {
public:
        A3_avp *add_avp(const char *name, const void *ptr, unsigned int length);
        A3_avp *add_int(const char *name, long long value);
        A3_avp *add_text(const char *name, const char *value);
        A3_avp *add_enum(const char *name, const char *ename);
};
```

These functions exactly mirror their libcircum counterparts (section 3.4) and have the same name for both the `A3_packet` and `A3_avp` class.

```
A3_avp *avpgroup = pkt->add_avp("Vendor-Specific-Application-Id", NULL, 0);
avpgroup->add_int("Auth-Application-Id", 1234);
avpgroup->add_text("Acct-Application-Id", 1234);
```

7.3 AVP structure

```
class A3_avp {
public:
        std::list<A3_avp *> avp_list;
        uint32_t code;
        uint8_t flags;
        enum a3_avp_type type;
        uint32_t vendor_id;

        union a3_avp_morph d;
        unsigned int blob_length;
        char blob[0];
};
```

This part of the class follows largely that of `struct a3_avp`.

7.4 Serialization

```
class A3_packet {
public:
        void *serialize(unsigned int *length) const;
        static A3_packet *unserialize(const void *ptr, unsigned int length);
};
```

These functions map directly to `a3_pkt_serialize` and `a3_pkt_unserialize`.

7.5 Locating AVPs and traversal

```
class A3_packet and class A3_avp {
public:
        A3_avp *find(const char *name) const;
};
```

The semantics for `A3_packet::find` and `A3_avp::find` are the same as for `a3_pkt_find` and `a3_avp_find`, respectively.

Since class `A3_packet` and `A3_avp` use `std::list` to store their AVPs, the `std::list::iterator` or `const_iterator` class is to be used for manual traversal.

```
for (std::list<A3_avp *>::const_iterator iter = pkt->avp_list.begin();
     iter != pkt->avp_list.end(); ++iter)
{
        const A3_avp *avp = *iter;
        printf("code %u\n", avp->code);
}
```

7.6 AVP list manipulations

A missing feature of libcircumplus is actually list manipulation. Just removing and reinserting AVPs within the `std::list` will not be sufficient, as the low-level AVPs are still linked in their original order.

8 Connection handling

```
#include <libcircumplus/net.hpp>

class A3_conn {
public:
        A3_conn(void);
        bool ctor_ok(void) const;
        bool open(const char *);
        A3_packet *recv(void);
        ssize_t send(A3_packet *);

        char *&session_id(void) const;
        uint32_t &appl_id(void) const;
};
```

Again, functions map directly the `a3_conn_*` group. After construction, `ctor_ok` is required to be checked and be acted upon, as the constructor will not throw due to the lower-level implementation. You will need to make sure that the destructor is called, too, in case `ctor_ok` failed.

```
static bool try_connect(void)
{
        A3_conn *c = new A3_conn();
        if (!c->ctor_ok()) {
                delete c;
                return false;
        }
        c->appl_id() = a3_idmap_app("FooBar");
        if (!c->open("aaas://localhost")) {
                delete c;
```

```
                return false;
        }
        return true;
}
```

Part IV
circumd – Diameter Server

The Circumference Diameter daemon is extensible by means of modules implemented as shared libraries, much like other projects such as Apache httpd2 or the OpenLDAP server. Each module must contain a metadata/VFT structure with a well-known name so that the server can look it up. The structure gets included through <circumd/module.h>:

```
struct circumd_module {
        const char *app_name;
        unsigned int app_id;

        int (*init)(void);
        unsigned int (*packet)(struct a3_conn *conn,
                               struct a3_packet *pkt);
        void (*exit)(void);
};
```

The members are not ordered in any particular order and it is required to use C99 named initializers for its definition:

```
struct circumd_module circum_module_data = {
        .app_name = "FooBar",
        .app_id   = 1234,
        .init     = foobar_init,
        .packet   = foobar_packet,
        .exit     = foobar_exit,
};
```

This structure must not be marked const, as some internal members will be modified by the server core during load.

The first two members `app_name` and `app_id` describe the module by a name, which is used for displaying errors, and the ID for this particular Diameter Application that is being implemented. This ID can actually be left out and instead be set in the initialization routine that is specified with the `init` member, or both can be combined. An Application ID of zero is considered special and acts as a wildcard. `init` gets called shortly after the module has been loaded, where the module can set itself up, allocate memory, and do whatever is necessary for initialization. It has to return a positive non-zero value to indicate success, negative values encode `errno`. An example:

```
static FILE *foobar_logfp;
static char secret[32];

static int foobar_init(void)
{
        if ((foobar_logfp = fopen("/var/log/foobar.log")) == NULL)
                return -errno;
        if (RAND_bytes(secret, sizeof(secret)) != sizeof(secret))
                return -EIO;
        return 1;
}
```

Similarly, each module can provide an `exit` function that is called when the server shuts down:

```
static void foobar_exit(void)
{
        fclose(foobar_logp);
}
```

The only other function to date is the `packet` function, which gets invoked for every received Diameter message that carries the same Application-Id as the module registered for, or when the registered ID is, as previously mentioned, zero. This for example makes it possible to write modules that take no action directly related to the packet (people sometimes call these modules "watchers"), for example logging every received Diameter message to a remote file. The possibilities lie within the developer and his/her ideas. The `packet` function will have to return a verdict that tells the server core how to proceed.

```
static unsigned int foobar_packet(struct a3_conn *conn,
                                  struct a3_packet *pkt)
{
        fprintf(foobar_logp, "Seen packet with appl_id %u\n", pkt->appl_id);
        return CMOD_CONTINUE;
}
```

The server connection structure will be passed in `conn`, in case the module wants to send a packet back on its behalf. If it does so, the verdict must be `CMOD_STOP`, to tell the caller to stop traversing the module list. `CMOD_CONTINUE` can instead be used to let the caller continue, but in this case, no reply to the packet may be sent by the module, so as to not confuse the connected client. If none of the loaded modules returned `CMOD_STOP` for the packet, the server sends back an appropriate error message to the client, as no module generated a corresponding reply.

The module gets a `struct a3_conn`, in case it takes care of answering the message in `pkt`.

```
static unsigned int foobar_packet(struct a3_conn *conn,
                                  struct a3_packet *pkt)
{
        struct a3_packet *reply = a3_pkt_new();

        a3_pkt_add_enum(reply, "Result-Code", "Success");
        a3_conn_send(conn, reply);
        return CMOD_STOP;
}
```

Appendix

References

[RFC6733] RFC 6733: Diameter Base Protocol
V. Fajardo, J. Arkko, J. Loughney, G. Zorn, 2012-10
http://tools.ietf.org/html/rfc6733

[RFC4346] RFC 4346: The Transport Layer Security Protocol (TLS)
T. Dierke, E. Rescorla
http://tools.ietf.org/html/rfc4346

Index

A
A3_avp class, 12
a3_avp structure, 5
a3_avp_add_avp function, 5
a3_avp_add_enum function, 5
a3_avp_add_int function, 5
a3_avp_add_text function, 5
a3_avp_find function, 6
a3_avp_morph union, 5
A3_AVPTYPE_* enumeration list, 6
A3_AVPTYPE_ENUM type, 6
A3_AVPTYPE_FLOAT32 constant, 6
A3_AVPTYPE_FLOAT64 constant, 6
A3_AVPTYPE_GROUP constant, 6
A3_AVPTYPE_INT32 constant, 6
A3_AVPTYPE_INT64 constant, 6
A3_AVPTYPE_STRING constant, 6
A3_AVPTYPE_UINT32 constant, 6
A3_AVPTYPE_UINT64 constant, 6
A3_CMDFLAG_* flags, 3
A3_CMDFLAG_ERROR flag, 4
A3_CMDFLAG_PROXYABLE flag, 4
A3_CMDFLAG_REQUEST flag, 3
A3_CMDFLAG_RESERVED flag, 4
A3_CMDFLAG_RETRANS flag, 4
A3_conn class, 13
a3_conn_close function, 8
a3_conn_new function, 7
a3_conn_open function, 8
a3_conn_recv function, 8
a3_conn_send function, 8, 16
a3_deinitialize function, 2
a3_dict_register function, 2
a3_idmap_app function, 3, 8
a3_idmap_avp function, 3
a3_idmap_cmd function, 3
a3_idmap_enum function, 3
a3_initialize function, 2
A3_packet class, 11, 12
a3_packet structure, 3
a3_pkt_add_avp function, 4
a3_pkt_add_enum function, 4
a3_pkt_add_int function, 4
a3_pkt_add_text function, 4
a3_pkt_destroy function, 4
a3_pkt_find function, 6
a3_pkt_new function, 3
a3_pkt_serialize function, 6
a3_pkt_unserialize function, 6
aaa://, 8
aaas://, 8

B
blob, 5

C
Capabilities-Exchange, 8
circum_conn_free function, 9
circum_conn_leavetls function, 9
circum_conn_starttls function, 9
circum_debug function, 10
circum_hexdump function, 10
circum_net_recv function, 9
circum_net_recv_bpkt function, 9
circum_net_send function, 9
circum_treedump_avp function, 10
circum_treedump_pkt function, 10
circumd/module.h file, 15
circumd_module structure, 15
CMOD_CONTINUE constant, 16
CMOD_STOP constant, 16

D
DiameterIdentity type, 6
DiameterURI type, 6

F
FQDN, 8

G
grouped AVP, 6

H
HXlist_for_each_entry macro, 7

L
libcircum/debug.h file, 10
libcircum/idmap.h file, 2
libcircum/net.h file, 7, 9
libcircum/packet.h file, 3
libcircumplus/net.hpp file, 13
libcircumplus/packet.hpp file, 11

O
OctetString type, 6

S
serialization, 6, 8

Session-Termination, 8
`std::list`, 13

T
TLS, 8
traversal, 6

U
UDP, 8
unserialization, 6, 8
URI, 8
UTF8String type, 6