Advances in Samba4

Volker Lendecke
VL@Samba.ORG

August 23, 2004

1 Introduction

Samba 4 has been in development for about 18 months now. It is a complete start from scratch, hardly anything in the core Samba code is used commonly between Samba 3 and Samba 4. Why was this done?

Samba 3 shows its age. Essentially, it is still the very first implementation started by Andrew Tridgell more than a decode ago. Certainly it has seen a huge amount of improvement since that first implementation, but the internal structure has not really changed during its development.

Since its start, the community around Samba has gained a lot of knowledge around the protocols Windows implements, but the implementation in Samba shows traces of the early phases when we did not know as much about these protocols. One good example is the implementation of the Microsoft Remote Procedure Calls, these days base for most of the extended functionality Microsoft Servers implement. When they were first implemented, it was done in a completely manual manner, as Samba has always been done: Push around bits until Windows accepts our stuff. This is closely reflected in the code, seen in the subdirectory `rpc_parse`.

Samba 4 was started to correct this mistake, to get the infrastructure right without having to take care of compatibility to existing code. This talk will cover four areas of code where Samba 4 is an improvement over Samba 3:

- The `talloc` memory allocator has been simplified and enhanced.
- The MSRPC subsystem has been completely rewritten, Samba is now at the same level as Microsoft was ten years ago.
- The `ldb` library is a big enhancement over the `tdb` databases introduced in Samba 2.

- The LDAP libraries normally used are in the process of being completely replaced by something we believe to be more flexible and better fitting into the Samba framework.

2 talloc

Managing dynamic data structures with `malloc()` can become a real nightmare when the structures become even moderately complex. C++ offers the concept of automatic destructors that can help a lot, even higher-level languages offer automatic garbage collection. It was never discussed in the Samba Team, but rewriting Samba in a different language than C has never been seen as an option. The one big advantage we get by using C is the wide availability of Compilers across the different Unix platforms Samba is ported to.

After reading this section, you might be a bit disappointed, you might ask yourself — what's the point? Where is the major innovation? Every major software project must have its own way to manage dynamic memory, this is just the way Samba does it.

In Samba 2 the Trivial Allocator `talloc` was introduced to make memory management a lot easier. The idea is absolutely simple: Initialize a `TALLOC_CTX` and use `talloc(ctx, size)` instead of `malloc`. Then after you're done, simply `talloc_destroy` your context, and all of the memory you have talloc'ed is gone immediately.

Why has this helped Samba so much? Samba is a rather simple daemon, it gets a request and has to reply to it. Samba 3 has been using this extensively in the MSRPC subsystem, and Samba 4 much more does so. When a request comes in, a talloc context is initialized for this request, and all routines called deep below can allocate memory using this talloc context. When the final response to the request has been sent out, the per-request talloc context is destroyed, and none of the internal routines have to care about freeing memory.

Samba 4 takes this to the extreme. `malloc` is used almost exclusively inside `talloc.c`, everyone allocating memory has to decide which context the memory has to be assigned to. Each smb request has his own structure, together with a private talloc context:

```
struct smbsrv_request {
        /* a talloc context for the lifetime of this request */
        TALLOC_CTX *mem_ctx;

        /* the server_context contains all context specific
           to this SMB socket */
        struct smbsrv_connection *smb_conn;

...
};
```

A common theme in Samba 4 was the following:

```
mem_ctx = talloc_init("request_context");
...
req = talloc(mem_ctx, sizeof(*req));
if (!req) {
        return NULL;
}

ZERO_STRUCTP(req);

/* setup the request context */
req->smb_conn = smb_conn;
req->mem_ctx = mem_ctx;
```

Every important and complex struture that has to live longer than a few lines of code gets its own talloc context.

Looking at current Samba 4 code, the snippet above is not entirely correct anymore. Tridge and the rest of the people do not hesitate to change a lot of internals of Samba 4 when it seems appropriate. Inspired by the Hierarchical allocator `http://swapped.cc/halloc/` the talloc interface has changed and simplified. The recurring pattern mentioned above inspired the idea that every piece of memory can be its own talloc context usable by further calls to `talloc()`.

What does this mean? You allocate some memory using `talloc`, and you decide whether it should die whenever another structure dies. A hypothetical example might be a structure representing an open file:

```
struct open_file {
  int fd;
  size_t length;
  const char *name;
};
```

The `struct open_file` itself is to be freed when the file is closed. The corresponding `close` routine has to make sure that the member `name` is also freed. The new talloc implementation makes this simpler: Use talloc for the main structure and the file name, and talloc_destroy will take care of freeing the file name for you:

```
struct open_file *f = talloc(NULL, sizeof(struct open_file));
f->name = talloc_strdup(f, "filename");
...
talloc_destroy(f);
/* f->name is gone as well */
```

This works in a hierarchical manner. Every talloc'ed piece of memory can be the root of more talloc'ed memory, destroying the root automatically frees all children. Everyone who has been bothered to do kerberos programming will see that this can simplify the APIs a lot. Free all memory that is used for a particular request with a single call can be a huge relief.

For example, when you allocate a string representing the name of a file just opened, you might decide that this string needs to be freed when the file is

3 The MSRPC-subsystem

Most of the functionality Microsoft servers offer beyond basic file sharing is using a variant of the Distributed Computing Environment Remote Procedure Calls (DCE-RPC). Even printing, which had traditionally been part of the basic CIFS protocol is using RPCs since Windows NT4. All of the domain functionalities that Windows offers is using RPCs, as does DCOM. Modern Windows clients simply expect basic RPC functionalities, even if you only want to share files.

The initial implementation of the Microsoft variant of DCE-RPC, here called MSRPC was driven by the need to become a member in a Windows NT4 domain, thus to implement the domain controller protocol of NT. At times before Samba 2.0, "security = server" was a bad hack to hand password queries to another server. Microsoft had invented a better method to ask a separate server whether a user had typed in his password correctly, but this method is based on MSRPC. With Samba 2.0, the first implementation of the MSRPC subsystem was published as stable.

This initial implementation was done exactly the same way as Samba has always been done: Tweak the bits we send and receive until Microsoft systems accept our language. This approach however completely neglects that there is a well-defined structure in the MSRPC system.

All of the Remote Procedure Calls Windows implements are defined in an Interface Definition Language (IDL), the relevant encoding on the wire is called Network Data Representation (NDR). Specifications for IDL as well as NDR are nowadays freely available on the net. At the time Samba 2 was written this was different: There have been no exact and specifications, although the general structure of the code Microsoft implemented was well-known.

When you look at the subdirectory rpc_parse in Samba 3, you see a lot of code that implements basic encoding and decoding of MSRPC requests and responses. This is a manual process that inevitably leads to subtle bugs in the data that Samba 3 sends and receives on the wire. Especially nasty are the padding rules. At which point do I have to increment a pointer to a multiple of 4, where can I just put my additional data on the wire? All of this stuff can be automatically be deduced from a brief high-level description in the Interface Definition.

How does this translate into code? Lets look at a very simple call from the SAMR interface. SAMR implements the remote interface to the Windows user database. The following piece of Samba 4 IDL code defines a call to delete a member from a domain group:

```
/************************/
/* Function    0x18     */
NTSTATUS samr_DeleteGroupMember(
        [in,ref]                  policy_handle *handle,
        [in]                      uint32 rid
        );
```

You have to pass in a so-called policy handle, an opaque data structure that you got back from the server by an OpenGroup call. The other argument is a 32-bit unsigned integer representing a group member. To implement the encoding and decoding routines for this simple call, in Samba 3 about 40 lines of code have been necessary. The core routine is the encoding of the request:

```
BOOL samr_io_q_del_groupmem(const char *desc,
                            SAMR_Q_DEL_GROUPMEM * q_e,
                            prs_struct *ps, int depth)
{
        if (q_e == NULL)
                return False;
        prs_debug(ps, depth, desc, "samr_io_q_del_groupmem");
        depth++;
        if(!prs_align(ps))
                return False;
        if(!smb_io_pol_hnd("pol", &q_e->pol, ps, depth))
                return False;
        if(!prs_uint32("rid", ps, depth, &q_e->rid))
                return False;
        return True;
}
```

Samba 4 implements pidl, the Perl IDL compiler. This compiler takes the IDL file describing all the code and auto-generates all the encoding and decoding routines, along with the function stubs necessary to use the routines. The amount of compiled object code that Samba 4 uses to implement the same functionality as Samba 3 provides is not less than in Samba 3, but the volume and complexity of the source code is vastly reduced. The complete SAMR encoding and decoding code in Samba 3 is a little less than 250 Kilobytes of hand-written code:

```
vlendec@delphin:/data/3_0> ls -l include/rpc_samr.h
rpc_parse/parse_samr.c
-rw-r--r-- ...  46165 2004-06-14 12:01 include/rpc_samr.h
-rw-r--r-- ... 203236 2004-06-14 12:01
rpc_parse/parse_samr.c
vlendec@delphin:/data/3_0>
```

In Samba 4 only about 10% of that code is used to implement even more functionality:

```
vlendec@delphin:/data/4_0/librpc/idl> ls -l samr.idl
-rw-r--r-- ...  30614 2004-08-14 03:11 samr.idl
vlendec@delphin:/data/4_0/librpc/idl>
```

You may ask with good reason why we haven't done this ages ago. The only answer to this is lack of knowledge, too much other stuff to do, history. However, with these foundations it should be a lot easier to follow what Microsoft does in the future as it has been before.

4 ldb

One of the internal changes from Samba 2.0 to Samba 2.2 has been the introduction of the trivial database tdb. Why was this necessary, and what is it used for?

The multiple `smbd`s on a running Samba server have to communicate in multiple ways. One example are the Windows-compatible byte range locks that are wildly different from the ones Posix offers. These byte range locks have to be implemented by Samba itself. Every file that is byte range locked by some client is locked for all other users of that file, even if the other users are working on a different client machine. Correctness and speed of these locks are absolutely critical. So Samba needs a way to inform all `smbd`s about the state of the byte range locks.

This is done using the tdb files. A tdb database essentially is a multi-writer hash table that loosely resembles the Berkely DB. At the time tdb was invented, Berkely DBs were single-writer only however. So tridge implemented a database library for Samba with its own API. Nowaday tdb files are used for many sorts of things, for example winbindd uses it as a storage for its mapping from foreign SIDs to local group and user IDs, and Samba as a domain controller can store the user specific data in a tdb.

As flexible as tdb's are, they have their limitations. They are tables mapping arbitrary keys to arbitrary values. Period. Every record in a tdb is a binary object with a single key. When you want to store complex data in a tdb, you have to convert the data structure into a byte stream before storing it, and get the data back into the structure after retrieving the byte stream. This is awkward in two respects: First you have to make sure that you have the correct endianness if your tdb file is to be ported across different architectures, and second it makes extending the structures rather difficult. You manually have to take care of version upgrades. Doing this is right is possible but error-prone.

Another problem with tdb files is the restriction to a single index. When looking at the user database of a Samba server, this needs to be indexed by at least three values: The user name, the SID and the unix uid. With Samba 3 this is done manually whenever a tdb needs to be indexed by more than one attribute. One of the attributes is chosen as the primary index for the databse, and the other two ones are maintained as separate records pointing at the primary key. Again, an error-prone process repeated again and again in Samba 3.

In Samba 4, tridge has taken a different approach. He has stripped the data model of LDAP to its bare core, and has implemented a limited implementation of an API close to the LDAP one on top of tdb. This new implementation is called ldb.

An ldb is a database of objects with arbitrary attributes, all of which can have arbitrary values, just like an LDAP database without any schema checking. Part of the ldb library is an ldif parser and printer, so that a text-based dump of your ldb database is easily possible.

Searching a ldb database is very similar to searching a LDAP database, the corresponding call is:

```
int ldb_search(struct ldb_context *ldb,
               const char *base,
               enum ldb_scope scope,
               const char *expression,
               const char * const *attrs, struct ldb_message ***res)
```

A typical search for an object might look like the following:

```
struct ldb_message **admins;
const char * const *attrs = { "displayName", NULL };
result = ldb_search(ldb, "dc=samba,dc=org", LDB_SCOPE_SUBTREE,
                    "(objectSid=S-1-5-32-544)", attrs, &res);
```

This example would do a search for the Administrators alias in a hypothetical ldb containing an Active Directory compatible user database. You can use complex filter strings that look for different attributes at the same time, you can change objects, and the indices are naturally updated correctly. When you add a new index, the library automatically re-indexes all database entries. The API falls back to a complete traversal of the database in case you are looking for an attribute that has not been indexed.

The AD compatible user database is the first and very obvious application this ldb API is used for. The Samba 3 notion of a passdb that stores additional attributes of a Unix user has been completely removed in the design of Samba 4. Samba 4 will implement a complete Windows compatible user database independent of the underlying Unix. Windows compatibility these days however means that you have to present your data compatible to what Windows expects as active directory. Samba 4 will store the data as exactly as possible like Windows 2003 does, so that the presentation via the LDAP protocol should become easily possible.

The similarity to LDAP is true for all other operations like adding, modifying and deleting entries. Everyone familiar with the LDAP API or even only the command line utilities should feel at home with the ldb API very easily.

To summarize it, in `ldb.h` tridge compares ldb with tdb and ldb with LDAP:

- Major restrictions as compared to normal LDAP:
  - no async calls.
  - each record must have a unique key field
  - the key must be representable as a NULL terminated C string and may not contain a comma or braces.
- Major restrictions as compared to tdb:
  - no explicit locking calls

5 LDAP

In the area of LDAP Samba has proceeded a lot in the last weeks. Initiated by the author of this paper and then improved by Stefan Metzmacher and Simo Sorce a completely new approach to LDAP has been initiated.

At a customer site I came across a winbind problem that turned out to be a LDAP server that would simply hang in a query and not come back. No idea why, it simply would not respond after taking the query. A weekend of frustrating hacking later I decided that the OpenLDAP libraries had given me enough pain that a replacement was necessary. I simply could not find out how to reliably make the OpenLDAP libs return after a maximum time under all conditions.

What I started closely followed the proven Samba 4 concept: Clearly separate concerns. There are at least two things to be taken care of when dealing with an OpenLDAP server:

- Encoding the LDAP query into a stream of bytes, decoding the response.
- Sending the data stream to the LDAP server and waiting for the reply.

The classical LDAP RFC-conforming libraries take care of both aspects at the same time. This poses some particular problems, even within Samba 3. Winbind for many queries has to talk to LDAP servers, for example getting the user list out of an AD Domain Controller is done via LDAP. Winbind however needs to be responsive even when a DC freaks out or is just slow.

The LDAP queries in question have to be embedded into an asynchronous server architecture as much as possible. A big select() loop needs to take care of any operation that might block, in particular the queries to external servers. An initial result of these considerations are two routines `ldap_encode()` and `ldap_decode()`. They convert high-level structures that correspond to LDAP queries into a stream of bytes and vice-versa.

The request structure for an LDAP search request for example looks like the following:

```
struct ldap_SearchRequest {
        const char *basedn;
        enum ldap_scope scope;
        enum ldap_deref deref;
        uint32 timelimit;
        uint32 sizelimit;
        BOOL attributesonly;
        char *filter;
        int num_attributes;
        const char **attributes;
};

union ldap_Request {
...
        struct ldap_SearchRequest       SearchRequest;
...
};

struct ldap_message {
        TALLOC_CTX              *mem_ctx;
        uint32                   messageid;
        uint8                    type;
        union  ldap_Request      r;
        int                      num_controls;
        struct ldap_Control     *controls;
};
```

A sample query for the root-DSE entry is set up the following way:

```
msg = new_ldap_message();
msg->type = LDAP_TAG_SearchRequest;
msg->r.SearchRequest.basedn = "";
msg->r.SearchRequest.scope = LDAP_SEARCH_SCOPE_BASE;
msg->r.SearchRequest.deref = LDAP_DEREFERENCE_NEVER;
msg->r.SearchRequest.timelimit = 0;
msg->r.SearchRequest.sizelimit = 0;
msg->r.SearchRequest.attributesonly = False;
msg->r.SearchRequest.filter = "(objectclass=*)";
msg->r.SearchRequest.num_attributes = 0;
msg->r.SearchRequest.attributes = NULL;
```

The structure `msg` is then passed to the routine `ldap_encode`, which has the following prototype:

```
BOOL ldap_encode(struct ldap_message *msg, DATA_BLOB *result);
```

DATA_BLOB itself is another Samba-ism, but a very simple one:

```
typedef struct data_blob
{
        uint8 *data;
        size_t length;
        void (*free)(struct data_blob *data_blob);
} DATA_BLOB;
```

The result of `ldap_encode` is the ASN1-encoded representation of the ldap query you encoded in a high-level fashion by setting up all the values in the `struct ldap_message`. This can then easily be sent to the LDAP server.

As you can see, this is a bit more verbose than the usual RFC-libs, but this verbosity pays off later. It's a lot more flexible. For example it was trivial to implement a set/get/endent-style interface to LDAP-searches:

```
if (!ldap_setsearchent(conn, msg, NULL)) {
        printf("Could not setsearchent\n");
        return -1;
}
while ((result = ldap_getsearchent(conn, NULL)) != NULL) {
        struct ldap_SearchResEntry *r = &result->r.SearchResultEntry;
        int i;
        printf("dn: %s\n", r->dn);
        for (i=0; i<r->num_attributes; i++) {
                int j;
                printf(" %s\n", r->attributes[i].name);
                for (j=0; j<r->attributes[i].num_values; j++) {
                        printf("  %s\n",
                               r->attributes[i].values[j].data);
                }
        }
}
ldap_endsearchent(conn, NULL);
```

In this example, `conn` represents an established connection to the LDAP server, and `ldap_setsearchent` expects a fully set up search message as shown above. To me this seems a lot simpler than messing around with the different RFC-conforming routines for extracting all the search results.

The fact that this interface could be made a lot easier to use than the RFC-libs was greatly helped by the talloc allocator mentioned above. The `ldap_message` can become a rather complex structure that nobody wants to manually `free()` after use. Having all memory for example for the distinguished name, the attributes and values of a search result entry hang off the `ldap_message` itself makes it trivial to get memory management right.

6 Summary

The four components presented in this paper are pieces of the core Samba 4 infrastructure. Glued together they make a Domain Controller that is as capable as Samba 3 is possible to implement with a lot less code than was necessary for Samba 3.

The two big missing pieces before Samba 4 is a Domain Controller that XP accepts as a full Active Directory DC are LDAP support and Kerberos. Samba 4 has to listen on the LDAP port 389 and present the same user database an AD DC presents. This can be done in at least two ways. We could use OpenLDAP and implement a backend that talks to Samba's ldb API, essentially translating the protocol on port 389 into ldb queries. The other alternative would be to use the new LDAP encoding/decoding routines and implement an LDAP server of our own. We might be found guilty of the Not-Invented-Here syndrome, but the OpenLDAP server has proven not to be the easiest piece of software to work with. So Samba 4 might end up implementing all of LDAP itself.

With Kerberos matters are completely different. Contrary to LDAP, Kerberos this is a very security sensitive protocol. History teaches that you should not implement security stuff when it already has been done for you. We have two competing Open Source implementations to choose from: MIT and Heimdal. Probably Samba 4 will use Heimdal to implement the necessary KDC functionality, because Heimdal already implements a pluggable database backend for the key material it uses.

But wait — The bread-and-butter business of Samba has always been file and print, right? This talk has not covered any of these. The reimplementation of the printing stuff should become a lot easier than the first one was in Samba 3, as we now have the MSRPC infrastructure available. File serving currently is the big missing piece in Samba 4, but the infrastructure is in place to make the implementation less painless than in Samba 3. For the absolutely impatient there's a bad hack available: Use a Samba 3 smbd as a file server forked off Samba 4 for each file share connection. For details send mail to vl@samba.org, or browse through the archives of samba-technical@samba.org ;-)