Samba's New Registry Based Configuration

Michael Adam
SerNet GmbH
Bahnhofsallee 1b
37081 Göttingen
Germany
obnox@samba.org

1 Abstract

Samba [1] is the most popular Open Souce software suite that offers file and print services to all kinds of SMB/CIFS clients and integrates UNIX systems into the world of Microsoft Windows networking. Starting with version 3.2.0, Samba offers the ability to store its configuration in the internal registry database as an extension of or even as a substitute for the traditional text file configuration. The implementation is realized via an additional configuration interface layer.

This paper describes the design of the configuration API and the registry based configuration. Application of the the registry configuration for writing configuration GUIs is discussed as well as the use in clustered Samba environments.

2 Introduction

Configuration is a topic that is usually pretty much neglected. Samba has been configured through a text file called `smb.conf` since the beginning of time. The configuration source code module `loadparm` is ugly, but it works, and it has been working quite well since a long time like this. So why touch it?...

But with version 3.2.0, relased on 1 July 2008, Samba has obtained a new configuration system that stores the configuration parameters in Samba's internal registry database. This new configuration backend can be used in combination with or as a substitute for the traditional text configuration backend.

The registry store has the advantage of being easily accessible programmatically through Samba's internal registry API. Furthermore, since Samba's registry is stored in as a *tdb* database (Samba's "trivial database", see [2]), the operations are secured by locks and transactions. The use of *tdb* makes the new configuration backend also especially interesting for use in a clustered environment, where with the help of *ctdb*, the configuration changes are immediately distributed to all cluster nodes. (*ctdb* is the clustered implementation of *tdb*, see [3]).

The fact that the registry is used (instead of any arbitrary *tdb* database), makes the configuration interface available remotely over the WINREG RPC interface without further effort. I.e. those who like it can now edit Samba's configuration with regedit.exe from a windows workstation...

The registry backend has been introduced together with a new interface abstraction layer for accessing Samba's configuration, the `libsmbconf` library. The command line program `net conf` is an example application using this interface. It provides a dedicated tool for reading and writing Samba's registry configuration.

2.1 Outline

This paper describes in detail the design of the original configuration code and the motivation for moving away from the monarchistic text file configuration system towards a database store. The design of the

registry configuration and the different levels of activating it in Samba 3.2 are presented next. After that, the `libsmbconf` API layer is discussed as well as the `net conf` tool as an example `libsmbconf` application. Then, a special purpose `libsmbconf` application, the `netdomjoin-gui` is introduced. Next, clusterd Samba using *ctdb* is briefly introduced and the application of the registry configuration for configuring all nodes of a samba cluster simultaneously is presented. Finally, the current and ongoing work on registry configuration is discussed.

3 Traditional Configuration: The `smb.conf` File

In order to describe the new features related to registry configuration and the new configuration abstraction layer, we first need to paint a picture of how configuration is done in Samba before version 3.2.

Traditionally, Samba is configured with a text file usually called `smb.conf`. The file is in `win.ini` style format. A typical `smb.conf` file might look like this:

```
[global]
  netbios name = fileserver
  work group = samba
  passdb backend = user

  ; debugging options
  log level = 3
  max log size = 10000
  debug hires timestamp = yes

  # include client specific configuration
  include = /etc/samba/smb.conf.%I

[homes]
  valid users = %S
  brosweable = no
  writeable = yes

[share1]
  path = /data/share1
  read only = no
```

3.1 The file format

The format of the `smb.conf` file is described in detail in the `smb.conf(8)` manual page. But here are the essentials. Logically, the configuration consists of sections and parameters. The file format is line-based: Each line represents the beginnig of a new section, a parameter or a comment. Lines beginning with a semicolon (;) or a hash sign (#) are comment lines. The start of a new section is indicated by a name in square brackets, like `[global]`. Each section consists of one ore more configuration parameters. Parameters are associated to the section that was last started. Sections can be started multiple times, and parameters for one section accumulate over all mentions of that section. Parameters may specified multiple times for one section as well. The last mention of one parameter overrides earlier occurences. A parameter consists of a parameter name and the assigned value separated by an equals sign (=). Only the first equals sign in a line is significant.

The `[global]` section has a special meaning: While all *other* sections define shared resources (shares), where the name of the section is the name of the share to be defined, the `[global]` section contains the configuration options that control the global behaviour of Samba and defaults for the share definitions (the other sections).

The [homes] section has a special meaning, too: It lets the server create a home share for the connecting user on the fly. But these special semantics are irrelevant for our considerations.

3.2 Includes

The include directive is not a parameter but a meta directive. It does not control the behaviour of Samba or the properties of a share but the distibution of the configuration over several source files. Include directives are evaluated as the smb.conf parser encounters them. If the file is found, then the include statement is effectively replaced by the contents of the include file.

3.3 Macros

The configuration is not completely static. A dynamic aspect is added by the so called macros that are expanded at runtime by data that can for instance depend on the client connection. Table 1 lists some of the most important macros.

macro	substituted with
%I	the client's IP address
%m	the client's NetBIOS name
%U	the session username
%S	the name of the current service (if any)

Table 1: List of important macros

Each client connecting to Samba spawns a new samba daemon (smbd) that reads the configuration from scratch. Hence, by combining the use of includes and macros, one can effectively produce client specific configurations. One typical and very useful applicatoin of this is for debugging client connections in production environments: One can have client-specific log files and set enhanced debugging options only for individual clients, keeping the overall debug load on the server at a minimum.

3.4 loadparm

The parameter loading code is located in the Samba source folder param/, in the module loadparm.c. the actual parser is param/params.c. This code had seen few modifications since a long time. Here parsing code and activation of the parsed parameters is not well separated, but parameters are interpreted as soon as they are read. The main parsing routine pm_process() takes callbacks for parsed section starts and parameters. This is its prototype:

```
bool pm_process(const char *filename,
                bool (*section_fn)(const char *),
                bool (*param_fn)(const char *, const char *));
```

In loadparm, the main configuration loading function lp_load() is in principle working like this:

1. Do macro substition of on file_name.
2. Call pm_process(file_name, do_section, do_parameter)

The do_section callback creates adds service structures for new services to the internal services array of loadparm, and do_parameter fills the either the global parameters or parameters of a corresponding service in the services array with the given values.

The Samba daemons regularly check the time stamps of all configuration files that constitute their current configuration, and re-read from the start if time stamp has changed since the configuration was last read.

3.5 Disadvantages

While the procedure described above has worked well for many years in production now, and still works well, there are several disadvantages of this design.

The major disadvantage is memory consumption. Each `smbd` process reads all of the configuration and builds up the corresponding services array when it is created, i.e. when a client connects. Now there are nearly 150 service-specific parameters, some of which are strings. On a 64 bit machine, a service entry typically occupies around one kilobyte of memory. Now in larger production environments, there are setups with hundreds, even tousands of shares defined. Hence the `smbd` for a client connecting to a specific share may well allocate one or even several megabytes of service data, that are plainly not needed. Now when in such an environment with one thousand shares, one thousand clients are connected to the server, this means that one gigabyte of RAM is wasted just with unneeded data. Even worse: when one changes just one global parameter in the `smb.conf` file on that system (or merely touches the file), the thousand `smbd` processes simultaneously start loading the configuration file which will for one thousand shares typically be some 250 kilobytes large. So this can present a serious bottleneck for I/O performance to the server. Furthermore, checking whether a newly parsed service section corresponds to an already allocated service structure or if a new service structure has to be allocated is perfomance critical for a thousand processes parsing a configuration with a thousand share definitions.

Note that these considerations are not purely academical. Production servers have become unresponsive for many minutes after changing the `smb.conf` file. Workarounds have been developed to speed up `lp_load()` by hashing the services array and to prevent the `smbd` processes from re-loading the configuration simultaneously by distributing this randomly over a small period of time. But these workarounds are just band-aids and do not eliminate the real cause.

Another disadvantage is the ease of use and lies in the nature of a text file used for configuration. While it is of course very easy and convenient for the administrator to open the configuration file with a text editor and modify it, there are some drawbacks here:

- When changing *one* parameter, the whole configuration needs to be written and re-read.
- When writing application such as GUIs for configuring Samba, one always has to write whole configuration files. There is no procetion for concurrent access to the configuration file by multiple porocesses. For a certain level of safety, applications need to prepare their own temporary version of the configuration file and copy it over the real one. Other applications may overwrite your changes.
- In clustered environments, one needs to copy the configuration file to all cluster nodes after making changes. Apart from that, the problem of concurrency is even bigger in this kind of setup.

All of these disadvantages can be overcome by using a database instead of (or in addition to) a plain text file for storing configuration parameters.

4 Towards Registry Configuration

Samba has an internal registry database. Some of the data stored in the registry is needed for communication with Windows clients. The registry is made available on the network over the WINREG rpc pipe.

4.1 The Windows Registry Data Model

The registry data model is as follows. The data is organized in a tree structure of *keys*. A key has a name and consists of the list of its subkeys and the list of its *values*. A registry value consists of its name and the value data. There are several possible data types for value data such as a string type,

several integer types and a binary type. For our purposes only the string type `REG_SZ` and the list of strings type `REG_MULTI_SZ` are important. The registry is split into a number of logical sections called *hives* that correspond roughly to the base keys (keys without a parent key). One very important hive is `HKEY_LOCAL_MACHINE` or `HKLM` for short. For more information on the registry, see the MSDN document [6] or the Wikipedia article [7].

The key-value data structure of the registry indicates that a registry key is a very natural candidate for storing Samba configuration data (or more generally any data from a file in `win.ini` format): Each configuration section corresponds to a subkey of the configuration key, and each parameter in a section corresponds to a registry value in the corresponding subkey. Since Windows and applications store configuration data under some subkey of the key `HKLM\Software`, it seems like a natural thing to use the key

```
HKLM\Software\Samba\smbconf
```

called *the smbconf key* in the sequel, to store Samba's configuration in registry.

There are two fundamental differences between a text configuration file and the registry database, though, that need to be taken into account.

- In a registry key, each value of a given value name occurs precisely once. So in registry configuration, one has to make sure that value names are canonicalized, and synonyms are taken care of.
- The order in which the values and subkeys are listed is unpredictable, usually by creation time (oldest first).

The only case where the order really matters and multiple mention is vital in the configuration text file is that of includes. This is because includes are not parameters but meta directives as detailed above. When includes are mixed with normal parameters, this can be used to create configurations where the placement of the include directives in the configuration file subtly matters.

Therefore, includes need a special handling for storing configuration in registry, and one needs to accept one drawback. The model is this:

- Includes of a service are stored in a special registry value called `includes`. This value is of type `REG_MULTI_SZ`, i.e. a list of strings. Each entry in this list is the path of one include file.
- To keep the simplicity of the data model for the normal parameters, one sacrifices the possibility to intersperse includes between the parameters. (Otherwise, an artificial ordering of the parameters would have to be introduced.) One further nails down the rule that that includes are always evaluated *after* the normal parameters have been activated.

Apart from these considerations, there are no problems in using the registry for storing configuration data. Note that Samba's registry is stored in a *tdb* database, `registry.tdb`. Therefore, concurrent write access is protected by locks and transactions.

4.2 Step One: Registry Shares

As a first step in the realization of this idea, Volker Lendecke introduced *registry shares*. These are stored under the smbconf registry key in the mapping of services and parameters to keys and values described above. This first step did not touch `loadparm` except for the introduction of the new boolean parameter `registry shares` that can be used to activate the support for registry shares, but made these registry shares available by the server on demand through two functions:

- `load_registry_service()` is used in `find_service()` to try and load one service definition from registry when a service is not found in the services array previously built when parsing the `smb.conf` file.

- `load_registry_shares()` to load all service definitions from registry when the list of shares is requested via the `NetShareEnum` and `NetShareEnumAll` rpc calls.

With this initial implementation that is still available in Samba 3.2, a mixed configuration with global options in `smb.conf` and shares exclusively defined in registry can be achieved with a `smb.conf` file like this:

```
[global]
  netbios name = nirvana
  workgroup = samba
  ...
  registry shares = yes
```

This has the big advantage that the `smbd` processes only need to read the relatively small global section at startup, and no services array is allocated a priori. Service structures are allocated on demand only when a client connects to a share or requests the list of shares.

While this essentially solved the memory consumption and performance problem of the static text file configuration, it was originally not very practical for the administrator since there was no especially convenient way to edit the configuration data stored in registry. Without a special tool at hand to modify the registry *tdb* database directly, the WINREG rpc service was the only way to edit the configuration. Hence the most convenient way to edit Samba's registry configuration data was to use the Windows registry editor `regedit.exe`, see figure 1.

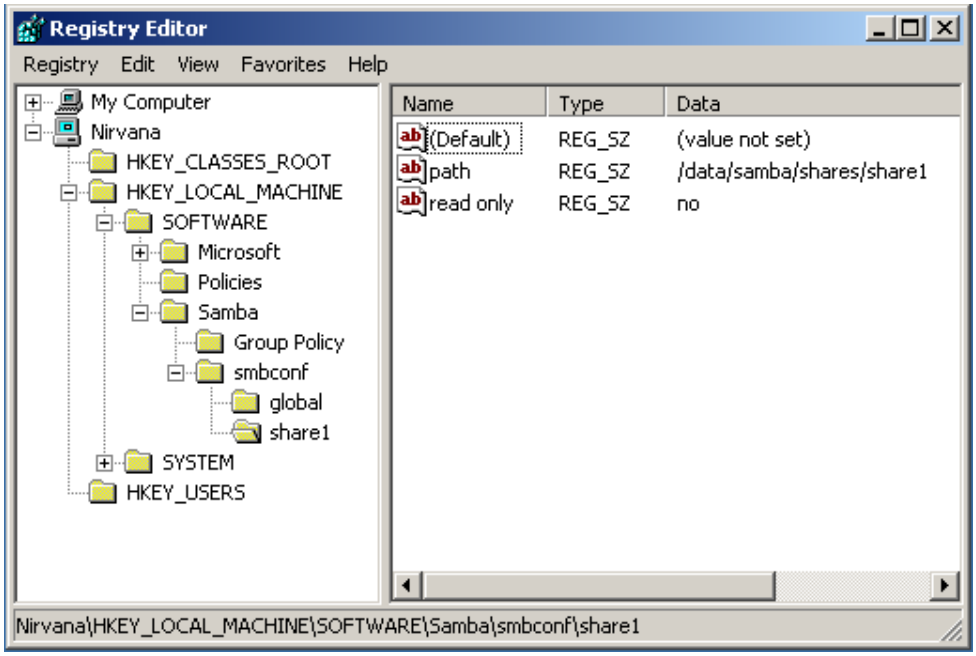

Figure 1: Editing samba configuration with `regedit.exe`

In the meantime, the `net rpc registry` tool of the Samba suite has been extended to be essentially a Unix command line replacement for the remote registry version of `regedit.exe`. And it is now accompanied by `net registry` to access the registry locally, i.e. without using the rpc layer. The functions of the `net registry` command are these:

```
net registry enumerate       Enumerate registry keys and values
net registry createkey       Create a new registry key
net registry deletekey       Delete a registry key
net registry getvalue        Print a registry value
net registry getvalueraw     Print a registry value (raw format)
```

```
net registry setvalue        Set a new registry value
net registry deletevalue     Delete a registry value
net registry getsd           Get security descriptor
```

While it is very convenient that one can thus edit the Samba configuration of a remote server from the Unix command line or even from a windows client without additional effort, it becomes quickly apparent that this interface is still clumsy.

4.3 The `net conf` tool

For a greater level of convenience, the `net` utility has been given a new subcommand `net conf` that provides a dedicated and natural interface to the configuration stored in the smbconf registry key. These are its functions:

4.3.1 Operations on the whole configuration

`net conf list`: Dump the complete configuration in `smb.conf` format to `stdout`.

`net conf listshares`: List only the share names.

`net conf drop`: Delete the complete configuration from registry.

`net conf import`: Import configuration from a file in smb.conf format. This deletes old configuration data first.

4.3.2 Operations on one share

`net conf showshare`: Show the definition of a share, like a portion of an smb.conf file.

`net conf addshare`: Create a new share. This is a high level convenience function that allows to create a service definition with the most common share parameters.

`net conf delshare`: Delete a share definition.

4.3.3 Operations on one parameter

`net conf setparm`: Store a parameter. The service is created if it does not exist.

`net conf getparm`: Retrieve the value of a parameter.

`net conf delparm`: Delete a parameter.

4.3.4 Operations on includes

As detailed above, includes need a special handling in registry configuratoin. Hence there are also special operations for includes. One cannot set, modify or delete single include in the list. Only setting and deleting the whole list of includes is possible.

`net conf getincludes`: Show the includes of a share definition.

`net conf setincludes`: Set includes for a share.

`net conf delincludes`: Delete includes from a share definition.

4.4 The `libsmbconf` API

The functionality of the `net conf` tool has been abstracted into a libray `libsmbconf` located in the `lib/smbconf` folder in the samba source tree to facilitate code reuse internally and even allow for building external applications that access the registry configuration. In creating the `libsmbconf`, another step has been taken, namely the introduction of a backend layer, of which registry is one. Another backend is a simple read only text backend that uses the `pm_process()` parser. The `libsmbconf` API is an abstraction of the parser. It takes a configuration source and returns the parsed result as lists of strings. The basic return structure defined in `lib/smbconf/smbconf.h` is the `smbconf_service`:

```
struct smbconf_service {
        char *name;
        uint32_t num_params;
        char **param_names;
        char **param_values;
};
```

4.4.1 Initialization and shutdown

There are two functions to initialize a smbconf context and to tear it down again and free the allocated memory.

```
WERROR smbconf_init(TALLOC_CTX *mem_ctx,
                    struct smbconf_ctx **conf_ctx,
                    const char *source);

void smbconf_shutdown(struct smbconf_ctx *ctx);
```

Here source is a string of the form `backend:path`, where backend can currently be one of `registry` (or `reg`) and `file`. The init function takes care of initializing the configuration backend operations requested in the source string. This is all stored in the `smbconf_context`.

All other API functions just operate on the context and don't need explicit mention of the backend. The function prototypes of the remaining functions should be self-explanatory, and are only listed without further discussion.

4.4.2 Functions operating on one parameter

```
WERROR smbconf_get_parameter(struct smbconf_ctx *conf_ctx,
                             TALLOC_CTX *mem_ctx,
                             const char *service,
                             const char *param,
                             char **valstr);

WERROR smbconf_set_parameter(struct smbconf_ctx *ctx,
                             const char *service,
                             const char *param,
                             const char *val);

WERROR smbconf_delete_parameter(struct smbconf_ctx *ctx,
                                const char *service,
                                const char *param);
```

4.4.3 Functions operationg on one share

```
WERROR smbconf_get_share(struct smbconf_ctx *ctx,
                         TALLOC_CTX *mem_ctx,
```

```
                            const char *servicename,
                            struct smbconf_service **service);

WERROR smbconf_create_share(struct smbconf_ctx *ctx,
                              const char *servicename);

WERROR smbconf_delete_share(struct smbconf_ctx *ctx,
                              const char *servicename);

bool smbconf_share_exists(struct smbconf_ctx *ctx,
                            const char *servicename);
```

4.4.4 Functions operating on the whole configuration

```
WERROR smbconf_drop(struct smbconf_ctx *ctx);

WERROR smbconf_get_share_names(struct smbconf_ctx *ctx,
                                 TALLOC_CTX *mem_ctx,
                                 uint32_t *num_shares,
                                 char ***share_names);

WERROR smbconf_get_config(struct smbconf_ctx *ctx,
                            TALLOC_CTX *mem_ctx,
                            uint32_t *num_shares,
                            struct smbconf_service ***services);
```

4.4.5 Functions for handling includes

```
WERROR smbconf_get_includes(struct smbconf_ctx *ctx,
                              TALLOC_CTX *mem_ctx,
                              const char *service,
                              uint32_t *num_includes,
                              char ***includes);

WERROR smbconf_set_includes(struct smbconf_ctx *ctx,
                              const char *service,
                              uint32_t num_includes,
                              const char **includes);

WERROR smbconf_delete_includes(struct smbconf_ctx *ctx,
                                 const char *service);
```

4.5 Using the `libsmbconf` interface

To give an impresson of how the libsmbconf interface is used, here is the implementation of the `net conf delshare` function, simplified by ommitting some error checks.

```
static int net_conf_delshare(struct net_context *c,
                             struct smbconf_ctx *conf_ctx,
                             int argc, const char **argv)
{
        int ret = -1;
        const char *sharename = NULL;
        WERROR werr = WERR_OK;
        TALLOC_CTX *mem_ctx = talloc_stackframe();

        sharename = talloc_strdup_lower(mem_ctx, argv[0]);
```

```
        werr = smbconf_delete_share(conf_ctx, sharename);
        if (!W_ERROR_IS_OK(werr)) {
                d_fprintf(stderr, "Error deleting share %s: %s\n",
                          sharename, dos_errstr(werr));
                goto done;
        }

        ret = 0;
done:
        TALLOC_FREE(mem_ctx);
        return ret;
}
```

4.6 Step Two: Globals - loadparm integration

Global parameters are simply stored in the `global` subkey of the smbconf registry key. The problem to solve here is how registry globals should be integrated into `loadparm`. In Samba 3.2, the `smb.conf` file is still the initial source of configuration data, but there are two possibilities to activate global registry configuration:

1. `include = registry` for a mixed text and registry configuration. This option gives the value `registry` for the include directive a special meaning. When the include handler encounters this value, it does not look for a file called `registry` but loads the globals from the smbconf registry key instead. Otherwise the complete semantics of the include directive are preserved, i.e. options from registry override parameters found before `include = registry` in the configuration file and parameters after the include directive will override registry global parameters.

2. `config backend = registry` for a registry only configuration. This works essentially like the `config file` directive in that it discards all data read so far, reads the configuration from the registry globals and does not continue to read the parameters after the `config backend` directive.

The core functionality of both mechanisms, loading and activating the global parameters from registry, is done by the new function `process_registry_globals()`. Here, parsing and activating the parameters is nicely separated. Thanks to the libsmbconf library, the code is very clear. Here is the activation routine, just slightly simplyfied by removing some error handling:

```
bool process_registry_service(struct smbconf_service *service)
{
        uint32_t count;

        if (!do_section(service->name, NULL)) {
                return false;
        }
        for (count = 0; count < service->num_params; count++) {
                do_parameter(service->param_names[count],
                             service->param_values[count],
                             NULL);
        }
        return true;
}
```

And `process_registry_globals()` now is essentially just a call to `smbconf_get_share()` and then `process_registry_service()`:

```
static bool process_registry_globals(void)
{
```

```
        WERROR werr;
        struct smbconf_service *service = NULL;
        TALLOC_CTX *mem_ctx = talloc_stackframe();
        struct smbconf_ctx *conf_ctx = lp_smbconf_ctx();
        bool ret = false;

        do_parameter("registry shares", "yes", NULL);

        if (!smbconf_share_exists(conf_ctx, GLOBAL_NAME)) {
                ret = true;
                goto done;
        }

        werr = smbconf_get_share(conf_ctx, mem_ctx, GLOBAL_NAME,
                                 &service);
        if (!W_ERROR_IS_OK(werr)) {
                goto done;
        }

        ret = process_registry_service(service);

done:
        TALLOC_FREE(mem_ctx);
        return ret;
}
```

In principle, there is nothing special to the registry here. This code would work for any thinkable backend, including the text backend based on the `pm_process` parser. But in Samba 3.2, as a first step, the old parse-and-activate mechanism for the text file configuration has been preserved, and use of the new `libsmbconf` based registry backend code has been intervowen into the central `lp_load()` function, which is now escpecially ugly to read. See the outlook in section 7 for some hints about the current work to get rid of that ugliness.

One thing to note is that according to previous explanations, the `testparm` utility won't show the registry shares without further effort, since they are only read by the `smbd` processes. Therefore, a special mode has been added to the `lp_load()` function to load and activate the whole registry configuration instead of just the globals, but `testparm` is as yet the only user of that otherwise counterproductive mode.

4.7 Summary: Using registry configuration

For the convenience of the potential user, here is a short summary on the three ways in which registry configuration can be activated in Samba 3.2.

1. By specifying `registry shares = yes` in the `[global]` section of `smb.conf`. The samba daemon is set up to load share definitions from registry on demand.

2. A mixed text and registry configuration where global options from text file and registry are used can be triggered by the new special semantics of the include directive `include = registry`.

3. A registry only configuration is effected by specifying `config backend = registry` in the `[global]` section of `smb.conf`. While the `smb.conf` file is still the initial source of configuration, all content other than the `config backend` directive is ignored.

5 Example: `libnetapi` and the `netdomjoin-gui`

Günther Deschner has recently put great efforts into creating a `netapi` library that is supposed to provide the Windows NetApi library (`netapi32.dll`) functionality for the Samba world. Together, we have given the `NetJoinDomain()` function the ability to modify the registry based configuration according to the domain to be joined, so that it is no longer necessary to prepare a suitable `smb.conf` file before doing the domain join operation. The code is no surprise anymore, but there are interesting examples.

As example applications, the Samba source folder `lib/netapi/examples/` contains the commandline tool `netdomjoin` that can be used to join the local Samba server or a remote Samba or Windows machine into a given domain, and the Gtk application `netdomjoin-gui` that is modeled on the Windows dialog `Computer→properties→network identification`. Figures 2–6 present a series of screenshots that were taken during a typical domain join operation, starting with a `smb.conf` file that only contains the global parameter `config backend = registry`. The workflow is precisely that of the Windows gui. There is a special option for modifying the winbind configuration in figure 3 that is still incative. Writing the code to automatically fill the winbind and idmap configuration is an open TODO here.

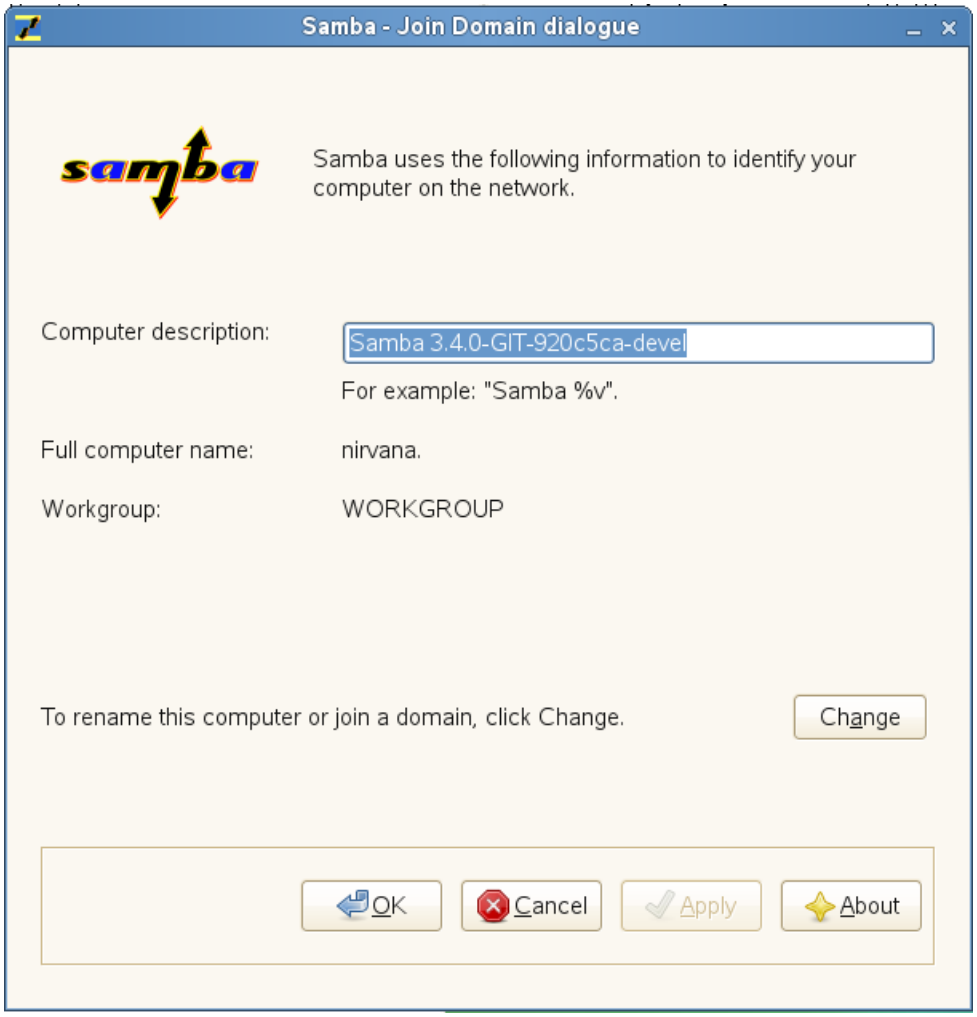

Figure 2: Launching the netdomjoin-gui

After the domain join operation is completed, configuration looks as follows.

```
nirvana:~# net conf list
```

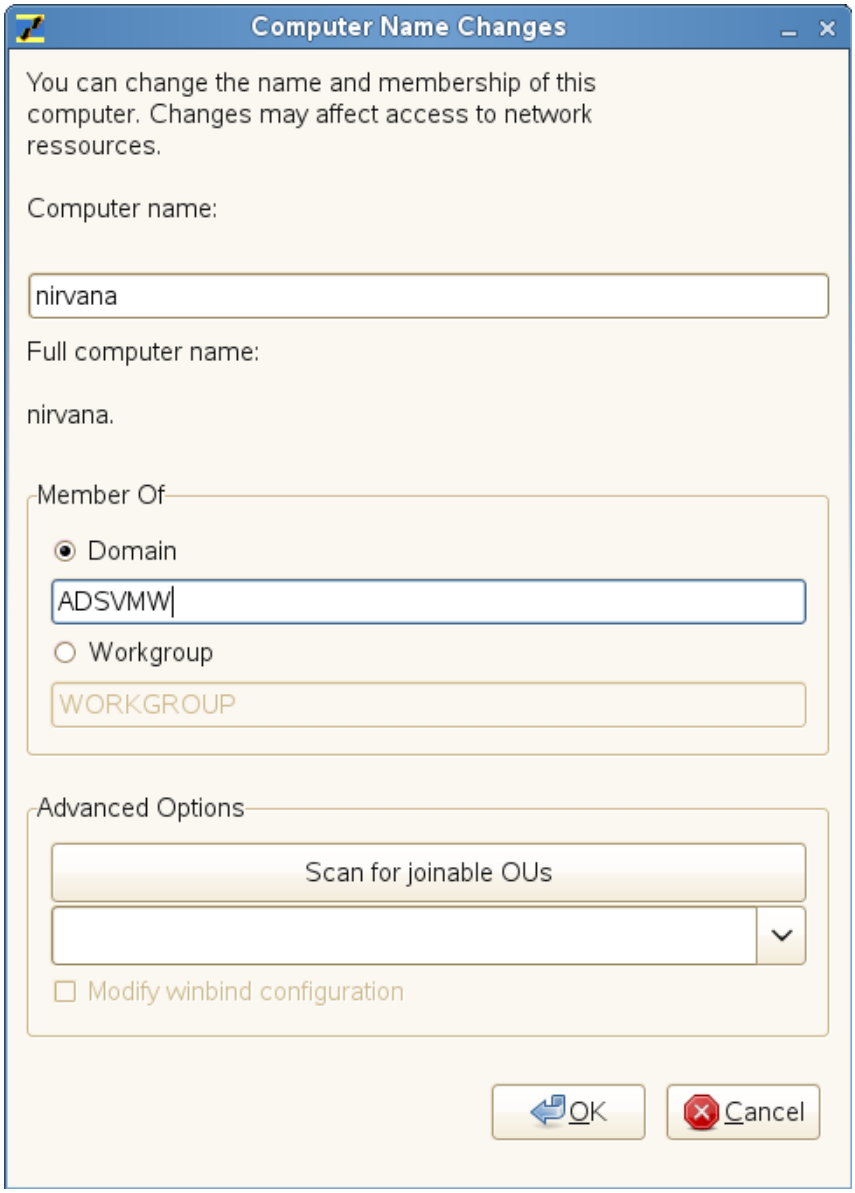

Figure 3: Chosing a new domain

```
[global]
        workgroup = ADSVMW
        security = ads
        realm = ads.vmware.private

nirvana:~#
```

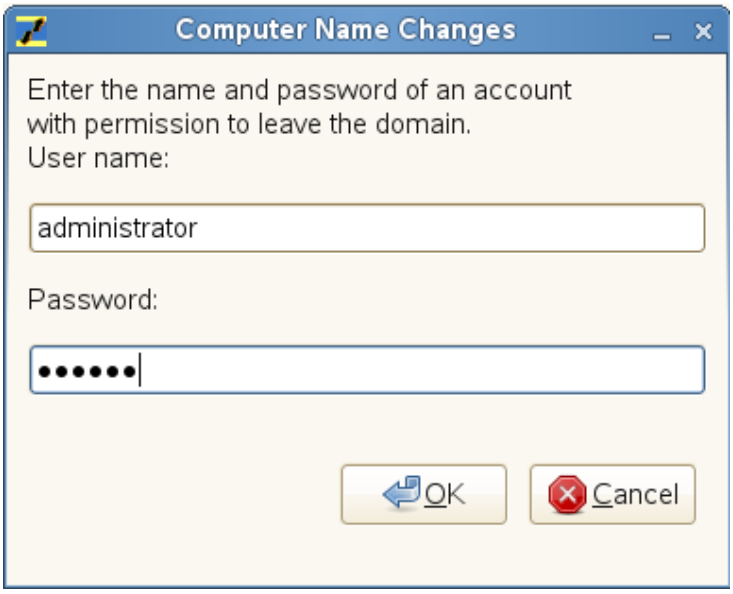

Figure 4: Entering the credentials

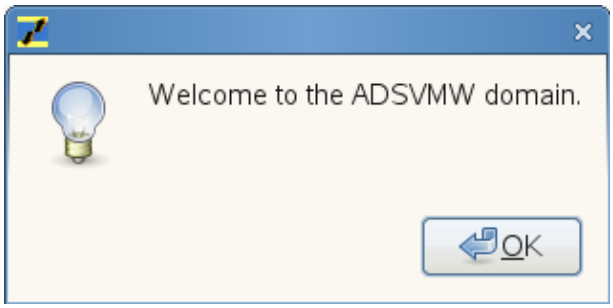

Figure 5: Welcome to the domain

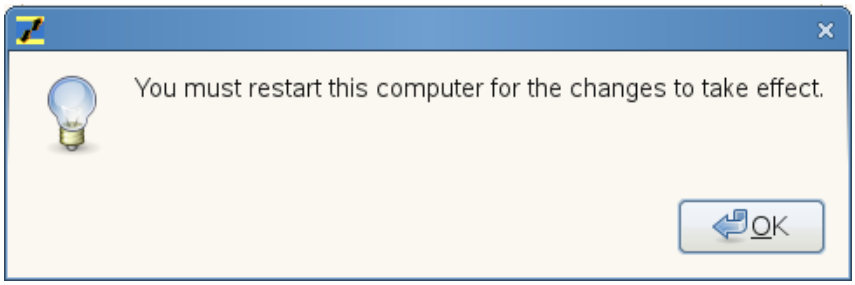

Figure 6: Please reboot... : −)

6 Application: Configuring Clustered Samba

The following intends to give a very brief overview of clustered samba using *ctdb*. This can not do more than convey a basic idea. For more detailed information, see the *ctdb* website [3] and the Samba wiki pages on clustered Samba [5] and ctdb [4].

6.1 The clustering problem

When one wants to deploy Samba in a clustered environment, one major problem is the data coherence and scalabililty of Samba's many internal *tdb* databases used for locking and storing other runtime data. Several persistent *tdb* databases used for storing password and our registry data are also to be taken into account, apart from those transient runtime *tdb*s. Excessive use of fcntl byte range locks to insure data coherence for *tdb*s stored on the cluster file system typically leads to *very* bad performance. In order to circumvent this problem, Andrew Tridgell [8] and others have launched the clustered tdb project *ctdb*.

6.2 ctdb: clustered tdb

The core idea behind *ctdb* is that there is no need to store all data of a *tdb* in the cluster. All nodes have their own local copy of the database, and one of the nodes is the *data master* at a time. This node holds the authoritativ copy of the data and can hence make changes to the database. The important thing is to ensure consistency of the metadata all the time. This is taken care of by the *ctdb* daemon `ctdbd`.

The *ctdb* suite comes with a set of tools, most notably the `ctdb` tool for managing the *ctdb* database and the `onnode` script that can be used to run commands on all or selected nodes of the cluster. Here is the output of the `ctdb status` on a three node cluster:

```
[root@node1 ~]# ctdb status
Number of nodes:3
pnn:0 192.168.42.188   OK (THIS NODE)
pnn:1 192.168.42.189   OK
pnn:2 192.168.42.190   OK
Generation:1391963232
Size:3
hash:0 lmaster:0
hash:1 lmaster:1
hash:2 lmaster:2
Recovery mode:NORMAL (0)
Recovery master:1
[root@node1 ~]#
```

6.3 Samba and ctdb

With the configuration parameter `clustering = yes`, a special mode is enabled in Samba that uses *ctdb* instead of plain *tdb* for many of the internal databases and makes Samba aware of the different cluster nodes it is running on. In order to have this functionality available, Samba must have been built with the configure option `-with-cluster-support`.

As an example, here is the output of a clustered smbstatus command. Note the modified form node:pid instead of a plain pid.

```
[root@node1 ~]# smbstatus -b -d0
Samba version 3.2.3
PID     Username      Group         Machine
-------------------------------------------------------------------
1:11570   W2K3AD\administrator  W2K3AD\domain-users      client1
0:8925    W2K3AD\test           W2K3AD\domain-users      client2
```

6.4 clustered configuration via registry.tdb

When Samba has been set up with cluster support, and the `smb.conf` file activates global registry configuration on *all* nodes, one can conveniently configure all instances of Samba running on all nodes with the `net conf` command on one single node without further effort (or even with `regedit.exe` if that is desired):

```
[root@node1 ~]# onnode 0 net conf getparm global ctdb:test
Error: given parameter 'ctdb:test' is not set.
[root@node1 ~]# onnode 1 net conf setparm global ctdb:test hello,world
[root@node1 ~]# onnode 2 net conf getparm global ctdb:test
hello,world
[root@node1 ~]# onnode 0 net conf delparm global ctdb:test
[root@node1 ~]# onnode 1 net conf getparm global ctdb:test
Error: given parameter 'ctdb:test' is not set.
[root@node1 ~]#
```

This should be sufficient to create an impression of how useful registry configuration is in a clustered Samba setup. This was the original impetus for the development of registry configuration.

7 Outlook: Current and Future work

As mentioned in section 4.6 on the `loadparm` integration of the registry configuration, the use of the `libsmbconf` API inside `loadparm` is not complete in Samba 3.2. Treatment of text files and registry configuration data is completely different.

- The text file is always the initial config source.
- Only the registry processing code is using the new API. This has been patched into the old parse-and-activate code of `lp_load`.
- The detection of changes in the registry configuration data via the tdb sequence number is patched as a special case into the `file_list_changed()` function that maintains a list of timestamps of all text configuration files.
- Shares from the text files are loaded at startup and service structures are immediately allocated for all shares read from text file sources.

Work has been started to make use of `libsmbconf` throughout `loadparm` and eliminate all direct refences to `param/pararms.c`. The plan that has already largely been carried out is to put registry and text file configuration sources on an equal level. Here are some details:

- The hard coded concept of a *config file* is replaced by that of a *config source*. A config source is specified in the form `backend:path`, where backend defaults to file. For the registry backend, the path defaults to the smbconf registry key, but other keys can be specified as well.
- In particular the configuration parameter `config file` is replaced by the more general `config source`.
- Includes take the full config sources, not only config files. This makes the special meaning of `include = registry` of Samba 3.2 appear nicely as a special case of the general case where the backend is registry and the path is the default path (i.e. the smbconf key).
- The macro CONFIGFILE is replaced by the more general CONFIGSOURCE, and the initial config source can be specified on the commandline of the Samba programs by commandline parameter `-configsource` that supersedes `-configfile`.

- A configure option is added to the build process to control the compiled in initial config source.
- In `loadparm`, `process_registry_globals` is replaced by a general purpose function `process_config_source` that is used to rewrite `lp_load` and the include handler to uniformly treat all possible configuration backends.
- In particular, the update decectoin mechanism `file_list_changed` is replaced by a uninform `source_list_changed` that uses the update detection of the libsmbconf API and maintains a `source_list` in which also the smbconf context for the corresponding sources are kept.

Finally, here is slightly simplified source code of the new config loading infrastructure in `loadparm`. Those who have looked more closely at the original design will like the simplicity and clearness of the code.

```
struct source_list {
        struct source_list *next;
        char *name;     /* the verbatim name */
        char *sub_name; /* the name with macros expanded */
        struct smbconf_ctx *ctx;
        struct smbconf_csn csn;
};

static struct source_list *sources_list = NULL;
```

`add_to_source_list` is the function that builds up the list of config sources.

```
struct source_list *add_to_source_list(struct source_list *list,
                                       const char *name,
                                       const char *sub_name)
{
        struct source_list *l, *prev;

        l = "find the entry in the list, or add a new entry"

        l->sub_name = talloc_strdup(l, sub_name);

        smbconf_init(l, &(l->ctx), l->sub_name);

        /* store current change sequence number */
        smbconf_changed(l->ctx, &(l->csn), NULL, NULL);

        return l;
}
```

`process_service` is the function for activating one parsed service.

```
bool lp_source_list_changed(void)
{
        struct source_list *list = sources_list;

        for (; list != NULL; list = list->next) {
                char *sub_name;

                sub_name = "substitute macors in list->name";

                if (smbconf_changed(list->ctx, &list->csn,
```

```
                                        NULL, NULL))
                {
                        return true;
                }

                if ((list->sub_name == NULL) ||
                    (strcmp(sub_name, list->sub_name) != 0))
                {
                        TALLOC_FREE(list->sub_name);
                        list->sub_name = sub_name;
                        return true;
                }
                TALLOC_FREE(sub_name);
        }
        return false;
}
```

The function `process_service` activates one parsed service.
This is the same as `process_registry_service` before.

```
void process_service(struct smbconf_service *service)
{
        uint32_t count;

        do_section(section->name, NULL);
        for (count = 0; count < service->num_params; count++) {
                do_parameter(service->param_names[count],
                             service->param_values[count],
                             NULL);
        }
}
```

The central parsing function is `process_source`. Apart from the fact, that this uses the source list infrastructure, this is basically the same as the original `process_registry_globals`, except that it has a mode for loading the whole configuration instead of globals only.

```
struct source_list *process_source(const char *source_name)
{
        bool ret;
        char *sub_name;
        struct source_list *source;
        TALLOC_CTX *mem_ctx = talloc_stackframe();

        sub_name = "substitute macors in list->name";

        source = add_to_source_list(sources_list, source_name, sub_name);

        if (bGlobalOnly) {
                struct smbconf_service *service;

                if (!smbconf_share_exists(source->ctx, GLOBAL_NAME)) {
                        return source;
                }
                smbconf_get_share(source->ctx, mem_ctx,
                                  GLOBAL_NAME, &service);
                process_service(service);
        } else {
```

```
                uint32_t count, num_shares = 0;
                struct smbconf_service **services;

                smbconf_get_config(source->ctx, mem_ctx,
                                   &num_shares, &services);
                for (count = 0; count < num_shares; count++) {
                        process_service(services[count]);
                }
        }

        return source;
}
```

The core of `lp_load` has now boiled down to a call to `process_source`.

```
bool lp_load(const char *source_name,
             bool global_only,
             bool save_defaults,
             bool add_ipc,
             bool initialize_globals)
{
        bool bRetval = false;
        struct source_list *source = NULL;

        bGlobalOnly = global_only;

        /* some preprocessing */

        source = process_source(source_name);
        smbconf_changed(source->ctx, &(source->csn), NULL, NULL);

        lp_add_auto_services(lp_auto_services());

        /* further setup ... */

        return bRetval;
}
```

The include handler is now extremely simple:

```
bool handle_include(int snum, const char *source_name, char **ptr)
{
        struct source_list *source;

        source = process_source(source_name);

        if (source == NULL) {
                return false;
        }

        string_set(ptr, source->sub_name);

        return true;
}
```

It is important to keep in mind that this rewrite is not merely cosmetical but has a lot of benefits detailed above. These changes should be released with Samba 3.4 which is scheduled for Summer 2009.

References

[1] Samba - the Open Source CIFS server for UNIX. http://www.samba.org/

[2] TDB - Samba's trivial database. http://tdb.samba.org/

[3] CTDB - the clustered tdb project. http://ctdb.samba.org/

[4] Samba Wiki: *CTDB project*: http://wiki.samba.org/index.php/CTDB_Project

[5] Samba Wiki: *Samba & Clustering*: http://wiki.samba.org/index.php/Samba_%26_Clustering

[6] MSDN: *Registry*, http://msdn.microsoft.com/en-us/library/ms724871.aspx

[7] Wikipedia: *Windows Registry*, http://en.wikipedia.org/wiki/Windows_Registry

[8] Andrew Tridge: http://www.samba.org/~tridge/