A new IDMAP subsystem

Problem Statement

The current IDMAP subsystem is plagued by a number of limitations and deficiencies that makes it sub-optimal for a number or widely deployed scenarios.
During the CIFS conference it was decided to create a new subsystem so that these issues could be attacked a resolved.
The most compelling problems regards the lack of an effective caching layer and the necessity to be able to use multiple backends at the same time. We need to be able to use a locally controlled backend for the BUILTIN domain and another one for AD where we don't have the authority to allocate new mappings. As AD and LDAP remote backends could be read-only for a specific installation we need also to change the way IDMAP works and we must not assume it is always possible to create a new mapping. A negative cache will allow us to immediately return if a mapping does not exist and the backend we are using is not allowed to create new mappings. The interface needs also to be extended to allow IDMAP backends to resolve multiple Ids with a single query. This will greatly reduce the roundtrips on slower network protocols (LDAP/AD).

Proposed Solution

The solution modify three aspects of the code: caching, backends, API.

The caching layer will be completely rewritten. The cache will use gencache and will be able to create both positive and negative caching entries. The administrators will be given a way to change the default timeouts for both the positive and negative cache entries separately. The cache will be totally opaque to the API users. Winbindd will be changed to always use the async interface to avoid potential blocking calls, smbd keeps the local in memory cache and this will hopefully make the extra cost of the async calls a non issue.

The backend layer will be completely changed. The administrator will be able to set per Domain backends. The default backend will be used in case a specific backend has not been set up for a given domain. Configuration options will also be per domain with fallback to the current available smb.conf options by default.

The API will change to allow queries to pass an array of ids and get back an array of mappings, this will allow to pack potentially multiple calls to slow backends into a single efficient query.
The new API will also avoid making any checks on UID/GID ranges, these checks will be performed by the single backends (when it make sense) and keeping the configuration consistent will be under the responsibility of the administrator.
For backends that are not authoritative for the mappings a range may act as a filter (IE

mappings that falls out of the mandatory range will not be reported back), in any case a mapping that falls off the range will report a warning.

Maybe: UID/GID ranges will also be merged in a single ID range. Backward compatibility against existing mappings will be guaranteed. Configurations where the UID and GID ranges are not identical will trigger a warning on initialization and where possible the smallest subset of ranges will be used.

No flags are passed down rather IDmap will be the only component be able to decide whether or not to map a sid. The decision will be taken based on all available info. Idmap will be able to make lookupsid calls to assess if a SID is valid and what type of SID it represent.

Public API

```
enum id_type {
     ID_TYPE_UID,
     ID_TYPE_GID
};

struct unixid {
     uint32_t id;
     enum id_type type;
};

struct id_map {
     DOM_SID *sid;
     struct unixid *xid;
};
```

ID mapper:

```
NTSTATUS imdap_unixids_to_sids(    struct id_map **ids);

NTSTATUS imdap_sids_to_unixids(    struct id_map **ids);

NTSTATUS idmap_set_mapping(        const struct id_map *id);

NTSTATUS idmap_remove_mapping(     const struct id_map *id);
```

ID allocator:

```
NTSTATUS idmap_allocate_uid(       struct unixid *id);

NTSTATUS idmap_allocate_gid(       struct unixid *id);
```

Other utility functions (like the current ones in source/sam/idmap_util.c):

```
NTSTATUS idmap_uid_to_sid(    DOM_SID *sid, uid_t uid);

NTSTATUS idmap_gid_to_sid(    DOM_SID *sid, gid_t gid);

NTSTATUS idmap_sid_to_uid(    uid_t *uid, DOM_SID *sid);

NTSTATUS idmap_sid_to_gid(    gid_t *gid, DOM_SID *sid);

NTSTATUS idmap_allocate_uid(  uid_t *uid);

NTSTATUS idmap_allocate_gid(  gid_t *gid);
```

Other possible internal structures:

```
struct idmap_methods {
     NTSTATUS (*unixids_to_sids)(struct id_map **),
     NTSTATUS (*sids_to_unixids)(struct id_map **),
     NTSTATUS (*set_mapping)(const struct id_map *),
     NTSTATUS (*remove_mapping)(const struct id_map *)

};

struct idmap_alloc_methods {
     NTSTATUS (*allocate_id)(struct unixid *),
     NTSTATUS (*set_id_hwm)(struct unixid *),
     NTSTATUS (*get_id_hwm)(struct unixid *),
};

struct id_domain {
     DOM_SID *domain_sid;
     const char *domain_name;
     struct idmap_methods *methods;
};
```

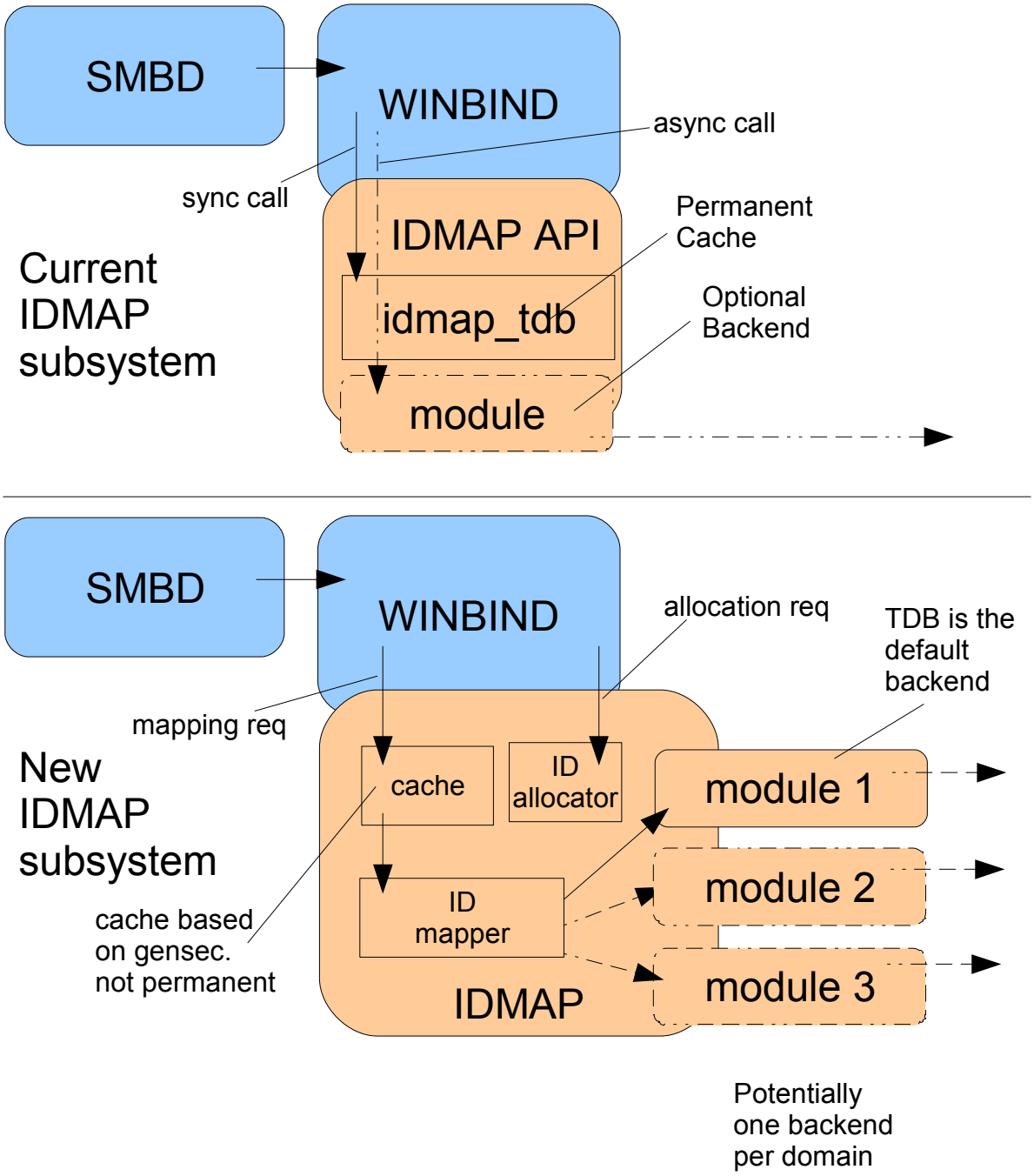

Illustration 1: Overview

IDmap internals

Idmap internally is divided in two separate units, the ID mapper and the ID allocator. The ID mapper may use the ID allocator when a new mapping is requested, but the allocator act as an independent unit.

The following schema shows how IDmap will work internally.

ID allocator interface

The IDmap allocator interface is asked for an allocation [A]. The allocator contacts the designed module [B] to fetch a new ID, and returns to the caller [C] with the answer (an ID or an error)

ID mapper interface

A request for a SID->UID mapping [1] comes in, the cache checks if the answer is readily available. If a cache entry is found a mapping or an error (negative cache) is returned[5].

If we have a cache miss, then [2] the ID mapper is called. the ID mapper determines the SID Domain and checks against the appropriate idmap module [3] to find a mapping. If the mapping is found the cache is filled [4] and the answer is returned to the caller [5].

If the mapping does not exist and the module is read-only a negative entry is set in the cache [4] and an error is returned to the caller [5].

If the module is not read-only then the mapping generator is called.

The mapping generator does a lookupsid call to verify if the SID is valid. If the SID is not valid a negative cache entry is set in the cache [9] and an error is returned to the caller [5].

If the SID is valid then a new ID is requested to the ID Allocator [A] that fetches it form the Allocator Module [B] and returns the result to the mapping generator [C].

If the result is an error a negative cache entry is set into the cache [9] and an error is returned to the caller [5].

If the mapping genrator gets back a valid ID, then it sets the mapping in the appropriate module [8], then the cache is filled with the mapping [9] and the mapping is returned to the caller[5].

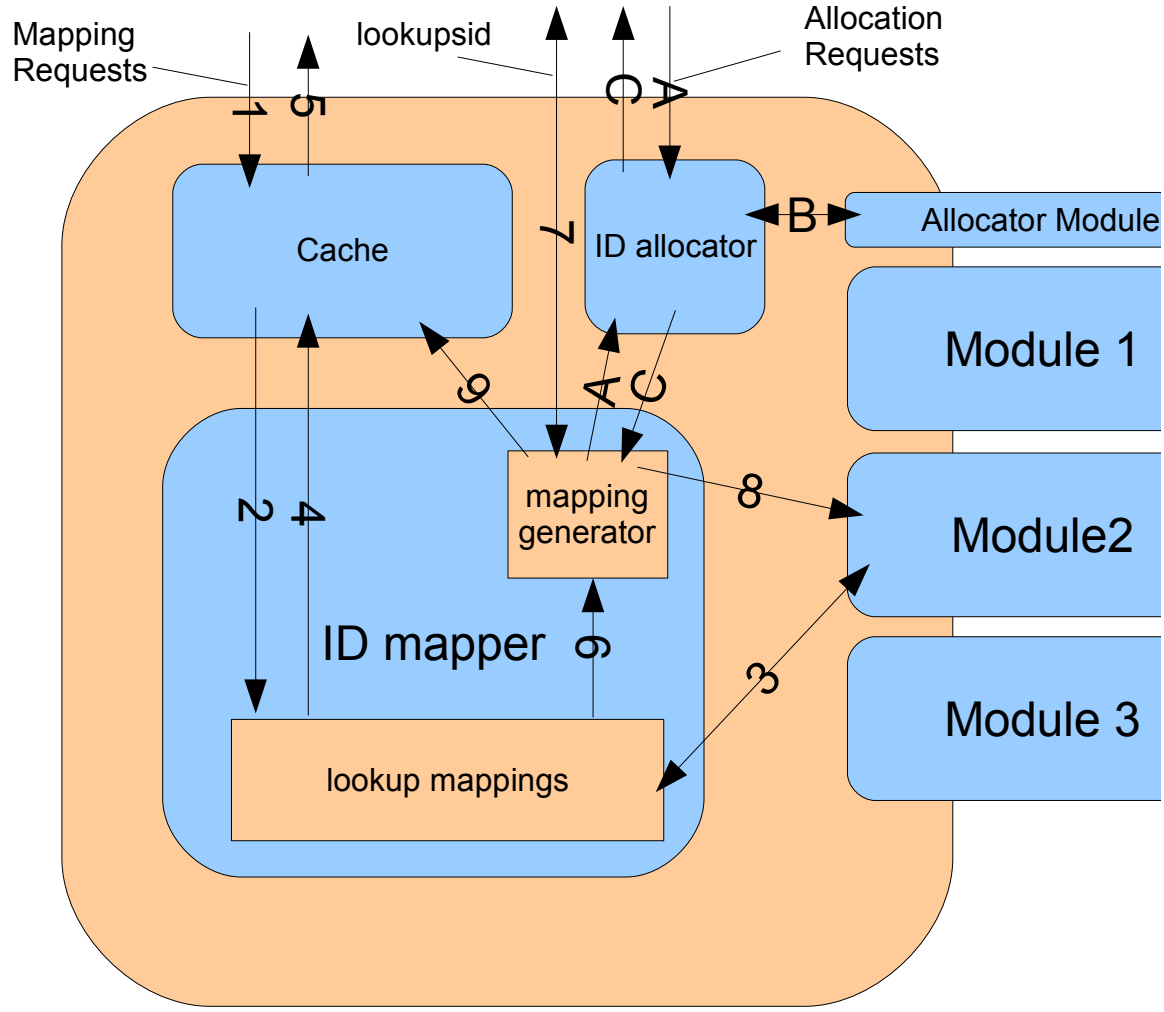

Illustration 2: IDmap internals

Consequences on Winbindd

All the alloc/query_only/cache_only flags will disappear.

Winbindd will probably end up using only 2 calls:

SID array->UNIXID array

UNIXID array->SID array

ID arrays are set in the winbindd request / winbind response extra_data data structure.

A limit on the size of the array must be set and bigger arrays must be split in multiple async calls.

Configuration options

New configuration options are need to be able to configure different domain/module pairs. Parametric options can be used to make it simpler to define options.
Former idmap options will be still valid and used for backward compatibility

Defining idmap domains

The new parameter *idmap domains* will be used to define the list of domains idmap will recognize. *idmap backend* and *idmap domains* will be mutually exclusive.

idmap backend will be ingored if *idmap domains* is present.

If the *idmap domains* parameter is present it will specify a list of domain names (case insensitive), that will be recognized. For each defined domain idmap expects to find a set of parametric options.

If not specified, the default backend is tdb and a default path is used.

The configuration for each domain is specified by using a parametric option in the form *idmap config* <*domain*> where <domain> is the domain name as listed in the *idmap domains* option.

If any range option is set the output from the module will be verified against the range and any mapping outside that range will not be allowed.

The *range* option can be used to specify both uid and gid ranges to be identical, *uid range* and *gid range* can be used to define different numerical ranges for the 2 sets.

Module specific options are in the form of *modname_option* = *value*, where modname is the name of the module. Module specific options are provided to the module on initialization.

If a default domain is not specified mappings for unspecified domains will fail (BUILTIN and LOCAL as well).

Allocator configuration

The option *idmap alloc backend* will define the backend to be used. The configuration of the backend will be provided using the parametric option *idmap alloc config*.

If the allocator configuration is not specified, the same configuration as the default idmap domain will be used.

If no range is defined the allocator will fail any allocation..

The current *idmap uid* and *idmap gid* options will be honored by the allocator.

Alternatively the *idmap alloc config* options named *range*, *uid range* and *gid range* can

also be used. These will take precedence in case they are defined.

Configuration Examples

Example 1:

```
idmap domains = DOMA, DOMB

idmap config DOMA: default = yes
idmap config DOMA: backend = tdb

idmap config DOMB: range = 100001-200000
idmap config DOMB: readonly = yes
idmap config DOMB: backend = ldap
idmap config DOMB: ldap_url = ldap://ldap.server
idmap config DOMB: ldap_anon = yes
idmap config DOMB: ldap_base_dn = OU=idmap,DC=example,DC=com

idmap alloc config: range = 1000-100000
```

Example 2:

```
idmap domains = default, dom.example.com

idmap config default: default = yes

idmap config dom.example.com: range = 100001-200000
idmap config dom.example.com: backend = ad


idmap alloc backend = tdb
idmap uid = 1000-50000
idmap gid = 10000-100000
```

Example 3:

```
;NOTE: the BUILTIN domain will use the tdb backend
;no range checking will be enforced for any domain
;allocation will use a range of 1000-10000

idmap domains = BUILTIN, MYDOM
```

```
idmap config MYDOM: default = yes
idmap config MYDOM: backend = ldap
idmap config MYDOM: ldap_url = ldap://ldap.mydom.com
idmap config MYDOM: ldap_anon = no
idmap config MYDOM: ldap_user_dn = CN=admin,DC=mydom,DC=com
idmap config MYDOM: ldap_base_dn = OU=idmap,DC=mydom,DC=com


idmap uid = 1000-100000
idmap gid = 1000-100000
idmap alloc backend = ldap
idmap alloc config: ldap_url = ldap://ldap.mydom.com
idmap alloc config: ldap_anon = no
idmap alloc config: ldap_user_dn = CN=admin,DC=mydom,DC=com
idmap alloc config: ldap_base_dn = OU=idmap,DC=mydom,DC=com
```

Example 4:

```
;NOTE: allocation will fail as alloc ranges are not defined
idmap domains = DOM

idmap config DOM: range = 100001-200000
idmap config DOM: readonly = yes
idmap config DOM: backend = ldap
idmap config DOM: ldap_url = ldap://ldap.server
idmap config DOM: ldap_anon = yes
idmap config DOM: ldap_base_dn = ou=idmap,dc=mydom,dc=org
```

Example 5:

```
;simplest possible configuration
;all domains on the tdb, allocation also uses tdb

idmap alloc config: range = 1000-100000
```