# High Level Design for Tdinal

Version 1.0 / Jan. 16, 2008

## 1 What's CTDB

CTDB/Samba is a cluster implementation of Samba. It provides failover support to recover from node failures among Samba servers over a cluster file system. Normal Samba uses local TDB (i.e. trivial database) to manage opened files and connections, and other context. CTDB/Samba stores the TDB database in the cluster file system, thus all Samba servers over the cluster file system will share the same TDB database.

TDB library provides a set of routines to add, change or remove records in TDB database. And CTDB will do extra locking via FLOCK to synchronize all accesses to the shared TDB database to keep it consistent. The TDB database stores all the context of any opened inode in a record. The context information might include the connections information (who opens this inode) and inode's opened instances and locks, attributes information. Every record is identified by a magic KEY, defined as structure TDB_DATA. TDB key is a 1:1 mapping to record, the same case for record and inode.

TDB key is defined as:

```
typedef struct TDB_DATA {
    unsigned char *dptr;
    size_t dsize;
} TDB_DATA;
dptr points to a file_id structure and dsize defines the buffer length. file_id is defi
struct file_id {
    uint64_t devid;
    uint64_t inode;
};
```

The inode member is just the inode no of the file. devid is a hash value, to be hashed via several methods depending on Samba configuration specified by user:

| mapping method | Samba hash routine | value to be hashed | no |
|---|---|---|---|
| fsid | device_mapping_fsid | file system id | |
| fsname | device_mapping_fsname | file system name | |
| dev | device_mapping_dev | dev_t struct pointer of the volume device | def |

Lustre is configured to use fsname with CTDB to identify the cluster, since fsname is identical among all clients and unique to this cluster, like MDS:/client for Lustre. fsid method couldn't work currently since llite always returns a zero value in ll_statfs. So devid could correctly represent the just cluster and inode represents the file, then the combined file_id structure identifies the identical inode on different Lustre clients. When one node fails, CTDB could replicate the same context of the dead node on another client with TDB records.

# 2 Sharing Violation Issue with CTDB

When CTDB/Samba treats a file on different Lustre clients as an identification, it brings an issue to pCIFS unfortunately: pCIFS will get failure of STATUS_SHARING_VIOLATION when trying to open inode on OST. Before redirecting io from MDS to OST servers, pCIFS need open the inode instance on OST nodes. At that time the inode is already opened on Samba server of MDS and the opened instance will be kept until i/o complets and will finally be released by user at any undetermined time. If the original instance is opened exclusively on MDS, Samba server on OST will deny the second open request. This case always happen when writing to non-exist files (new files will be created), thus pCIFS often fails to open and then nothing of parallel i/o could be made.

Besides share mode conflicts, windows opportunity locks also bring issue and increase the complexity. Normally the first open operation on MDS will be granted an exclusive oplock, then when the second open operation arrives at OST servers, the Samba server on OST will issue an oplock break request to revoke the granted oplock. The break request will be sent to CIFS client, pCIFS node in this case, to perform cache flush, then pCIFS node acks a notification to Samba server and in turn Samba server gets acknowledged and perform the second open. Obvious it induces extra network transfer and also a delay in processing the second open until the original oplock is borken (break notification acked) or it timeouts in around 30 seconds. When file open is to be processed many times, it influences i/o performance badly.

# 3 File-Open Processing Flow

## 3.1 Window Open API

Like Linux api "open" has mode and flags in parameter list to indict how to perform an open operation, Windows api CreateFile carries similar parameters

to indict the intent operation (read, write or delete) and share mode attributes:

```
Prototype:
HANDLE
CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
Parameters explaination:
dwDesiredAccess defines the indent access that we want to do with the file, normally it
#define FILE_READ_DATA ( 0x0001 )      // read data (i/o)
#define FILE_WRITE_DATA ( 0x0002 )     // write data (i/o)
#define FILE_APPEND_DATA ( 0x0004 )    // write file in the tail (i/o)
#define FILE_READ_EA ( 0x0008 )        // EA read
#define FILE_WRITE_EA ( 0x0010 )       // EA write
#define FILE_EXECUTE ( 0x0020 )        // read and execute
#define FILE_TRAVERSE ( 0x0020 )       // directory traverse
#define FILE_ADD_FILE ( 0x0002 )       // add an entry into directory
#define FILE_ADD_SUBDIRECTORY (0x0004) // add subdir entry
#define FILE_DELETE_CHILD ( 0x0040 )   // directory
#define FILE_LIST_DIRECTORY ( 0x0001 ) // list directory
#define FILE_READ_ATTRIBUTES ( 0x0080 ) // attribute get
#define FILE_WRITE_ATTRIBUTES ( 0x0100 ) // attribute set
#define FILE_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | SYNCHRONIZE | 0x1FF)
#define FILE_GENERIC_READ (STANDARD_RIGHTS_READ |\
                           FILE_READ_DATA |\
                           FILE_READ_ATTRIBUTES |\
                           FILE_READ_EA |\
                           SYNCHRONIZE)
#define FILE_GENERIC_WRITE (STANDARD_RIGHTS_WRITE |\
                           FILE_WRITE_DATA |\
                           FILE_WRITE_ATTRIBUTES |\
                           FILE_WRITE_EA |\
                           FILE_APPEND_DATA |\
                           SYNCHRONIZE)
#define FILE_GENERIC_EXECUTE (STANDARD_RIGHTS_EXECUTE |\
                           FILE_READ_ATTRIBUTES |\
                           FILE_EXECUTE |\
                           SYNCHRONIZE)
dwShareMode defines whether this operation could be shared with other accesses or not.
The share mode value could be any one or any combination of the following values:
```

```
FILE_SHARE_DELETE: make all possible delete operations sharable
FILE_SHARE_READ: share all reading operations
FILE_SHARE_WRITE: enables subsequent open operations to request writing.
lpSecurityAttributes: contain a SECURITY_ATTRIBUTES structure that determines whether c
dwCreationDisposition defines the intent action to be taken on this file. It could be a
CREATE_ALWAYS Creates a new file, always. If a file exists, the function overwrites the
CREATE_NEW Creates a new file. The function fails if a specified file exists.
OPEN_ALWAYS Opens a file, always. If a file does not exist, the function creates a file
OPEN_EXISTING Opens a file. The function fails if the file does not exist.
TRUNCATE_EXISTING Opens a file and truncates it so that its size is 0 (zero) bytes. The
The calling process must open the file with the GENERIC_WRITE access right.
dwFlagsAndAttributes : file attributes and flags.
FILE_ATTRIBUTE_HIDDEN: hidden attribute
FILE_ATTRIBUTE_NORMAL: normal, no special attributes set
FILE_ATTRIBUTE_READONLY: readonly
FILE_ATTRIBUTE_SYSTEM: system
FILE_FLAG_DELETE_ON_CLOSE: system will deletes the file immediately after all of its ha
FILE_FLAG_NO_BUFFERING: similar to Linux direct i/o
FILE_FLAG_OVERLAPPED: asynchronous i/o
FILE_FLAG_POSIX_SEMANTICS: to be accessed according to POSIX rules.
FILE_FLAG_WRITE_THROUGH: The system writes through any intermediate cache and goes dire
```

### 3.2 Samba Open File

CIFS defines 4 different methods to create/open a file or directory, thus Samba implements 4 different callback routines to response these 4 requests:

| CIFS / SMB request | value of the SMB comm | Samba callback routine |
|---|---|---|
| SMB_COM_OPEN | 0x02 | reply_open |
| SMB_COM_CREATE | 0x03 | reply_mknew |
| SMB_COM_OPEN_ANDX | 0x2d | reply_open_and_X |
| SMB_COM_NT_CREATE_ANDX | 0xa2 | reply_ntcreate_and_X |

The function logics to process these 4 requests are similar and finally it will call open_file_ntcreate to perform the real file creation or opening. The 4th request is mostly used. Here we use replay_open_and_X as an example to explain the creation/opening process logic:

```
reply_ntcreate_and_X()
{
    1, map_open_params_to_ntcreate()
    2, open_file_ntcreate()
    3, construct replay package (oplock information)
       just grant oplock if option "fake oplocks" is set.
}
open_file_ntcreate()
{
```

```
    1, clear all oplock flags if oplocks support is disabled.
    2, check if the file name is valid or not (containing invalid chars )
    3, convert Windows open mode/flags to Linux
    4, allocate necessary Sambe internal file data structure
    5, if the file to be opened exists (stat reports to us), it will do
       the followings:
    5.1, call get_share_mode tograb TDB record FLOCK and query the access
         list from TDB database
    5.2  call delay_for_oplcoks to check whether there is oplock
         conflict or not. If there is, then issue a break request and
         defer the open operation to wait oplock util break completes
    5.3  call open_mode_check to check whether all the opened access
         lists are compatible with this open operation. It it conflicts,
         return STATUS_SHARING_VIOLATION
    6, call open_file to do the real creation or open operations by VFS
    7, if this file is just created in previous step, then do:
    7.1, call get_share_mode() to allocate record in TDB database and
         grab the FLOCK  in TDB database file.
    7.2, call open_mode_check to avoid possible race condition between
         other smbd processes. It will defer this open request to next
         round if race condtions happen.
    8, call set_share_mode to add this instance into access list. The new
       share mode will be synced to TDB database.
    9, then fill structures and release necessary resources (TDB FLOCK,
       memory, structures), then return
}
```

### 3.3 Sharing Mode Checking Rules

In subsection 3.1 we get that among all the share mode access flags, only FILE_WRITE_DATA, FILE_APPEND_DATA, FILE_READ_DATA, FILE_EXECUTE, DELETE_ACCESS could bring conflicts. So we only concern these 5 flags. And then there form 8 general rules to detect share mode conflictions. Samba realizes these rules in function share_conflict:

```
static
BOOL
share_conflict(
    struct share_mode_entry *entry,
    uint32 desired_access,
    uint32 desired_share);
```

| No | Share Mode Check (conditions of no conflicts) | Not |
|---|---|---|
| 1 | entry doesn't contain any above flags | |
| 2 | desired_access doesn't contains any above | |
| 3 | if entry contains writing flags, FILE_SHARE_WRITE must be be set for both | |
| 4 | if desired_access contains writing flags, both should be shared in writing | |
| 5 | if entry contains reading flags, FILE_SHARE_READ must be be set for both | |
| 6 | if desired_access contains reading flags, both should be shared in reading | |
| 7 | if entry contains deleting flags, FILE_SHARE_DELETE must be be set for both | |
| 8 | if desired_access contains deleting flags, both should be shared in deleting | |

### 3.4 Samba Configuration Option

Samba already implements an option named "Share Modes" to define the behavior of share_conflicts. If it's defined as "yes" in smb.conf, share_conficts just return OK for all with all the above 8 rules skipped. So we needn't any patch to let Samba always grant any opening operations.

## 4 Opportunity Locks

Windows uses opportunity locks to keep the cache consistency between the Lanman redirector (CIFS client) and the file server (server side). The client can keep file data in cache only when it's granted the oplocks.

### 4.1 Oplocks Types

There are three main types of oplock:

1. Level I oplock is granted when a client has exclusive access to a file. A client holding this type of oplock for a file can cache both reads and writes on the client system.

2. Level II oplock represents a shared file lock. Clients that hold a Level II oplock can cache reads, but writing to the file invalidates the Level II oplock.

3. Batch oplock is the most permissive kind of oplock. A client with this oplock can cache both reads and writes to the file as well as open and close the file without requesting additional oplocks. Batch oplocks are typically used only to support the execution of batch files, which can open and close a file repeatedly as they execute.

There's also another type: Filter oplock, which only interferes with file attributes. It seems that CIFS protocol does nothing with it and so Samba needn't do anything about it. At the moment it's safe for us to ignore it. Some research will be done later on it.

## 4.2 Oplock Request and Grant

Oplocks are requested in combination with the open/create operation and CIFS server will set the grant flag if oplock is granted or just leave it as 0 no oplock is granted. Here's a typical package traffic of a file open process:

```
Request from Client to Server:
Create AndX Request, FID: 0x2bc3, Path: \2Ma.dat
NT Create AndX Request (0xa2)
FID: 0x2bc1 (\2Ma.dat)
Create Flags: 0x00000016
.... .... .... ...1 .... = Extended Response: Extended responses required
.... .... .... .... 0... = Create Directory: Target of open can be a file
.... .... .... .... .1.. = Batch Oplock: Requesting BATCH OPLOCK
.... .... .... .... .... ..1. = Exclusive Oplock: Requesting OPLOCK
Response from Server to Client:
Response: SMB NT Create AndX Response, FID: 0x2bc3
NT Create AndX Response (0xa2)
Oplock level: Batch oplock granted (2)
FID: 0x2bc1 (\2Ma.dat)
Create action: The file did not exist but was created (2)
... ...
```

## 4.3 Oplock Break

Samba calls delay_for_oplock to issue an oplock break request, as described in section 3.2. Here's a general oplock break process:

## 4.4 Samba options on oplocks

Samba has bunch of oplock options and some of them are vague and hard to understand. That was also the reason why I had to make a patch to let smbd ignore oplocks to make pCIFS work with CTDB.

Here are the oplock configuration options of Samba:

1. oplocks: support oplocks, or don;t grant any oplocks when it's FALSE
2. fake oplocks: always let any oplocks granted
3. kernel oplocks: enable oplocks with linux kernel support, client access need oplock granted
4. level2 oplocks: always try to break oplocks to level II if non target oplock (or NONE) is not specified
5. oplock break wait time: timeout value to assume that oplock is broken

6. oplock contention limit: this parameter limits the response of the Samba server to grant an oplock if the configured number of contending clients reaches the limit specified by the parameter. Samba recommends "DO NOT CHANGE THIS PARAMETER UNLESS YOU HAVE READ AND UNDERSTOOD THE SAMBA OPLOCK CODE." Oplock Break Contention Limit can be enable on a per-share basis, or globally for the entire server, in the smb.conf file.

7. veto oplock files: Veto oplocks is a smb.conf parameter that identifies specific files for which Oplocks are disabled. When a Windows client opens a file that has been configured for veto oplocks, the client will not be granted the oplock, and all operations will be executed on the original file on disk instead of a client-cached file copy. By explicitly identifying files that are shared with UNIX processes, and disabling Oplocks for those files, the server-wide Oplock configuration can be enabled to allow Windows clients to utilize the performance benefit of 8 file caching without the risk of data corruption.

Among all these 7 options, we only need take count of "oplcoks". With "oplcoks" defined as "no", no any oplocks are granted and all CIFS clients won't cache the file data, that just what we want on OST nodes. If user will try to modify data via Lustre Linux native client, we'd better turn "oplocks" off on all nodes.

# 5 Conclusion

From the above analysis, we can conclude that we needn't any further work or patch any more. Samba options "Share Modes" and "oplocks" are enough for us to solve the SHARING_VIOLATION issue.

For the MDS nodes, we can turn "oplocks" on, thus files shared by Samba on MDS node can benefit from cache. Actually it's data is read from / written to OST nodes. The "Share Modes" option is better to turn off, or open requests to MDS node might be denied since the share modes of opened instances on OST nodes are incompatible. For OST nodes, we'd better turn both options off. Lustre could do the rest of jobs to keep cache consistence.

A comparison test is better to scheduled to check the performance differences on MDS server between with "oplocks" option on and with it off.

# 6 References


1. Windows WINDDK 6000

2. Windows Internals 4e

3. Samba and CTDB sources