

Implementing Persistent Handles in Samba

Ralph Böhme, SerNet, Samba Team

2019-01-30

Contents

1	Introduction	2
1.1	Persistent Handles in Samba, Design and Prototype	2
1.2	Current Status of Transparent Failover Features in Samba	3
1.3	System Requirements	3
1.4	Restrictions	4
1.5	Scope	4
2	Design	4
2.1	Overview	4
2.2	Data Model	5
2.3	Persistent Handles Cleanup	8
3	Future Improvements	9
3.1	Interop with Other File-Sharing Protocols or Local Access	9
4	MS Protocol Spec Notes	10
4.1	SMB layer	10
4.2	FSA layer	12
5	Development Plan	12
5.1	Research	12
5.2	Implementation	12
5.3	Clustered Samba CI	14
5.4	CI Coverage of Persistent Handles	14
5.5	End to End Tests	15
5.6	Related Changes	15
5.7	Summary	15
6	Appendix	15
6.1	Revision Info	15

1 Introduction

1.1 Persistent Handles in Samba, Design and Prototype

This document describes the design of the Samba Persistent Handles prototype by Ralph Böhme from SerNet and the Samba Team.

Persistent Handles are part of a larger Windows Server feature called **SMB Transparent Failover** which consists of the following related technologies:

- Clustered SMB File Service
- Persistent Handles
- Continuously Available Shares (a share with Persistent Handles enabled)
- SMB Multi Channel
- Client Application Failover (Application Instance ID)
- Witness RPC Service
- Replay Detection

When Continuous Availability (CA) is enabled on a share, the SMB3 server stores specific file metadata associated with the Persistent Handles on disk. When an SMB cluster node failover occurs, the other cluster nodes can read the metadata from disk.

A Persistent Handle requires the metadata of the file be committed to the backend storage before a request is completed, which includes:

- file data
- file metadata
- name-space change (eg delete-on-close)

For any event that causes the client to loose connectivity to the cluster, the cluster will preserve the state of the "disconnected" Persistent Handles. Any new open conflicting with state of the disconnected Persistent Handle will be rejected or suspended.

Enabling Continuous Availability will make all I/O on Persistent Handles to be performed in synchronous or write-through mode. Performance may be affected and the impact will be similar to enabling the uncached mount option on a file system

Continuous Availability is designed for applications, such as Microsoft Hyper-V or SQL Server, where the number of open/close operations is limited and that implement their own application layer caching. It is not recommended to use CA in scenarios like Home Directory workloads.

Code of the prototype can be found on [git](#). Presentations from SDC 2018: [slides](#), [recording](#).

1.2 Current Status of Transparent Failover Features in Samba

1.2.1 Clustered SMB File Service

Samba already supports clustered flesharing with ctdb and cluster filesystems.

1.2.2 Persistent Handles

Not supported in Samba.

1.2.3 Continuously Available Shares

Not supported, requires Persistent Handles.

1.2.4 SMB Multi Channel

Experimental support available. Known issues: Samba bugs [11897](#) and [11898](#).

1.2.5 Client Application Failover

Not supported

1.2.6 Witness RPC Service

Not supported, existing prototype, requires considerable improvements in the RPC server code used by the Samba fileserver.

1.2.7 Replay detection

Create, read and write replay detection supported, lock replay not supported yet, patches pending for inclusion in Samba.

1.3 System Requirements

The prototype has no additional requirements beyond Clustered Samba:

- POSIX compatible operating system

- use of ctdb and "clustering = yes" in smb.conf
- Clustered file system

Basically any system that is capable of running clustered Samba will also support Persistent Handles.

1.4 Restrictions

Persistent Handle state is maintained in Samba's internal databases, therefore there's no interop with other protocols like NFS or local file system access. See the section [Future Improvements](#) for additional information.

1.5 Scope

Transparent Failover features that will be implemented:

- Persistent Handles
- Resilient Handles
- Continuously Available Shares (a share with Persistent Handles enabled)
- Replay Detection

The following features will not be implemented or completed:

- SMB Multi Channel
- Client Application Failover (Application Instance ID)
- Witness RPC Service

2 Design

2.1 Overview

To implement Persistent Handles in Samba a new dbwrap storage backend is implemented with support for persistent storage and fast retrieval of file-handle state.

dbwrap is the core Samba abstraction for database access. It's an API with support for different storage backends.

By combining two existing dbwrap backends, one with volatile semantics, which is fast, and the other with persistent semantics, which is slow, it is possible to meet the requirements of Persistent Handles. See the section [Database Layer](#) for additional details.

Persistent Handles support will impose the same restrictions as the Durable Handle implementation namely SMB only access to files.

Support for Persistent Handles will be a per share configuration option.

2.2 Data Model

2.2.1 Database Layer

Implement a new dbwrap backend with the following properties:

- persistency can be requested on a per record basis
- fetching records must be fast
- storing persistent records can be slow
- records marked persistent survive a full cluster crash and reboot

The new dbwrap backend combines two existing backends: a volatile and a persistent one and the dbwrap API adds a new flag `DBWRAP_PERSISTENT` to `dbwrap_record_store()`.

Persistency can be requested on a per record basis by passing the new flag to `dbwrap_record_store()`. This provides the core building block for the higher level feature.

The following Samba database store the relevant Persistent Handle state:

- SMB2 layer filehandle state in `smbXsrv_open_global.tdb`
- FSA state: `locking.tdb`, `leases_db.tdb`, `brlock.tdb`

`smbXsrv_open_global.tdb` and `locking.tdb` will be converted to the new per-record persistent dbwrap backend.

`brlock.tdb` will likely be merged with `locking.tdb`.

State from `leases_db.tdb` would be needed to support Windows Terminal Server access. Alternatively, we could prevent enabling Persistent Handles on Samba shares that use variables in the `path` option.

Samba currently implements SMB2 Create Replay by storing file handle state with Create GUID as record key in a process private in-memory database. This means that Create Replay is not functional, as any session reconnect will result in a new Samba process that can't access

the private in-memory database of the process previously servicing the session. To address this, we need either an additional database or store the records in the `smbXsrv_open_global.tdb` database.

2.2.2 Global

- SMB3 capability: `SMB2_GLOBAL_CAP_PERSISTENT_HANDLES`
- Samba option: "persistent handles = yes | no"

2.2.3 Per Share

Capabilities:

- `SMB2_SHARE_CAP_CONTINUOUS_AVAILABILITY`
- `SMB2_SHARE_CAP_SCALEOUT`
- `SMB2_SHARE_CAP_CLUSTER`.

`Share.CA_Timeout` initialized to 0. Possible add a Samba option to make it configurable. (MS-SMB2 3.3.1.5)

```
struct smbXsrv_tcon_global0 {
    ...
    uint32_t capabilities;
    uint32_t ca_timeout;
}
```

2.2.4 Per Open

According to MS-SMB2 3.3.1.10 the following are needed in SMB 3.x:

- `Open.IsPersistent`
- `Open.FileName`
- `Open.DesiredAccess`
- `Open.ShareMode`
- `Open.CreateOptions`
- `Open.FileAttributes`
- `Open.CreateDisposition`

Which of those must be persisted on-disk in `smbXsrv_open_global.tdb`?

`Open.IsPersistent` is only strictly needed if the Persistent Handles cleanup implementation only scans `smbXsrv_open_global.tdb` and not `locking.tdb`. Otherwise we could probably just add it to struct `files_struct`.

`Open.FileName`, `Open.DesiredAccess`, `Open.ShareMode` can be taken from the FSA layer as they're already stored there in `locking.tdb`.

`Open.CreateOptions` is needed as it's used for `SVHDX_OPEN_DEVICE_CONTEXT`, `SMB2_WRITEFLAG_WRITE_THROUGH` and `SMB2_WRITEFLAG_WRITE_UNBUFFERED`.

To implement correct delete-on-close semantics we need the `initial_delete_on_close` and `delete_on_close` file handle flags stored in the Persistent Handle record. That means we probably don't need `Open.CreateDisposition`.

`Open.FileAttributes` is likely not needed, as any access to files that might change the flags while a Persistent Handle is disconnected will be blocked anyway.

We have to record persistent handle expiration timer as absolute `NTTIME` or similar. When a concurrent opener tries to open a file that has an associated Persistent Handle in disconnected state, it will set this timer. This ensures we don't preserve Persistent Handle for too long in case of a node crash running the cleanup master.

```
struct smbXsrv_open_global0 {
    ...

    bool is_persistent;
    uint64_t ph_gen_id;
    uint32_t create_options;
    uint64_t initial_allocation_size;
    NTTIME ph_expiration;
    bool delete_on_close;
    bool initial_delete_on_close;
};
```

2.2.5 FSA Layer, per File

For the [disconnected persistent handle](#) check at the FSA layer, we must be able to determine if there's an associated persistent handle without handle-lease or batch-oplock. To implement this, we need a new flag `SHARE_MODE_FLAG_PERSISTENT_OPEN` in struct `share_mode_entry.flags`.

2.3 Persistent Handles Cleanup

2.3.1 Cleanup Design

With file handle state stored in persistent database, it becomes vitally important to implement correct cleanup for the following failure types:

- transport loss
- graceful process termination
- graceful node shutdown
- graceful cluster restart
- involuntary process termination
- involuntary node crash
- involuntary cluster crash followed by restart

cleanupd's of all nodes in a cluster elect a single cleanup master by trying to acquire a global lock with `g_lock_lock_send()`. The winner, called the cleanup master, listens for `CTDB_SRVID_SAMBA_NOTIFY` messages.

Certain failure types require that the master cleanupd process traverses the relevant databases persistent records. This will be traverses on node local persistent databases which do take a global glock, so other writers will not be able to make progress, but the traverse will be fast as it's a local traverse.

By traversing the persistent `locking.tdb`, cleanupd would be able to scavenge both `locking.tdb` and `smbXsrv_open_global.tdb`, as the `locking.tdb` records contain the key (the Persistent File ID) of the `smbXsrv_open_global.tdb` database.

smbd processes track the number of Persistent Handles. If this number increased from 0 to 1 they call `ctdb_watch_us()` to request to be monitored by ctdb. If the number decreases from 1 to 0 they unregister with `ctdb_unwatch_us()`. In case a monitored process crashes, ctdb will send a `CTDB_SRVID_SAMBA_NOTIFY` to the cleanup master.

After a cleanup master election, or when receiving a `CTDB_SRVID_SAMBA_NOTIFY` message, the elected cleanup master traverses the `smbXsrv_open_global.tdb` and notifies the scavenger by calling `scavenger_schedule_disconnected()`.

cleanupd should delay actions based on receiving `CTDB_SRVID_SAMBA_NOTIFY` messages to allow folding of multiple quickly succeeding messages into one.

2.3.2 Failure Types

1. transport loss, graceful process

Process directly notifies scavenger. Basically the same behaviour as for Durable Handles.

2. graceful node

If the node was not the cleanup master, same as "transport loss". If the node was the cleanup master, another node will become the cleanup master and traverse the persistent `smbXsrv_open_global.tdb`.

3. graceful cluster, involuntary cluster

On startup, after the cleanup master election, the cleanup master traverses persistent `smbXsrv_open_global.tdb`.

4. involuntary process, involuntary node

The node with the cleanup master receives `CTDB_SRVID_SAMBA_NOTIFY`.

2.3.3 Persistent Handles Generation ID

From an administrative perspective it may be desirable in certain situations, to be able to force expiration of Persistent Handles.

One way to achieve this would be using an additional cluster global persistent generation ID. The ID would be stored alongside the handle state. When a client attempts to reconnect a Persistent Handle, the server compares the value of the current generation ID with the value from the handle. Only if they don't match the reconnect would be rejected.

The method to increase the cluster-wide generation ID would be a user command like a net subcommand or `smbcontrol`.

Alternatively we may just do a traverse on the persistent `locking.tdb` and cleanup the records.

3 Future Improvements

3.1 Interop with Other File-Sharing Protocols or Local Access

The core functions that store and retrieve Persistent Handle state are the existing VFS functions for Durable Handles. Samba VFS modules can therefor implement the VFS function `SMB_VFS_DURABLE_COOKIE()` in order to pass Persistent Handle state to the `vxfs` file system.

As back-channel from the file system a notification mechanism is required, preferably with a pollable file descriptor.

4 MS Protocol Spec Notes

4.1 SMB layer

4.1.1 Flag checks in DH2C (MS-SMB2 3.3.5.9.12)

MS-SMB2 3.3.5.9.12:

If `Open.IsPersistent` is `TRUE` and the `SMB2_DHANDLE_FLAG_PERSISTENT` bit is not set in the `Flags` field of the `SMB2_CREATE_DURABLE_HANDLE_RECONNECT_V2` Create Context, the server SHOULD<286> fail the request with `STATUS_OBJECT_NAME_NOT_FOUND`.

If `Open.IsPersistent` is `FALSE` and the `SMB2_DHANDLE_FLAG_PERSISTENT` bit is set in the `Flags` field of the `SMB2_CREATE_DURABLE_HANDLE_RECONNECT_V2` Create Context, the server SHOULD<288> fail the request with `STATUS_INVALID_PARAMETER`.

4.1.2 Ignoring invalid DH2Q requests

The server MUST ignore this create context and skip this section if:

- the `SMB2_DHANDLE_FLAG_PERSISTENT` bit is set in the `Flags` field of this create context and `TreeConnect.Share.IsCA` is `FALSE`, or
- the `SMB2_DHANDLE_FLAG_PERSISTENT` bit is not set in the `Flags` field of this createcontext

4.1.3 `Open.DurableTimeout` for Persistent Handles (MS-SMB2 3.3.5.9.10)

- If the `Timeout` value in the request is not zero, the `Timeout` value in the response SHOULD <280> be set to whichever is smaller, the `Timeout` value in the request or 300 seconds.
- If the `Timeout` value in the request is zero, the `Timeout` value in the response SHOULD <281> be set to an implementation-specific value.
- `Open.DurableOpenTimeout` SHOULD <282> be set to the `Timeout` value in the response.

<281>: If the `Timeout` value in the request is zero, Windows 8.1 and Windows Server 2012 R2 SMB2 servers set `Timeout` to 180 seconds.

<282>: Windows 8 and Windows Server 2012 R2 SMB2 servers set `Open.DurableOpenTimeout` to 60 seconds

4.1.4 Disconnected persistent opens without handle lease (MS-SMB2 3.3.5.9)

When receiving a (non-reconnect) create request we must check if there's any disconnected Persistent Handle

MS-SMB2 3.3.5.9 Receiving an SMB2 CREATE Request

When the server receives a request with an SMB2 header with a Command value equal to SMB2 CREATE, message handling proceeds as described in the following sections.

If Connection.Dialect belongs to the SMB 3.x dialect family and the request does not contain SMB2_CREATE_DURABLE_HANDLE_RECONNECT Create Context or SMB2_CREATE_DURABLE_HANDLE_RECONNECT_V2 Create Context, the server MUST look up an existing open in the GlobalOpenTable where Open.FileName matches the file name in the Buffer field of the request. If an Open entry is found, and if all the following conditions are satisfied, the +server SHOULD <WBN> fail the request with STATUS_FILE_NOT_AVAILABLE:

- Open.IsPersistent is TRUE
- Open.Connection is NULL

<WBN> If Open.ClientGuid is not equal to the ClientGuid of the connection that received this request, Open.Lease.LeaseState is equal to RWH or Open.OplockLevel is equal to SMB2_OPLOCK_LEVEL_BATCH, windows servers will attempt to break the lease/oplock and return STATUS_PENDING to process the create request asynchronously. Otherwise, if Open.Lease.LeaseState does not include SMB2_LEASE_HANDLE_CACHING and Open.OplockLevel is not equal to SMB2_OPLOCK_LEVEL_BATCH, windows servers return STATUS_FILE_NOT_AVAILABLE.

4.1.5 Set Open.IsPersistent to FALSE when closing files

4.1.6 Set Open.IsPersistent to FALSE when disconnecting shares

4.1.7 When to grant PH

Windows server grant PH for (**SMB 2.2: Bigger, Faster, Scalier**):

- Handles opened for delete access
- Handles opened for read, write or execute access
- State changing creates (CREATE, OVERWRITE, OVERWRITE_IF)

4.2 FSA layer

4.2.1 Subsystem: leases

"The server is not required to explicitly track lease state across server failovers. The client is expected to re-request its lease when reconnecting Persistent Handle."

"For lease breaks on Persistent Handles, the server holds lease breaks until the client reconnects and resumes its handle. Server waits for up to DurableHandleV2 timeout."

([SMB 2.2: Bigger, Faster, Scalier](#)).

5 Development Plan

5.1 Research

- SMB3 capabilities
- Persistent Handles and Leases
 - network vs server failure
 - lease epoch
- merge new create replay database (uses CreateGUID as record key) and smbXsrv_open_global.tdb database (uses FileId.Persistent as record key)
- fsp->gen_id, prototype stores unhashed Persistent File ID
- Persistent Handles on directories
- Write Time Handling ([Samba Bug 13594](#))
- Delete-on-close behaviour
- unbuffered IO on Persistent Handles and alignment requirements

Estimate: 6 weeks

5.2 Implementation

5.2.1 Data Model

- add record versioning support to locking.tdb, leases_db.tdb and brlock.tdb

- maintain and use additional CreateGUID index on `smbXsrv_open_global.tdb` ([Samba bug #13703](#))
- performance: maybe merge `brlock.tdb` into `locking.tdb`

Estimate: 2 weeks

5.2.2 dbwrap

- finalize prototype patches
- add support for caller maintained additional indexes
- batch persistent database stores
- performance: cluster node local Persistent File IDs

Estimate: 4 weeks

5.2.3 VFS

- finalize prototype patches

Estimate: 1 week

5.2.4 FSA

- finalize prototype patches
- granting leases/oplocks on Persistent Handles
- handling lease breaks on PH: network disconnects
- handling lease breaks on PH: server failure
- lease break delay and Persistent Handles (use `Durablehandlev2` timeout)
- unbuffered IO on Persistent Handles
- enforce **strict rename** semantics

Estimate: 5 weeks

5.2.5 SMB

- finalize prototype patches
- SMB3 capabilities
- Resilient Handles
- alignment checks for unbuffered IO
- Lock Replay (**proposed patch**)

Estimate: 3 weeks

5.2.6 Persistent Handle Cleanup

- review design of current cleanup master election
- finalize prototype patches

Estimate: 3 weeks

5.3 Clustered Samba CI

- add Clustered Samba to CI

Estimate: 2 weeks

5.4 CI Coverage of Persistent Handles

- test granting Persistent Handles
- test reconnecting Persistent Handles
- test contending Persistent Handles
- test Resilient Handles
- test delayed lease breaks
- test uninterrupted IO
- test lock replay
- test write fencing
- test Persistent state cleanup

Estimate: 8 weeks

5.5 End to End Tests

- failover test with real Windows clients

Estimate: 2 weeks

5.6 Related Changes

- Write Time Handling ([Samba Bug 13594](#))

Estimate: 4 weeks

5.7 Summary

Estimated development effort: 200 days

6 Appendix

6.1 Revision Info

git revision 8ec8063e6a4a37495de944aab135425de6d4ad5b from 2019-01-30.