NFS locks

After the publication of the article "Linux-HA-NFS-Cluster für v3 und v4" in the iX issue 12/23, a reader commented that the topic of file-locking was too short. This topic will be taken up again here.

With NFS locking, a general distinction must be made between the behavior of NFSv3 and NFSv4.

NFSv3

NFSv3 is the first version of NFS where all tasks are transferred to several daemons. This includes file locking. The following tasks are assigned to the daemons:

The nfsd:

Files requested by the client are provided by the *nfsd* daemon. The deamon itself cannot lock the open files, other deamons are responsible for this.

The statd

The statd is used to monitor the status of a connection; it manages the respective statuses of both the client and the server. The *statd* uses the *network status monitor (nsm) to* ensure that the status is always up-to-date during the client's access time.

The Network Lock Manager (nlm)

The nlm, which is provided via the *lockd,* now ensures on an NFSv3 server that two clients do not open the same file for writing. In addition, the *nlm* can detect together with the *nsm* when a client reboots. The *nlm* can then release all the client's locks.

The locks are always stored in the file system of the NFS server, namely in the directories */var/lib/nfs/sm* and */var/lib/nfs/sm.back. The* directories contain files with the IP addresses of the clients that have locks on the NFS server. More detailed information can be found in the man page for *rpc.statd*. You can use *flock*, a small utility program, to check whether the locks are working.

As long as there is only one NFS server, this works very well. But what happens if a failover cluster is to be set up for NFSv3? Because the files in which the locks are managed are only ever available locally on the NFSv3 server. In this case, the locks must somehow be transferred to all nodes of the cluster. Because if the locks cannot be read, data may be lost. When using the Ganesha NFS server, the locks can be transferred to the cluster file system via two variables. This means that all nodes access the same information.

This option is not available with the *nfs-kernel-server.* CTDB and Gluster must therefore take over the management of the locks here. CTDB manages all sessions in TDB databases. As the TDB databases are cluster-capable, all nodes can access the information and all nodes always know which client is using which file.

At file system level, Gluster takes over the locking of the files. Even when using Gluster without NFS, the system must always be able to manage competing locks.

Gluster and other cluster file systems use *fcntl()* for advisory locks, see more:
https://pubs.opengroup.org/onlinepubs/9699919799/functions/fcntl.html
Another thing to note about *fcntl() is that there is* no way to tell an application when a lock is somehow lost. Therefore, locks must not be lost.

But what happens when an NFS server is restarted? After an NFS server is restarted, it is in the "grace period" for a guaranteed period of time. During this state, the individual locks can only be released by the clients that held the lock before the reboot.

Since NFSv3 only ever manages the locks locally via its own daemons, CTDB must ensure that the loccs are always identical on all nodes, even in the clustered environment. For this purpose, the script */etc/ctdb/statd-callout* is called each time the monitor is run. As the locks are stored in TDB files and are identical on all nodes, all locks are available even if a node fails.

If there is a problem with the locks when running the script, the *nlm is* restarted on all CTDB nodes and thus all nodes are moved to the "grace period".

NFSv4
With NFSv4, the support of locks has become part of the NFS protocol; an extra daemon is no longer required. Support for locks is structured in such a way that an RPC call is no longer required. The status associated with a lock is managed on the server according to a lease-based model. The server defines a time period (the lease) for all states that an NFS client has. If the client does not renew its lease within the defined period, the server can release all states associated with the client's lease. The client can renew its lease using a RENEW operation, or implicitly using other operations (especially READ). A large part of the management of the locks is therefore performed by the client. The client requests the lease, it can extend or terminate the lease. The server monitors the leases. In the event that a client no longer responds or another client requests the lock on the file, the server can terminate the lease.

With NFSv4, a client can choose whether to lock the entire file or a byte range within a file. NFS client requests to lock files can be managed with the *llock*(Y|N) parameter of the mount command or as an installation default. However, the use of *llock* as a mount option must not be used in a cluster environment. With the *-o llock* option, all locks are always local to the client and other clients do not receive any information about the locks. This type of lock only makes sense if only one client is accessing an NFS share. The local locks then speed up access.

NFS only supports advisory locks. With advisory locking, the operating system keeps track of which files have been locked by which process, but does not prevent a process from writing to a file that has been locked by another process. This means that a process can ignore an advisory lock if it has the appropriate rights.

But then why no mandatory locks?
On the one hand, the mandatory locks under Linux are not very reliable and on the other hand, additional steps are necessary for the file system to support the mandatory locks. Also, in the event that a process holding a mandatory lock crashes, mandatory locks can no longer be released even by the root. More information can be found at https://www.baeldung.com/linux/file-locking or https://linuxhandbook.com/file-locking/.

How does CTDB work with GlusterFS to provide the locks?
CTDB basically only manages the individual client sessions, whereby all client information is always saved in TDB files. These TDB files are always managed on all nodes.
If a client now changes the node with its IP address, the information as to which
Client locks are of course always present on all nodes. This is realized via TDB files that can be read and written by all nodes. CTDB transfers the lock status between all nodes during the monitor events. This means that up to ~15 seconds of the lock status can be lost during a failover. The value can be set via the CTDB variable "MonitoringInterval". The command "ctdb setvar MonitorInterval <seconds>" can be used for this purpose.

However, the actual locking is always performed by the file system on which the files are stored. In the case of a setup with GlusterFS and CTDB, this is GlusterFS.
File locking in a gluster volume works in a similar way to other distributed file systems, although there are some specific aspects to consider:

GlusterFS supports POSIX locks, which make it possible to lock files on a Gluster volume. POSIX locks include both exclusive (write) and shared (read) locks. These locks are used by applications to ensure that only one process can write to a file at any given time.

GlusterFS also supports flock() system calls. This is a simpler way of locking files by locking an entire file object.

Stale locks and self-healing:
GlusterFS keeps track of locks even if a particular process crashes or a network interruption occurs. After such an event, GlusterFS ensures that no *stale locks* are present by checking the lock status and cleaning it up if necessary. This is a task of GlusterFS *self-healing*

Locks can be tracked and deleted in the gluster volume. This is always important when an application no longer releases a lock. You can find out more about file locking in GlusterFS at https://rajeshjoseph.gitbooks.io/test-guide/content/appendices/chap-Troubleshooting.html.

And the client?
Locking can be influenced not only on the server side, but also on the client side with the mount options. It should be noted here that the different behavior of the soft/hard options, as well as the *timeo* option already mentioned in the previous article, influences the delay when restoring a connection.

soft vs. hard
The *hard* option is the default if neither of the two options has been specified. In this case, NFS requests from the client are repeated indefinitely until an NFS server responds again. If the *soft option is selected, the* additional option *retrans* can be used to specify how often the client should attempt to reconnect to the server. If no value is specified for *retrans*, the client attempts to establish the TCP connection to the server twice.
Reference is often made to the *intr/nointr* options, but these options are only relevant for systems with a kernel prior to 2.6.25. The option is ignored for current systems.

If *soft is* now selected for Client and the NFS node is suddenly no longer accessible, the client will only perform a predefined number of connection attempts. The client will then release the locks held by it, but only if it was mounted via NFSv4. If the connection to another NFS node is then re-established, the locks must be requested again by the client. The application that has established a connection to the NFS server on the client receives the error message *"server not responding"*.
If the *hard* option is selected instead, the client will keep trying to establish the connection. If the IP address now switches to another node and the client reconnects, the lock would still be active. The switching time of the IP address from one CTDB node to another is approx. one second.

But what if another client wanted to open the file in the meantime? The NFS node that is still active would try to reach the client that originally held the lock and release the lock. However, the client is temporarily unavailable while the IP address is being switched, so the NFS server would release the lock for this reason and pass the lock on to the new client. If the original client now tries to lease the lock again, the lock would no longer be valid and the NFS server would reject the connection.

But which is the better option? That depends on what is more important: data security or the client's ability to react quickly. If responsiveness is important, then the *soft* option should be selected. If data security is more important, *hard should be selected*. However, the problems of the *soft option* can be improved by the *retrans* option. More frequent attempts to re-establish the connection improve data consistency in the event of a node failure.

Conclusion:
If NFS is used together with CTDB and GlusterFS, it is also guaranteed that files are locked in such a way that no data loss can occur. However, since NFS can no longer perform all locking tasks independently at this point, in extreme cases it can lead to a split-brain in the cluster, which then has to be resolved manually. NFSv4 clearly has the advantage here that locking is initiated by the client and therefore the locks are retained on the client if a CTDB node fails and a node is panned as a result.