17

The Remaining Oddities

> The first 90% of the job
> takes 90% of the time.
> The remaining 10%
> of the job requires
> another 90% of the time.
>
> — Unknown (but oh so true)

Promises were made, and promises should be kept.

Remember that closet full of concepts that burst open and spilled out all over the floor? Well, we have managed to clean up a good deal of the mess, but there are still a few things that we said we would put away — and we will. We can provide a brief explanation of each of these as we shove them back into the closet, just so you are not surprised when you stumble across them in the literature.

17.1 Opportunistic Locks (OpLocks)

OpLocks are a caching mechanism.

A client may request an OpLock from an SMB server when it opens a file. If the server grants the request, then the client knows that it can safely cache large chunks of the file and not tell the server what it is doing with those cached chunks until it is finished. That saves a lot of network I/O round-trip time and is a very big boost to performance.

The problem, of course, is that other clients may want to access the same file at the same time. As long as everyone is just reading the file things are okay,

but if even one client makes a change then all of the cached copies held by the other clients will be out of sync. That's why OpLock handling is a bit tricky.

There are two types of OpLocks that a client may request:

- Exclusive, or
- Batch.

We came across these two when digging into the `SMB_HEADER.FLAGS` field way back in Section 12.2 on page 202. In olden times, the client would request an OpLock by setting the `SMB_FLAGS_REQUEST_OPLOCK` bit and, optionally, the `SMB_FLAGS_REQUEST_BATCH_OPLOCK` bit in the `FLAGS` field when opening a file. Now-a-days the `FLAGS` bits are (supposedly) ignored and fields within newer-style SMBs are used instead.

Anyway, an Exclusive OpLock can be granted if no other client or application is accessing the file at all. The client may then read, write, lock, and unlock the cached portions of the file without informing the server. As long as the client holds the Exclusive OpLock, it knows that it won't cause any conflicts. It's sort of like a kid sitting in a corner of the kitchen with a spoon and a big ol' carton of ice cream. As long as no one else is looking, that kid's world is just the spoon and the ice cream.

Batch OpLocks are similar to Exclusive OpLocks except that they cause the client to delay sending a `CLOSE` SMB to the server. This is done specifically to bypass a weirdity in the way DOS handles batch files (batch files are the DOS equivalent of shell scripts). The problem is that DOS executes these scripts in the following way:

1. Set offset to zero (`0`).
2. Open the batch file.
3. Seek to the stored offset.
4. If EOF, then exit.
5. Read one line.
6. Store the current offset.
7. Close the batch file.
8. Execute the line.
9. Go back to step 2.

Yes, you've read that correctly. The batch file is opened and closed for every line. It's ugly, but that's what DOS reportedly does and that's why there are Batch OpLocks.

To make Batch OpLocks effective, the client's SMB layer simply delays sending the `CLOSE` message. If the file is opened again, the `CLOSE` and `OPEN` simply cancel each other out and nothing needs to be sent over the wire at all. That also means that the client can keep hold of the cached copy of the batch file so that it doesn't have to re-read it for every line of the script.

There is also a third type of OpLock, known as a Level II OpLock, which the client cannot request but the server may grant. Level II OpLocks are, essentially, "read-only" OpLocks. They permit the client to cache data for reading only. All operations which would change the file or meta-data must still be sent to the server.

Level II OpLocks may be granted when the server cannot grant an Exclusive or Batch OpLock. They allow multiple clients to cache the same file at the same time so, unlike the other two, Level II OpLocks are not exclusive. As long as all of the clients are just reading their cached copies there is no chance of conflict. If one client makes a change, however, then all of the other clients need to be notified that their cached copies are no longer valid. That's called an OpLock Break.

17.1.1 *OpLock Breaks*

It's called an "OpLock Break" because it involves breaking an existing OpLock. The more formal term is "revocation," but no one actually says that when they get together after hours to sit around, drink tea, and whine about CIFS.

OpLock Breaks are sent from the server to the client. This is unusual, because SMB request/response pairs are *always* initiated by the client. The OpLock Break is sent out-of-band by the server, which is against the rules... but this is CIFS we're talking about. Who needs rules?

The OpLock Break is sent in the form of a `SMB_COM_LOCKING_ANDX` message. The server may send this to reduce an Exclusive or Batch OpLock to a Level II OpLock, or to revoke an existing OpLock entirely. In either case, the client's immediate responsibility is to flush its cache to comply with the new OpLock status. If the client held an Exclusive or Batch OpLock, it must send all writes to the server and request any byte-range locks that it needs in

order to continue processing. If the OpLock has been reduced to a Level II OpLock, the client may keep its local cache for read-only purposes.

Note that there is a big difference between OpLocks and the more traditional types of locks. With a traditional file or byte-range lock, the client is in charge once it has obtained a lock. It can maintain it as long as needed, relinquishing it only when it is finished using it. In contrast, an OpLock is like borrowing your neighbor's lawnmower. You have to give it back when your neighbor asks for it.

Support for OpLocks is optional on both the client and the server side, but implementing them provides a hefty performance boost. More information on OpLocks may be found in the Paul Leach/Dan Perry article *CIFS: A Common Internet File System* (listed in the References), as well as the usual sources.

17.2 Distributed File System (DFS)

The CIFS **D**istributed **F**ile **S**ystem (DFS) is not nearly as fancy as it sounds. It is simply a way to collect separate shares into a single, virtual tree structure. It also has some limited ability to provide fileserver redundancy and load balancing.

The key feature of DFS is that it can create links from within a shared tree on one server to shares and directories on another, thus providing a single point of entry to a *virtual* SMB tree. From the user's perspective, the whole thing looks like a single share, even though the resources are scattered across separate SMB servers.

Clear as mud? Perhaps an illustration will help...

In Figure 17.1, the client is shown attempting to access a file on server `PETSERVER`. Well, that's where the client *thinks* the file resides. On the server side, the name `CORGIS` in the `DOGS` directory is actually a link to another UNC pathname, `\\DATADOG\CORGIS`. Following that link leads us to a different share on a different server.

The server offering the DFS share (`PETSERVER`, in our example) does not act as a proxy for the client. That is, it won't follow the DFS links itself. Instead, the server sends an error code to the client indicating that there is some additional work to be done. The error code is either a DOS code of `ERRSRV/ERRbadtype` (`0x02/0x0003`), or an NT_STATUS code of `STATUS_DFS_PATH_NOT_COVERED` (`0xC0000257`).

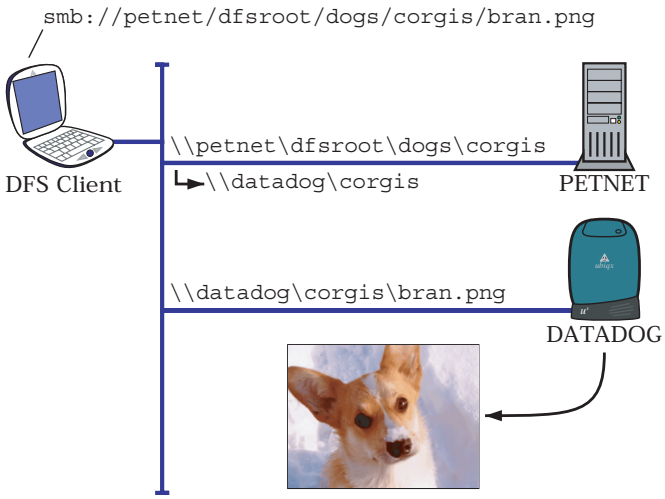

Figure 17.1: *Distributed File System*

The client is using an SMB URL to access an image of a Pembrokeshire Welsh Corgi playing in the snow. The URL is translated into UNC format for use with the SMB protocol. The server traverses the UNC path until it reaches `corgis`, which is a link to a share on a different server.

```
CORGIS ==> \\DATADOG\CORGIS
```

The client is redirected to the new server, where it finally finds the file it wanted.

The client's task, at this point, is to query the server to resolve the link. The client sends a `TRANS2_GET_DFS_REFERRAL` which is passed to the server via the Trans2 transaction mechanism, briefly described earlier. The client will use the information provided in the query response to create a new UNC path. It must then establish an SMB session with the new server. This whole mess is known as a "DFS referral."

It was mentioned above that DFS can provide a certain amount of redundancy. This is possible because the links in the DFS tree may contain multiple references. If the client fails to connect to the first server listed in the referral it can try the second, and so on. DFS can also provide a simple form of load balancing by reshuffling the order in which the list of links is presented each time it is queried. Of course, load balancing and redundancy are only workable if all of the linked copies are in sync.

A quick search on the web will turn up a lot of articles and papers that do a better job of describing the behavior of DFS that the blurb provided here.

If you are planning on implementing DFS, it is worthwhile to read up on the subject a bit, just to get a complete sense of how it is supposed to work from the user or network administrator's perspective. The SNIA doc provides enough information to get you started building a working client implementation. The server side is more complex because doing it right involves implementing a set of management functions as well.

17.3 DOS Attributes, Extended File Attributes, Long Filenames, and Suchlike

These all present the same problem.

The CIFS protocol suite is designed, in its heart and soul, to work with DOS, OS/2, and Windows systems. As a result, the protocols that make up the CIFS suite have a tendency to reflect the behavior of those operating systems.

DOS, of course, is the oldest and simplest of the IBM/Microsoft family of PC OSes. The filesystem used with DOS is the venerable **F**ile **A**llocation **T**able (FAT) filesystem which, according to legend, was originally coded up by Bill Gates himself. The characteristics of the FAT filesystem should be familiar to anyone who has spent any time working with DOS. Consider, for example, the following FAT features:

Case-insensitive filenames

Case is ignored, though file and directory names are stored in uppercase.

8.3 filename format

A name is a maximum of eight bytes in length, optionally followed by a dot ('.') and an extension of at most three bytes. For example: `FILENAME.EXT`.

No users or groups

FAT does not understand ownership of files.

Six attribute bits

FAT supports six attribute bits, stored in a single byte. The best known are the Archive, Hidden, Read-only, and System bits but there are two

more: Volume Label and Directory. These are used to identify the type and handling of a FAT table entry.

It's a fairly spartan system.

There are improvements and extensions that have appeared over the years. The FAT32 filesystem, for example, is a modified version of FAT that uses disk space more efficiently and also supports much larger disk sizes than the original FAT. There is also VFAT, which keeps track of both 8.3 format filenames and longer secondary filenames that may contain a wider variety of characters than the 8.3 format allows. VFAT long filenames are case-preserving (but not case sensitive) so, overall, VFAT allows a lot more creativity with file and directory names.[1]

Even with these extensions, the semantics of the FAT filesystem are not sufficient to meet the needs of more powerful OSes such as OS/2 and Windows NT. These OSes have newer, more complex filesystems which they support in addition to FAT. Specifically, OS/2 has HPFS (**H**igh **P**erformance **F**ile **S**ystem), and Windows NT and W2K can make use of NTFS (**N**ew **T**echnology **F**ile **S**ystem). These newer filesystems have lots and lots of features which, in turn, have to be supported by CIFS.

Problems arise when the server semantics (made available via CIFS) do not match those expected by the client. Consider, for instance, Samba running on a Unix system. Unix filesystems typically have these general characteristics:

Case-sensitive filenames

Case is significant in most Unix filesystems. File and directory names are stored with case preserved.

Longer, more complex filenames

Unix filesystems allow for a great deal of creativity in naming files and directories.

Users and groups

Unix filesystems assign user and group ownership to each directory entry.

---

1. Digging through the documentation, it appears that the FAT family consists of FAT12, FAT16, FAT32, and VFAT. There is documentation on the web that provides implementation details, if you are so inclined.

More, and different, attribute bits

There are three sets of three bits each used for basic file access (read, write, and execute permissions for user, group, and world). There are additional bits defined for more esoteric purposes.

Now consider a Windows application that requires the old 8.3 name format. (Such applications do exist. They make calls to older, 16-bit OS functions that assume 8.3 format.) Unlike VFAT, Unix filesystems do not normally keep track of both long and short names. That causes a problem, and Samba has to compensate by generating 8.3 format names on the fly. The process is called "Name Mangling."

There are other gotchas too. Indeed, name mangling is just the tip of the proverbial iceberg.

One solution that some CIFS vendors have been able to implement is to develop a whole new filesystem for their server platform, one that maintains all of the required attributes and maps between them as necessary. This is a pain, but it works in situations in which the server vendor has control over the deployment of their product. One such filesystem is Microsoft's NTFS, which can handle a very wide variety of attributes and map them to the semantics required by Apple Macintosh clients, Unix clients, DOS clients, OS/2 clients...

You've got the basic idea. Let's run through some of the trouble spots to give you a sense of what you're up against.

Long filenames

Long filenames can be much more descriptive than the old 8.3 names. The problem, of course, is that CIFS must support both long and short (8.3) names to be fully compatible with all of the potential clients out there. Even if a server supports only the NT LM 0.12 dialect, there will still be instances when the 8.3 format is required. Sigh.

DOS attributes

These are the six attribute bits that are supported by the FAT filesystem. These do not map well to the file protections offered by other filesystems. Compare these, for example, against the attribute bits offered by Unix systems.

The timestamps stored in the FAT filesystem may also be different from those used by other systems.

Extended File Attributes

These are an extended set of attribute bits and flags available on systems using the NTFS filesystem. They are a 32-bit superset of the set offered by the FAT system. Extended File Attributes are described in Section 3.13 of the SNIA doc.

The term "Extended File Attributes" is also sometimes misused when discussing NTFS permissions. Permissions are different; they are associated with **A**ccess **C**ontrol **E**ntries (ACEs), and ACEs are gathered together into **A**ccess **C**ontrol **L**ists. There's a whole bigbunch of stuff there that could be explored — and would be, if this were a book about implementing NTFS.

Extended Attributes

These should get special mention because, it seems, CIFS is sufficiently complex that terminology has to be recycled. **E**xtended **A**ttributes (EAs) are not the same as Extended File Attributes.

EAs are a feature of HPFS and, therefore, are supported by NTFS. Basically, they are a separate data space associated with a file into which applications may store additional data or metadata specific to the application (things like author name or a file comment).[2]

CIFS offers facilities to support all of these features and more. That's good news if you are writing client code, because you can pick and choose the sets of attributes you want to support. It's bad for server systems, which may need to offer various levels of compatibility in order to contend with client expectations.

---

2. NTFS is a complex filesystem based on some simple concepts. One such concept is that each "file" is actually a set of "attributes" (records). Many of these attributes are predefined to contain such things as the short name, the long name, file creation and access times, etc. The actual content of the file is stored in a specific, predefined "stream," where a stream is a particular kind of attribute. NTFS supports OS/2-style Extended Attributes in another type of NTFS attribute... and it just gets more confusing from there. There is a lot of documentation on the web about the workings of NTFS, and there is a project aimed at implementing NTFS for Linux.