Exploring SMB

Andrew Tridgell

May 8, 2000

1 Introduction

The Samba Team have been developing an implementation of an SMB server for many years. Recently we have reached the stage where we are looking to fill in some of the darker recesses of the implementation - implementing not just a "good enough" emulation of the SMB protocol but instead aiming for as close to a perfect emulation of the file sharing semantics of a NT server as is possible within a Unix environment. The problem is in working out exactly what those semantics are.

This paper describes three challenges along these lines and how we met them by building tools to analyze the exact behavior of an NT server. The tools provide a way of exactly determining the semantics of share modes, wildcard matching and byte range locks. These three critical but complex areas of file sharing semantics have been rather mysterious and poorly described in the available documentation, and are thus excellent candidates for automatic analysis.

Note that this paper only gives a high level overview of these tools and the resulting analysis. For a complete analysis you will need to run the tools themselves, which are freely available in the Samba CVS tree.

2 Wildcard Matching

The first area I will consider is wildcard matching. There has scarcely been a release of Samba where we have not made some adjustment to the wildcard matching code as we discover new and ever more esoteric features of the wildcard matching algorithms that NT uses.

You might wonder why we bother with the fine details of these algorithms. The reason is that application developers become unconsciously reliant on the exact wildcard matching semantics of NT - their applications tend

to be tested against NT servers and often fail when a SMB server deviates even slightly from the wildcard matching rules that NT implements. As we want these applications to run perfectly on a Samba server we must copy the exact semantics of NT.

Wildcard matching in the SMB protocol is quite strange indeed. A MS-Windows user sees only a very simple wildcard matching system involving * and ?, but underneath something much more complex is going on. The client OS converts these simple wildcard requests into patterns that contain 5 wildcard characters * ? ¡ ¿ and “. The first two are fairly familiar beasts, but the last three are quite strange.

The CIFS documentation dismisses these three extra wildcard characters by providing a very simple mapping between these characters and the usual * and ? characters. Unfortunately this simple mapping is so simple because it is completely incorrect, as a few minutes of testing confirms. So now we are left with the problem of working out exactly what these characters actually mean.

To worth this out I resorted to automatic testing. A program called masktest was written that connects to a SMB server and tests an unending stream of random wildcard/filename pairs against a proposed wildcard matching algorithm built into the test program. For each test pair the following is done:

- The filename is created on the server in an otherwise empty directory
- A wildcard directory search is done on the directory using the chosen wildcard pattern
- The return result is compared against the internal algorithm

The comparison of return result is interesting. We are not only looking for a match/no-match for the wildcard pattern to the filename, we are also looking to see whether that wildcard pattern produced a match on the special “.” and “..” directories. Thus each test produces 3 results, each of which is compared against the 3 results from the built in algorithm.

The other interesting part of the problem is the selection of character sets to choose the wildcard and filename patterns from. If all possible characters were allowed then the probability of a match would be very low, and testing would be too slow. Instead we allow the user to choose the subsets of characters to choose the random filenames and wildcard patterns from. A good choice is usually to keep the range of general-purpose characters (such as A to Z) quite small.

The next challenge is to work out the wildcard matching algorithm used by NT. To do this I restricted the wildcard list to include only 3 lowercase letters "abc" plus only one wildcard character, and worked on getting the builtin algorithm to work correctly with these simplified sets of wildcards. This allowed much easier analysis of mis-matches than would be possible if more than one wildcard character could be used at a time.

After a number of days of analysis I ended up with an algorithm that treats each type of wildcard identically to NT. Luckily the result ended up quite simple, and was implemented in about 60 lines of C code. For the final result see the module ms_fnmatch.c in the Samba development branch.

3 Share Modes

The next challenge was the share mode access table. Share modes are quite a complex but critical part of the SMB protocol - they determine what open modes are possible on a file that is already opened. Some experimentation had shown that there are a number of dimensions to the problem:

- which of the 6 share modes (DENY_READ, DENY_WRITE, DENY_NONE, DENY_ALL, DENY_DOS and DENY_FCB) was used by the first open of the file.
- which of the 6 share modes was used in the second open
- which of 3 open types was requested on the first open (RDONLY, RDWR or WRONLY)
- which of the 3 open types was requested on the second open
- what the filename itself is (files ending in *.exe, *.sym, *.dll and *.com are treated specially)
- whether the second open is made by the same SMB connection as the first open

This gives a total of 432 combinations. Because this number is quite small the obvious was to solve this problem is to test each combination against a NT server and put the result in a table for Samba to use. While this was done at first (using the "DENY1" and "DENY2" tests in smbtorture), it didn't provide a very elegant solution and didn't give any insight into how NT was behaving and why particular behavior was implemented.

To provide a more elegant solution a public challenge (with a Samba t-shirt as the prize) was issued for someone to produce the smallest neat C function to implement the table. This produced quite a large response, with about a dozen people putting quite a large effort into reducing the table to a small logic function. Several electrical engineering and software engineering logic tools were used, plus quite a lot of manual function tuning.

The eventual winner was Dr Paul Mackerras, with a very simple and elegant function that used a small number of bit operations and a particular ordering of the enumerated types to produced the desired result. His entry also took advantage of the fact that the DENY_FCB mode can only be specified with a RDWR open mode due to the way that particular share mode is encoded on the wire.

The end result was the following snippet of code:

```
static int access_fcb(int new_deny, int old_deny, int old_mode,
                      BOOL same_pid, BOOL isexe)
{
        if (!same_pid || old_deny < DENY_DOS || new_deny < DENY_DOS
            || old_deny == DENY_DOS && (isexe || old_mode == DOS_OPEN_RDONLY))
                return AFAIL;

        return AALL;
}

static int access_dos(int new_deny, int old_deny, int old_mode,
                      BOOL same_pid, BOOL isexe)
{
        if ((new_deny == DENY_DOS && old_deny == DENY_DOS && same_pid)
            || isexe || old_mode == DOS_OPEN_RDONLY)
                return (isexe? AALL: old_mode | AREAD) & ~old_deny;

        return AFAIL;
}

static int access_table2(int new_deny, int old_deny, int old_mode,
                         BOOL same_pid, BOOL isexe)
{
        if (new_deny == DENY_FCB || old_deny == DENY_FCB)
                return access_fcb(new_deny, old_deny, old_mode,
                                  same_pid, isexe);
```

```
        if (new_deny & old_mode)
                return AFAIL;

        if (new_deny == DENY_DOS || old_deny == DENY_DOS)
                return access_dos(new_deny, old_deny, old_mode,
                                  same_pid, isexe);

        return AALL & ~old_deny;
}
```

4 Byte Range Locking

Byte range locking is another one of those essential components of the protocol that is very badly documented. MS-Windows programs use large numbers of byte range locks and often rely on the exact semantics of the servers locking implementation.

To address this problem I developed a program that can test one byte range locking implementation against another by connecting to 2 SMB servers in parallel and perform identical randomized operations on the two servers until a difference in behavior is observed. The program is then able to analyze the different behaviors to produce a minimal set of random lock operations that triggers the difference.

The program (called rather boringly "locktest") creates two connections to each server, and opens the same file 2 times on each connection, giving a total of 8 file handles. Random locking operations are then chosen from the following set:

- read lock a random range of bytes
- write lock a random range of bytes
- unlock a random range of bytes
- close and re-open one of the handles on each server

To ensure that locks overlap quite often a limited range of bytes is used (defaulting to 100).

Once a difference is found, the system analyses the series of locking requests performed thus far to try to find the smallest set of locks that reproduce the difference. This is done by removing, one at a time, operations

from the list of lock operations and testing to see whether the difference remains. If the difference does remain then we know that the operation that was removed was not essential for the difference, and thus it can be discarded, otherwise it must be re-instated.

In this way locktest is able to whittle down the list of lock operations that produce a difference until a very small list remains - typically a list just a handful of operations long. This remaining list can then be hand analyzed to see which of the two implementations is correct and how to fix the problem.

One interesting thing that came out of this testing is that NT has a serious byte range locking bug. We found that a series of 4 operations like the following fails where it should succeed:

```
lock bytes 1-10 on file descriptor 1
lock bytes 20-30 on file descriptor 2
close and re-open file descriptor 2
lock bytes 20-30 on file descriptor 1
```

The exact numbers aren't all that important. The problem appears to be that NT sometimes does not remove locks on file descriptors when the file is closed. Interestingly, NT does get this right sometimes. If you run exactly the same series of operations a hundred times then NT will get the right answer about 20% of the time.

One unfortunate side-effect of this bug is that we can't completely validate the Samba implementation against NT. We are looking forward to Microsoft fixing this bug so we can verify that our byte range locking implementations are identical.

5 Conclusion

Automated randomized testing has proved very valuable in probing the darker recesses of the SMB protocol. I would encourage all SMB implementors to seriously look at automated tests, or use the ones provided with Samba.