Towards Full Coverage Testing

Jim Myers and Andrew Tridgell
IBM Almaden Research Center

myersjj@us.ibm.com, tridge@au.ibm.com
Old testing method

• The Samba project has previously developed testsuites of 3 main kinds:
  • ad-hoc tests for a range of specific conditions
  • full-coverage tests for a very small range of operations
  • randomised testing for a very small range of operations

• This approach did work to some extent, but suffered from some major drawbacks:
  • many parts of the protocol remained completely untested
  • many fields untested within the tested parts of the protocol
  • difficult to expand to be comprehensive
New Testing Methodology

• The new testing system in Samba4 is based on a few basic components:
  • a comprehensive raw client library
  • individual tests covering every field of every call
  • a randomised dual-server tester with broad coverage
  • a "CIFS on CIFS" storage backend for the Samba4 server

• These components work together to provide a testing capability far beyond what could be achieved with our earlier testsuites
Raw Client Library

• The heart of the new testing system is a 'raw' comprehensive client library. Unlike our previous client library this allows easy generation of all SMBs, with control over all fields in each request.

• New features include:
  • async interfaces
  • oplock support
  • no 'smarts' - send exactly what is asked for

• Note that it takes a lot code to use the new interface compared to the old one. The old interface is still available as a wrapper.
C interface to raw library

Old interface:

```c
int fnum = cli_open(cli, "\\test.dat", O_RDWR, DENY_READ);
```

New Interface:

```c
NTSTATUS status;
union smb_open io;

io.generic.level = RAW_OPEN_OPENX;
io.openx.in.flags = OPENX_FLAGS_ADDITIONAL_INFO;
io.openx.in.open_mode = OPEN_MODE_ACCESS_RDWR;
io.openx.in.search_attrs = FILE_ATTRIBUTE_SYSTEM | FILE_ATTRIBUTE_HIDDEN;
io.openx.in.file_athrs = 0;
io.openx.in.write_time = 0;
io.openx.in.open_func = OPENX_OPEN_FUNC_OPEN;
io.openx.in.size = 0;
io.openx.in.timeout = 0;
io.openx.in.fname = "\\test.dat";

req = smb_raw_open_send(tree, &io);
status = smb_raw_open_recv(req, mem_ctx, &io);
```
Individual tests

• Built on top of the raw client library is a set of individual tests:
  • Each SMB request is individually tested, with separate tests for every information level of every call
  • Every field of every request is tested, but only with a limited range of values
  • 'Correct' results are in most cases defined by how W2K3 behaves, except where this is very obviously incorrect
  • If a value can be returned in N ways, then all N are tested to confirm that they are equal
  • Includes testing of EAs, streams and many unusual requests
String Termination

• Testing for correct string termination by servers has proved to be very important

• Each test that retrieves a string tests that the server uses correct alignment and termination for that request

• The 'wire length' fields are also tested, as sometimes these should include the termination and sometimes they should not
Level Scanners

- A level scanner is a program that tries every subcall and information level of a CIFS transaction request such as TRANS2
- The test suite includes two types of level scanners:
  - a scanner that finds calls and levels, their size and their request type
  - a scanner that automatically determines what levels are aliases of other levels
CIFS Backend

- A new feature in Samba4 is the ability to define arbitrary storage backends at the 'raw' CIFS level
- A backend that has proved incredibly useful for testing is the 'CIFS' backend, that uses a remote CIFS server for all operations:
  - uses the raw client library for remote server access
  - ideal for testing core server infrastructure
  - combined with the individual tests and gentest it allows the server side CIFS parsing to be tested in isolation
gentest

- gentest is the 'big gun' CIFS test program that I have wanted to build for many years. Basic features include:
  - dual server, dual instance testing
  - randomised, broad coverage request generation
  - automatic backtracking for finding minimal request subset
  - can cover all fields of all requests
  - full async oplock testing
Dual Server Testing

- The basis of gentest is 'dual server testing', the same basic technique used in the 'locktest' program from earlier versions of Samba:
  - The test program establishes two connections to each of two servers
  - Random requests are then generated, with identical requests sent to the two servers
  - At each step gentest compares every field of every response between the two servers
  - When a response differs gentest uses backtracking to find the minimal subset of the requests sent so far that generates a difference in response
Request Generation

- Request generation is based on the concept of a 'generator' function for each request in CIFS.
- The generator for a CIFS request calls into a library of 'field generators' that produce constrained random values for each type of field in the protocol.
- Field generators include things like gen_timeout(), gen_io_count(), gen_fnum(), gen_fname() etc.
Field Generation

• The generators for individual fields are heavily biased towards interesting values, while allowing for arbitrary values in most cases:
  
  • `gen_fnum()` will most of the time generate an open file handle (if one exists), but will sometimes generate an invalid handle
  
  • Some fields (like IO counts) are tightly constrained to prevent filling of disks
  
  • Flags fields are heavily biased towards valid sets of flags, but have a small chance of generating arbitrary sets of bits
Backtracking

- When a difference is discovered between the two servers gentest goes into 'analyze' mode, using a backtracking technique to find the minimal subset of requests that produce a difference:
  - successively smaller chunks of the request streams are blocked out
  - If a difference is still reported when a chunk is blocked out then that chunk is not needed and can be discarded
  - reconnects to the servers and wipes all files at each pass
  - The final pattern of requests can be replayed for analysis with a network sniffer
Oplock testing

- It has previously proved very difficult to write a good oplock test program. With gentest it is quite easy:
  - The field generators often randomly produce open requests with oplock flags set
  - At each request oplock break requests are checked for, and compared between the two servers
  - When an oplock break is received gentest chooses at random whether the break will be acknowledged or the file closed
Ignore Patterns

- Some portions of the protocol are expected to vary between servers, and some portions are known to be unimplemented by some servers.
- To cope with this, gentest allows for a set of 'ignore patterns'. These come in several forms:
  - patterns matching types of requests that should not be generated at all
  - patterns matching "don't care" fields that are allowed to differ
  - patterns matching generated data and information levels that tells gentest not to generate those requests
Standard Ignore Patterns

- I have found the following set of ignore patterns to be necessary for operation between two W2K3 servers:

  - all_info.out.fname
  - compression_info.out.*_shift
  - internal_information.out.*
gentest problems

- There are a number of limitations and problems with the gentest approach to testing:
  - it can be very slow, especially with servers that response slowly to certain failed operations
  - no multiple void testing yet
  - tests are avoided that would kill the connection
  - some filesystem properties (like sticky create times) can cause problems

- The biggest problem is that before gentest is useful for testing against other servers you have to be very close in behaviour
Major uses for gentest

• gentest can be used for quite a wide range of purposes:
  
  • the obvious use is to compare behaviour to a reference server
  • very useful for comparing two versions of your server to see what you broke
  • allows checking for internal consistancy of your server by running against two shares on the same server. This finds intermittent bugs and uninitialised values quickly
  • gives very wide code coverage, which makes it ideal to run in combination with memory testers like valgrind
Questions?

Legal statement:
This work represents the views of the authors, and does not necessarily represent the views of IBM