The guy
Like magic...
lots of hard work
mostly other peoples'

VFS layer
Kerberos & delegated credential support
Samba generally
Samba4 WAFS Discourse

- The problem
- The solution
- Implementation
- All the other bits
The problem

- CIFS performs poorly over the internet
- Measurable in terms of:
  - # seconds to save a file
  - % idle bandwidth wasted
  - Low ROI on bandwidth investment
  - Users waste of even more time while waiting
  - Cost of WAIFS solutions
The Causes

- Low bandwidth
- High latency
- Chattiness

Nigella Lawson chatting instead of book signing. I'll bet there's a long queue moving slowly.
Bandwidth as a cause

Special antique low-bandwidth pen

- LAN link speeds: 10 - 1000 Mb/s
  - File transfer speeds 80 – 400 Mb/s
- WAN link speeds: 300 Kb/s – 10 Mb/s
  - Up to 30 times slower
Bandwidth as a cause

- robot pen from coolest-gadgets.com
  - Will it help speed things up?
Latency as a cause

- LAN latency 2ms
  - Theoretical 500 requests per second
- WAN latency 50 – 100ms
  - Theoretical 10 – 20 requests per second
  - A process needing 500 requests takes 50 seconds
- WAN at least 25 – 50 times slower than LAN
- Taking message size into account means even slower due to lower bandwidth
Chattiness – the worst of both

- Most applications are synchronous
  - CIFS client waits for file to open before reading
  - Waits for read to finish before reading more
  - Repeated requests for the same meta-data
  - The problem can't be solved with a bigger pipe
- Chattiness / Poor CIFS pipelining
  - latency adds up
  - Under-utilisation of available bandwidth
Chattiness – the maths

- **Request time**
  - \(\text{SIZE}_{\text{request}} / \text{BW}_{\text{upstream}} + \text{LATENCY}_{\text{upstream}}\)

- **Response time**
  - \(\text{SIZE}_{\text{response}} / \text{BW}_{\text{downstream}} + \text{LATENCY}_{\text{downstream}}\)

- **Total**
  - \(\text{TIME}_{\text{request}} + \text{TIME}_{\text{response}} + \text{LATENCY}_{\text{server}}\)
Chattiness - examples

<table>
<thead>
<tr>
<th>Request / Response</th>
<th>Size / bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count: 1000</td>
<td>Request: 64</td>
</tr>
<tr>
<td></td>
<td>Response: 4100</td>
</tr>
</tbody>
</table>

Combined total request response time in seconds

<table>
<thead>
<tr>
<th>RTT/mS</th>
<th>Symmetric Link Bandwidth Kbit/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>102400</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

At 50ms latency a bandwidth increase of 2,000% decreases load time to about 50%

More bandwidth doesn't help much!
Chattiness – the graphs
The Solution

- Remove harms of chattiness
  - Of course!
- Reduce latency with read-ahead
- Reduce bandwidth demands with compression
  - Also reducing link contention
An opportunity

A device at each site to extend CIFS protocol
Read-ahead

- Done already, Jeeves?
- I trust that sir is satisfied?
Read-ahead

- Abolish RTT latency
- Response processed before related request
- Read ahead by $\text{RTT} \times \text{bandwidth}$ to get link speed
Read-ahead

- File-read using full available bandwidth
- Latency still problem for folder browsing
- In early tests, read-ahead on a 600Kb/s ~50ms link reduced the time to read a file by 25%
With read-ahead the new apparent LAN-side latency is effectively:

\[
\text{SIZE}_{\text{request}} / \text{BW}_{\text{upstream}} - \text{LATENCY}_{\text{lan}}
\]

- 500Kb/s =~ 500bits per millisecond
  - 4Kbyte response takes 65ms
  - With LAN latency of 2ms effective LAN latency is 63ms at LAN bandwidth
## Read-ahead vs Latency

<table>
<thead>
<tr>
<th>Request / Response</th>
<th>Size / bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count: 1000</td>
<td>Request: 64</td>
</tr>
<tr>
<td></td>
<td>Response: 4100</td>
</tr>
</tbody>
</table>

**Combined total request response time in seconds**

<table>
<thead>
<tr>
<th>RTT/mS</th>
<th>Symmetric Link Bandwidth Kbit/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>102400</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

Reducing latency to LAN levels makes a BIG difference even at moderate bandwidth
Compression

- Increase effective bandwidth
- Zlib often gives 50% compression rates
- Custom dictionaries can give better compression
Compression and read-ahead make great savings of 67% off Symmetric Link Bandwidth Kbit/s.

### Request / Response vs. Size / bytes

<table>
<thead>
<tr>
<th>Count: 1000</th>
<th>Request: 64</th>
<th>Response: 4100</th>
</tr>
</thead>
</table>

### Combined total request response time in seconds

<table>
<thead>
<tr>
<th>RTT/mS</th>
<th>102400</th>
<th>10240</th>
<th>2048</th>
<th>1024</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>17</td>
<td>34</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>18</td>
<td>35</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>38</td>
<td>70</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>23</td>
<td>36</td>
<td>53</td>
<td>85</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>53</td>
<td>66</td>
<td>83</td>
<td>115</td>
</tr>
</tbody>
</table>

Compressed and read-ahead make great savings of 67% off.
LAN Speeds over the WAN

1. If the file is previously cached
2. If the cache can be cheaply validated on open
   - Then READ operations are at
     - LAN speeds
     - LAN latency
   - Validation-on-open strategy not simple
     - Avoid processing unwanted cache
     - Avoid extra latency on open
Caching

- Solves latency and bandwidth issues entirely
- Non-validated cache can help compression
  - MD5 to validate cache
  - Use cache contents as a dictionary
  - Unroll rsync / rdiff
  - Dynamic dictionary management
## Caching-Compression

### Request / Response

| Count: 1000 | Request: 64 | Response: 4100 |

### Combined total request response time in seconds

<table>
<thead>
<tr>
<th>RTT/mS</th>
<th>Symmetric Link Bandwidth Kbit/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>102400</td>
</tr>
<tr>
<td>Read from cache</td>
<td>1</td>
</tr>
<tr>
<td>Compress from cache</td>
<td></td>
</tr>
<tr>
<td>Zlib compress</td>
<td></td>
</tr>
<tr>
<td>Read-ahead</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

Caching and compression and read-ahead make great savings

95% off
Cache Coherency

- A nasty headache, see Coda, Intermezzo, AFS
- Nobody wants to resolve conflicts anyway
- Oplocks and notifications to the rescue
- Cache validated while an oplock is held stays valid – well worth reading ahead in this case!
- Metadata can be cached when folder change notifications are registered – no more repeats
- All other requests to the server – but optimized
Other requirements

- Maintain user identity
  - ACL's
  - Permissions
  - Ownerships
  - Quotas
- Maintain locking
- Cache coherency
Samba4 platform benefits

- Samba4 maintains CIFS semantics
- Samba4 already has a CIFS proxy
- Samba4 integrates with AD trust system
- Kerberos supports delegated credentials
- Trust of proxies can be managed standard AD management tools or set when provisioning
- Proxies can read-ahead using users credentials
- There's a load of brains working on it already
Implementing the solution

- Based on Samba4 proxy module
- Keep caching engine separate
- All reads requests consult a cache and validate from server where required
- All read responses stored in a cache
- Do writes hit the cache after completion?
  - What if a read comes in the meantime?
- Meta data can be cached too
- oplock breaks and notifications invalidate cache
Deployment and Provisioning

- Directly access shares from the proxy
  - Maybe DFS referrals could pick nearest proxy?
Implementing the solution

It all works together so well in theory
Samba4 infrastructure

- Proof of concept very simple
- It's all there, it looked so easy right away
- Read-ahead and zlib easy to add to cifs_proxy
- Code was well structured so I didn't have to get to grips with all of it.
  - At first
- My first bug: oplock handling in cifs_proxy
  - Took 3 months to get patched - exciting
Multiple proxies

- Extend share definition to match called name
  ```
  \proxy-alias\share
  ```

- Use additional SPN's for each proxied server
  ```
  \local-accounts\secret
  server=accounts.realm.net
  share=top_secret$
  \local-games
  server=games.realm.net
  [*]
  server=main.realm.net
  ```
Implementing the solution
New opcode? New nttrans?

New ntioclt – 0xACE

Iocltl gives the option of implementing natively in windows server, so I'm told

Use the dcerpc NDR code to marshall RPC
  - transport over ntiocltl
    - which transports over nttrans
      - Which transports over SMB
        - Which transports over...

 Lots of copying anyway!
How reads work

- Look for a pending read and attach to the callback handler as a read-fragment
- Read from cache and issue optimized reads
- Repeat until all mincount is satisfied
- Callback handlers re-assemble read buffer
- Make sure attached read-fragment isn't free'd by original caller before we've finished with it.
- Now I've got to stop excess simultaneous reads!
Problems

- Client negotiates large write with proxy
  Server negotiates small writes with proxy
  Likewise for reads
    - Simple request proxying won't work
- Requires fragmenting reads and writes and collating results.
- What happens if a middle request fails?
- What happens if the server thinks we queued too many simultaneous fragments?
Attaching to existing requests

- `talloc_referencing` multiple handlers sticking onto each-other's memory
- Changed whole async callback mechanism
- Callback chains to reverse map incoming responses – `ntioctl`, `nttrans` etc
- New meta-infrastructure that selects between proxy-proxy comms or proxy-server
- Will change again to avoid need for references
New callback mechanism

- Related calls typically use same smb_* struct
  Not any more!
- Related calls now have different encapsulations
  - smb_read as standard
  - proxy_read uses NDR / NTIOCL / NTTRANS
  - The encapsulator queue's a de-encapsulator
  - So the caller gets an unpacked struct
  - The first callback calls smb_receive()
- Sync or async have same handlers!
Simple cache

- Simple file-based linear extents
  - Length
  - Validated length
  - Pending length
- No holes in cache
- Cache key is user + server + share + path
- Delete random cache content when full
Better Cache management

- Ideally fragments should be selected based on reimen polynomials
  - rolling_checksum % frag_size_key == 0
- This could also be the fragment key
  - to avoid the birthday problem, we probably want to negotiate a unique key between all caches
- Per-user file cache becomes index of fragments
- Duplicate data is stored only once
- Delete low value content when full
The pain of the blessed Samba

- nttrans and ioctl had various bugs
  - multi-packet requests/responses
  - >64K requests responses
  - Is >64K ntioctl allowed? Dunno
- I wasn't wanting to have to fix these!
  - Forced acquaintance with code base and tools
- But at least I got 0xACE is my ntioctl
- Hope no-one else picks such a cool function id
They might, it's so cool; agghhhh
Pain of rejection

- No-one likes DLIST_FIND

```c
#define DLIST_FIND(list, result, test) \
do { \\
    for (((result) = (list); \\
        (result) && !(test); \\
        (result) = (result)->next); \\
} while (0)

DLIST_FIND(thingy->list; item; item->id==id);
```
The joy of acceptance

- Poor-mans debug_ctx()
  Uses a DEBUG() scoped variable instead of a static variable.
  - Compatible with samba3 debug_ctx()
  - Wastes a lot of memory
  - Works without DEBUG being thread-safe
The joy of acceptance

- Fix large request fixups in receive.c
  - Were taking wrong affect on non AND_X requests
  - Allows >64K nttrans to be handled
- Fix OP_LOCK breaks on vfs_proxy
- smb_abort macro for talloc_get_type_abort
  - allows per-caller abort mechanism
- talloc_memdup_type also clones struct name
It works!

- Testers like it – saves time
  - I'm not lying
  - No longer feel let down or hurt by performance
  - Or give up and play around while waiting
  - Goodbye!