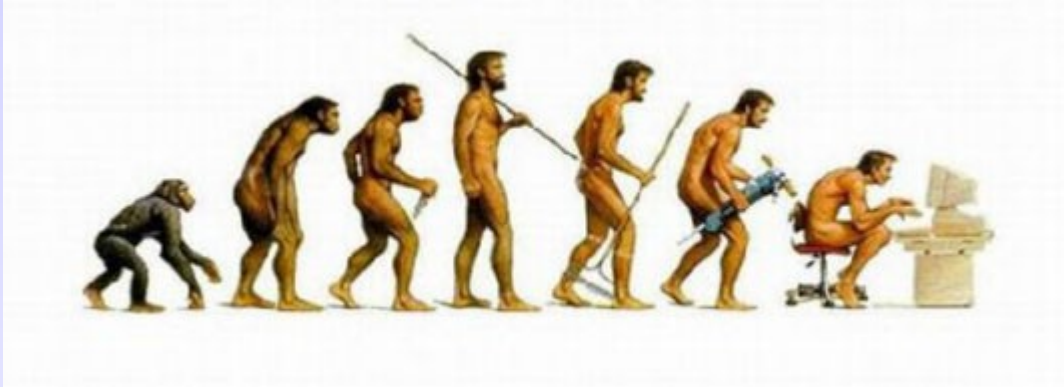
SAMBA Opening Windows to a Wider World

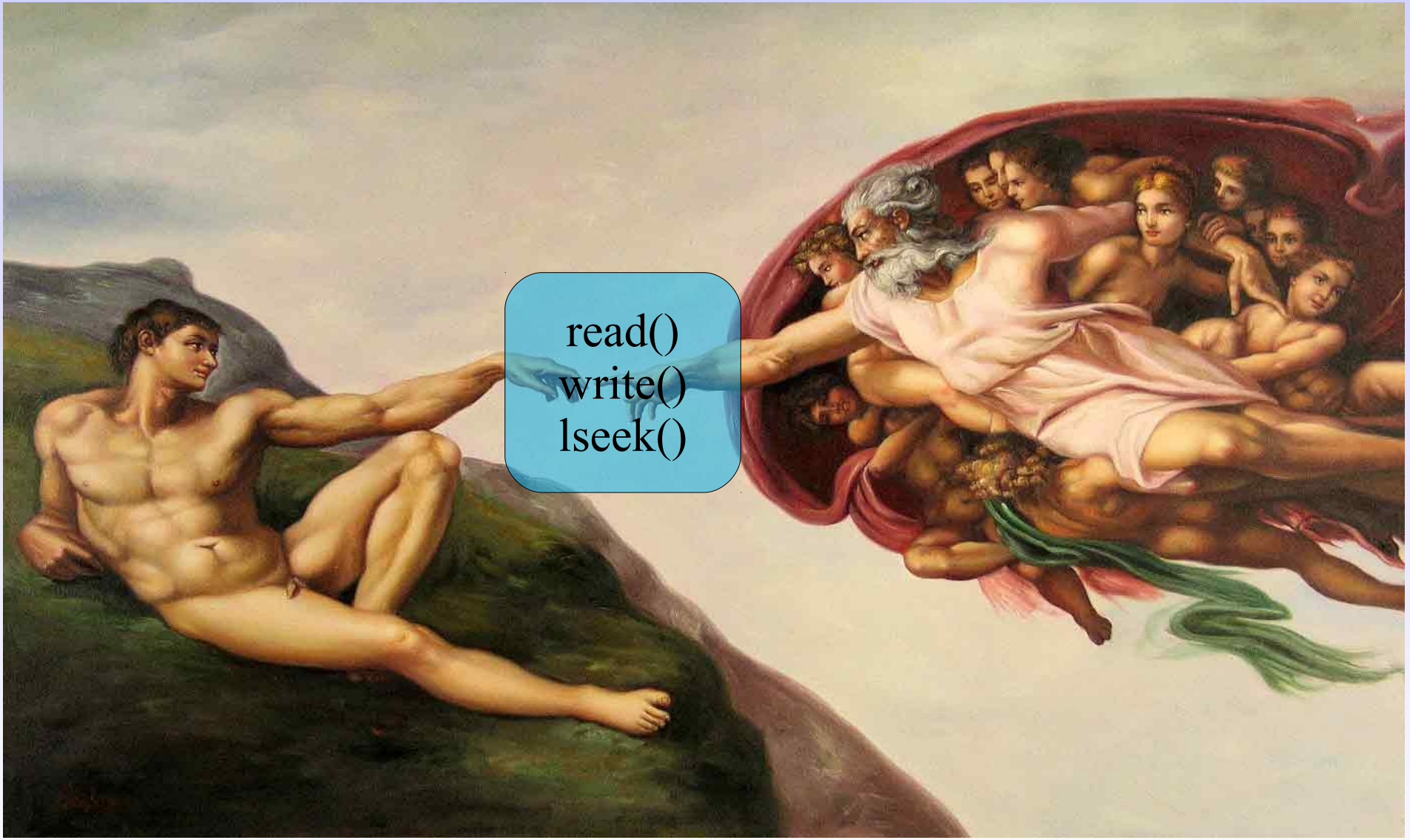
The Evolution of I/O in Samba



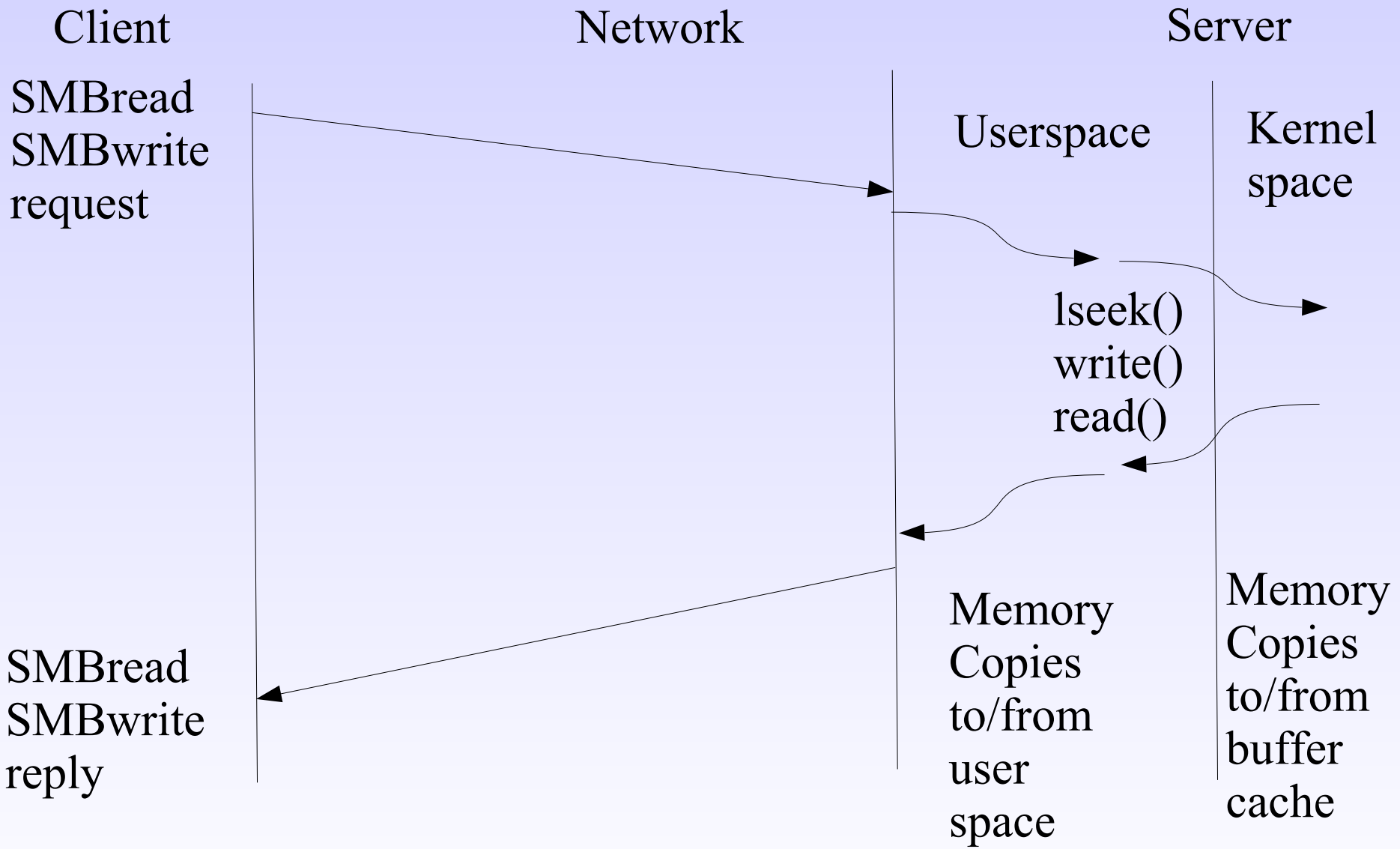
SAMBA

Jeremy Allison
Samba Team
jra@samba.org

In the beginning..



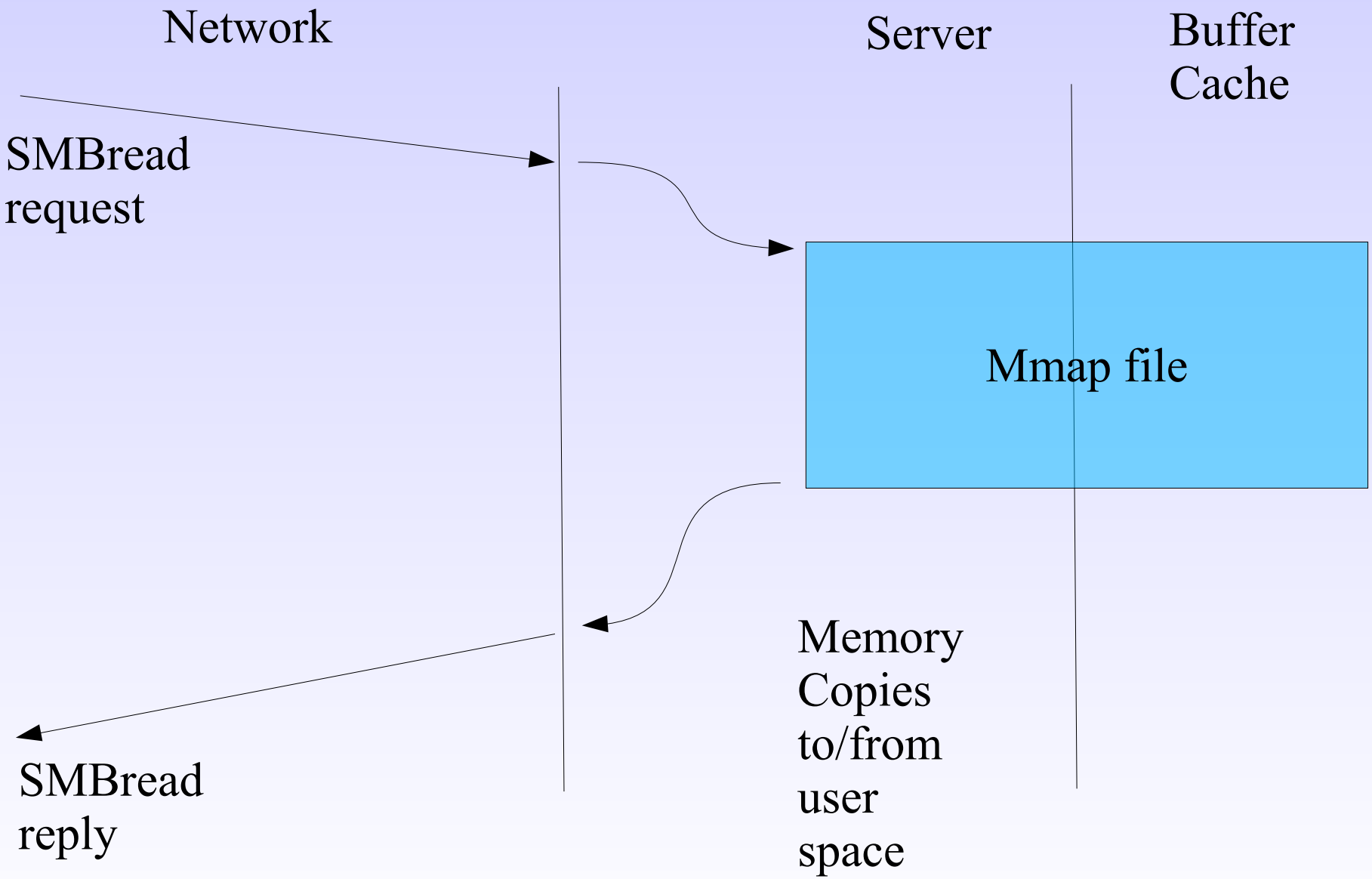
The initial I/O path



"A simpler server for a more simple age of clients"

- Old (DOS/Windows 3.x/Windows 9.x) clients did not issue multiple simultaneous read/write requests on the wire.
- Maximum read/write requests were 64k, most clients used much smaller sizes (max. of 60k from Windows clients).
- Preceeded the pread/pwrite system calls, so lseek()/ESPIPE errors were ignored to allow communication with UNIX fifos.

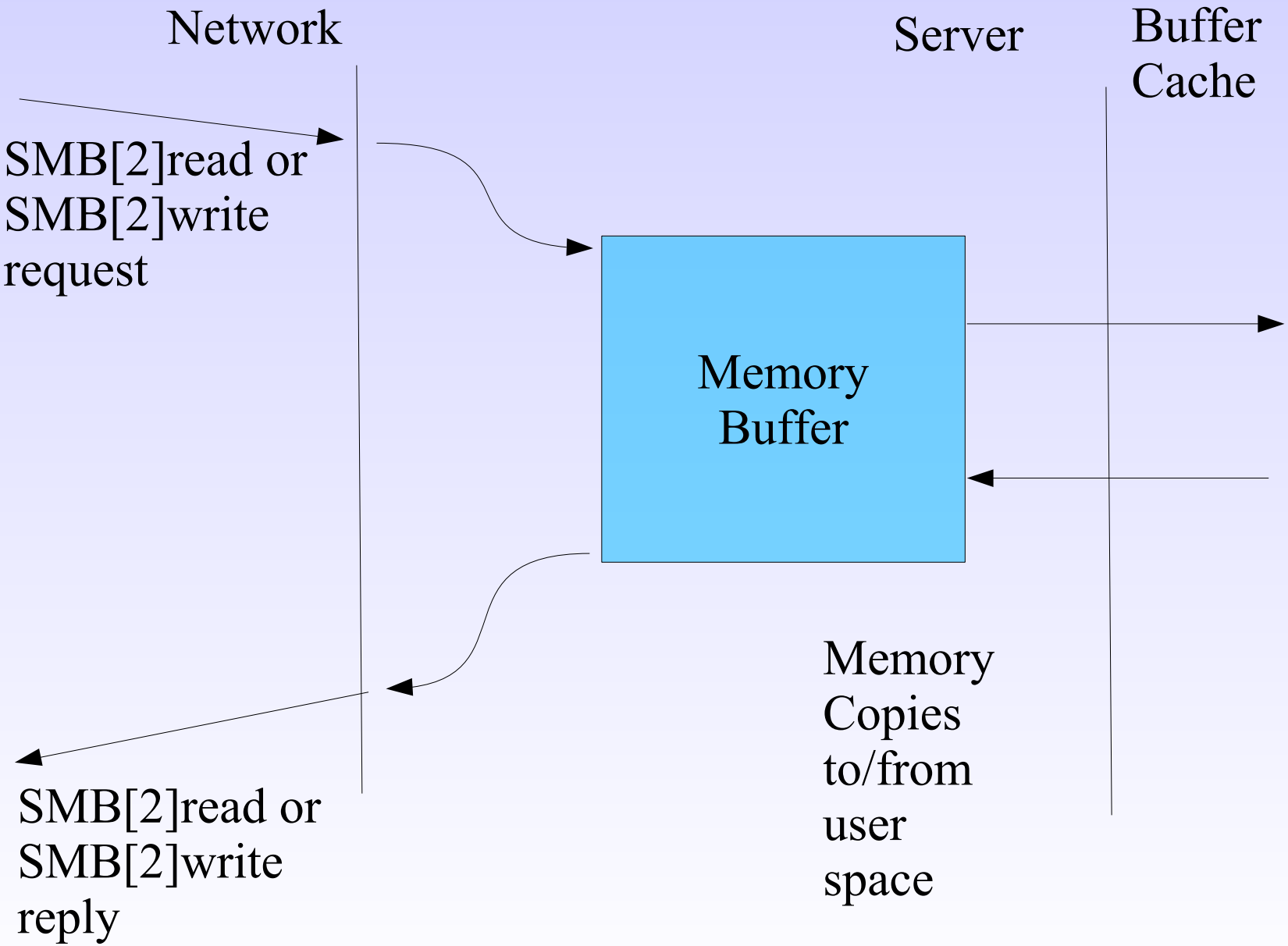
First read optimization



A read optimization

- Early work – Samba 1.x days.
- On opening a read-only file we attempted to mmap() the entire file.
- Direct copy from the buffer cache to the outgoing packet.
  - Still did two copies, but no lseek()/read() system call pair needed.
  - File truncation causes segfaults. Oops.
- “read prediction” was added to read one SMBread segments into memory on read-only file open.
  - Removed once oplocks were added into

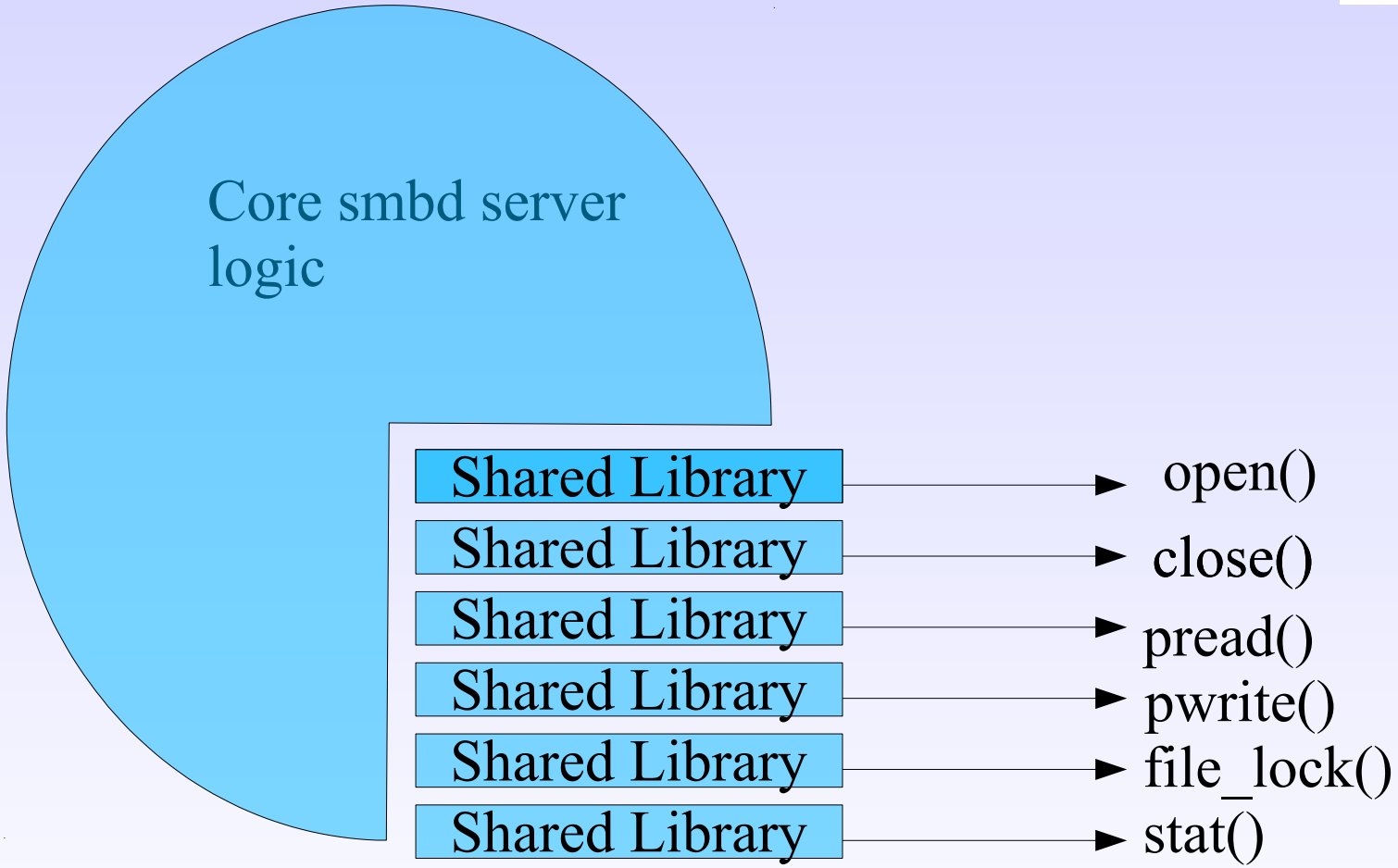
"write cache" optimization



“write cache” optimization

- Only valid on oplocked files, allowes server to buffer up small writes until a RAID stripe size is reached.
  - Improved performance on SGI XFS filesystem.
- Reads served out of userspace cache if within range.
  - Complex logic needed to cope with read or write positions partially overlapping with the cache.
  - More complex logic to keep cache coheret.
- Dynamic movement of the cache to cover “hotspots” in client activity.

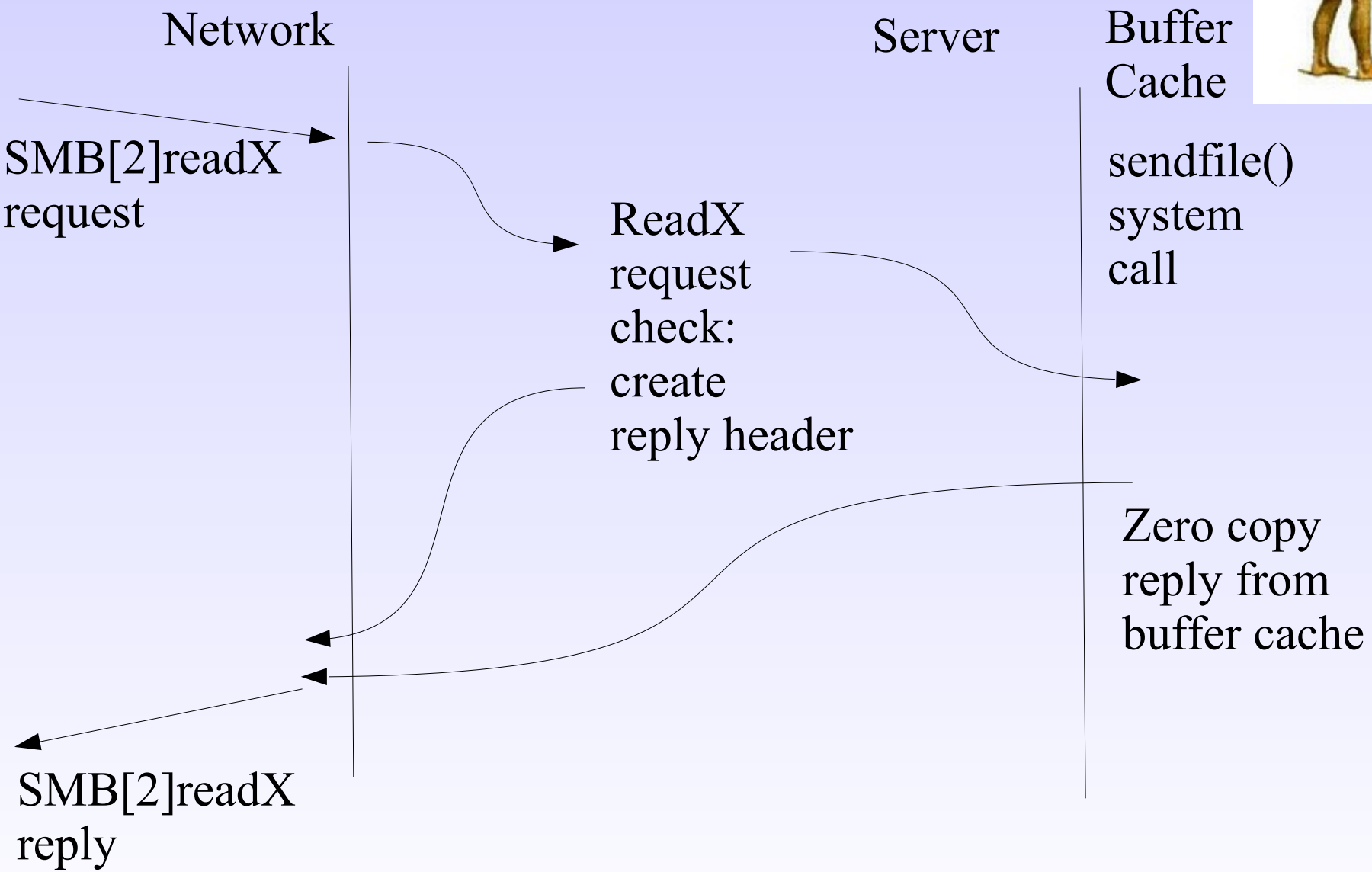
First tool use – Samba VFS !



The Samba VFS

- A completely configurable, easily programmed read/write path allowed an explosion of creativity inside Samba.
  - Modules created to allow read-ahead on files, preallocate space, provide a recycle bin for deleted files, interface with anti-virus and many other options.
- Every code path that touches the disk can be trivially intercepted.
  - Stackable design means only implement the calls you need.
  - Freed up vendors and OEMs and users depending on Samba from having to patch the source code.
  - Interface is currently kept stable within a

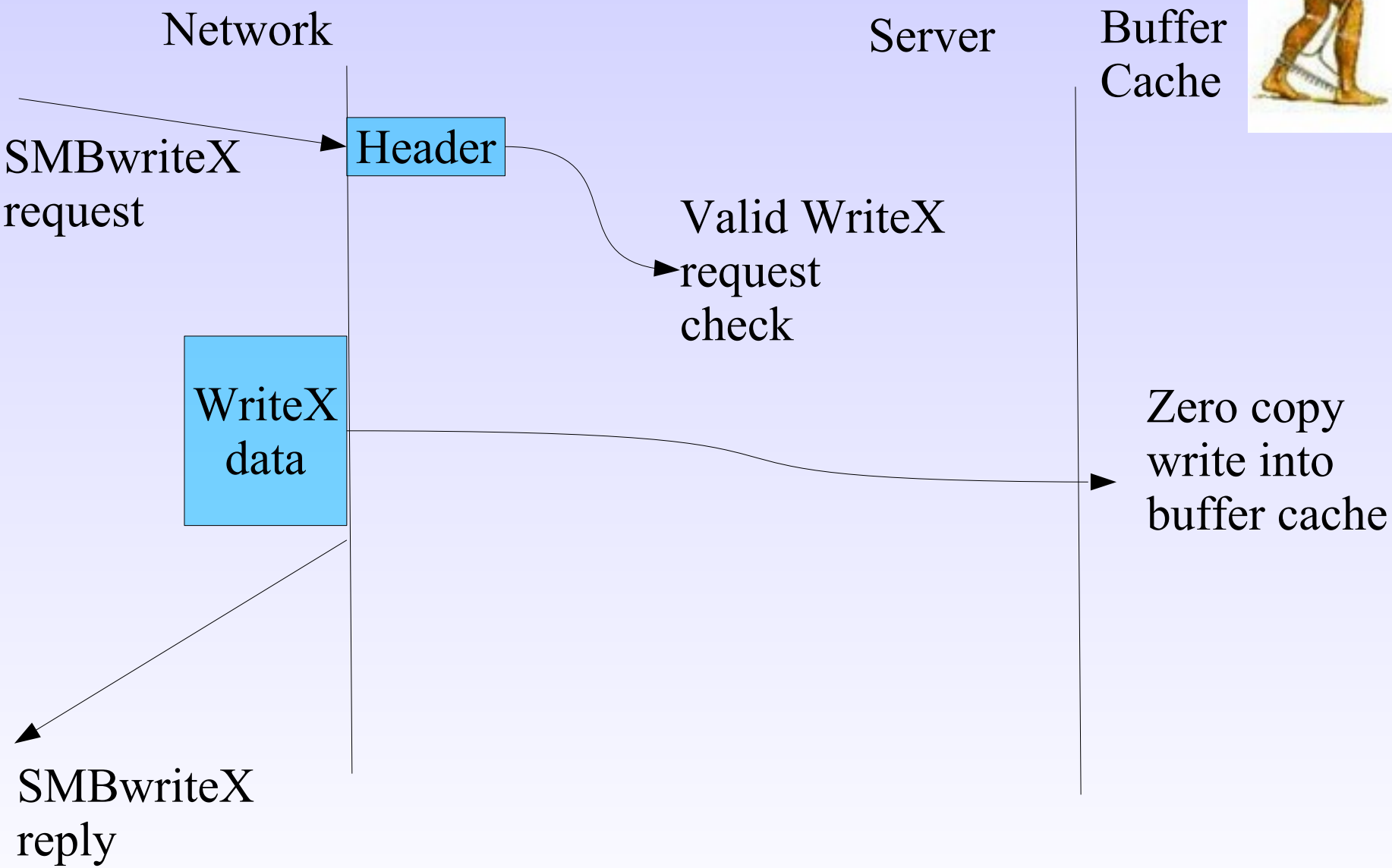
“zero copy” - sendfile()



“zero copy” - sendfile()

- Only done on oplocked files with a SMBreadX or SMB2 READ call.
  - Does not work for signed requests.
- Header is created in smbd layer, then sent back to client.
  - Data payload zero-copied from the buffer cache via a sendfile() call.
  - Fun with different sendfile() implementations.
- Can really improve performance.
  - Usually around 10%, depending on workload, disks, network, the way the wind is blowing etc.

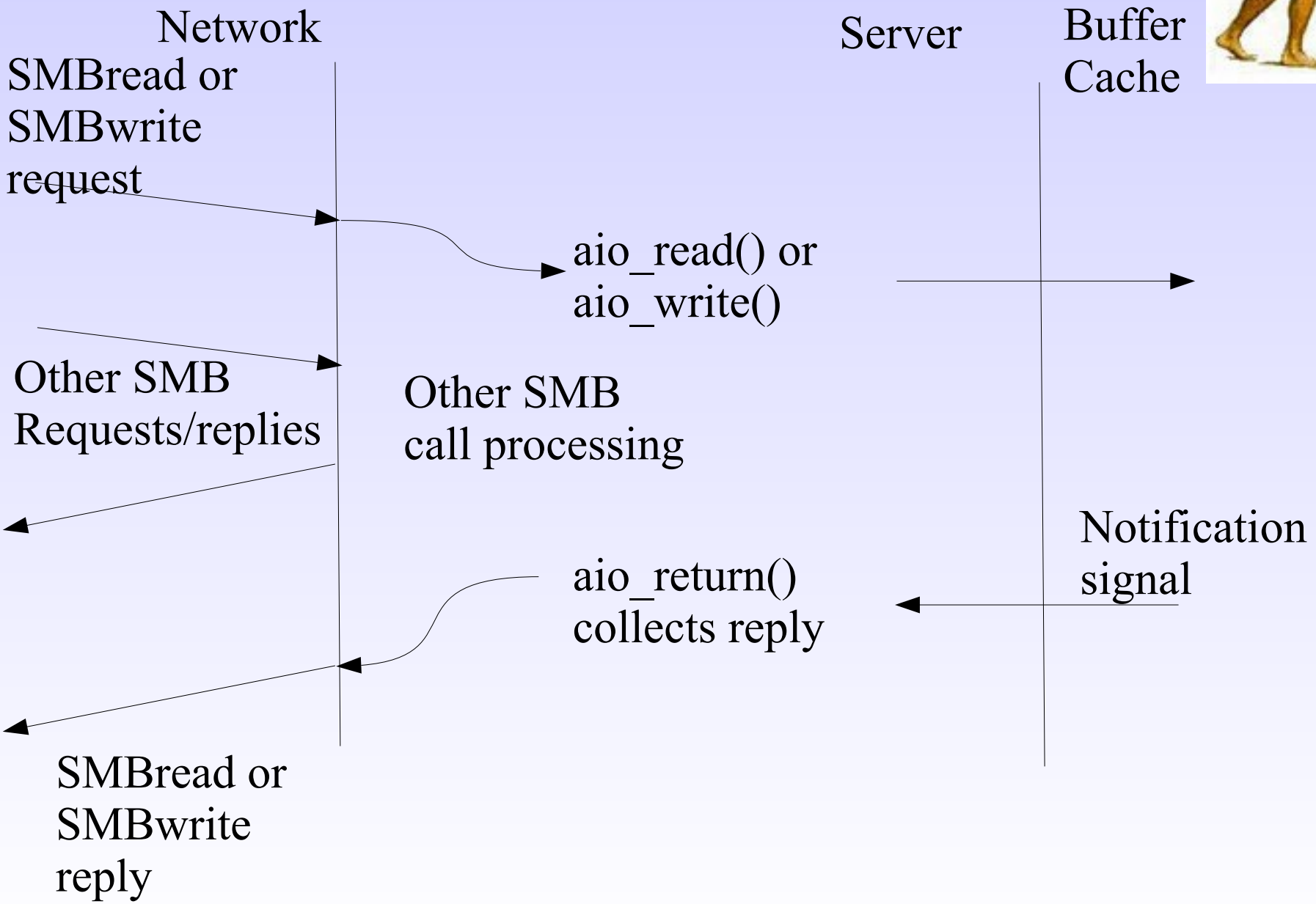
“zero copy” - recvfile()



“zero copy” - recvfile()

- NOT a standard POSIX/Unix/Linux system call.
  - Isilon (now EMC) implemented the initial code paths in Samba.
  - Custom *BSD code implemented this in their kernel.
  - Doesn't work with signed requests.
- Problems on other systems is sendfile() is not symmetric.
  - Can't do zero copy from network to buffer cache.
- Linux tried to solve this by using splice() call.
  - Idea is to use a pipe handle as a way to

Asynchronous at last – POSIX AIO



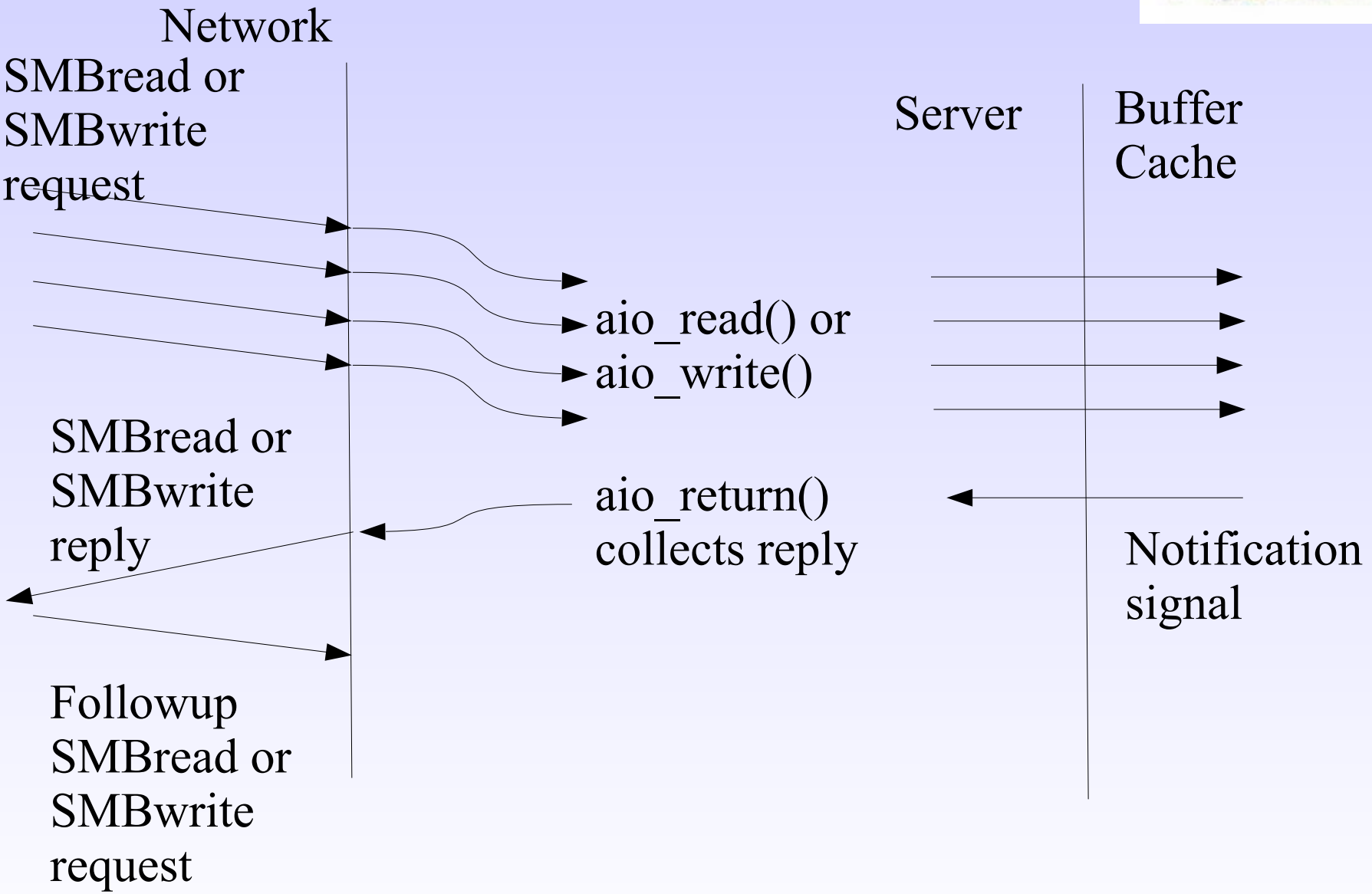
Asynchronous at last – POSIX AIO

- Horribly complex interface. Dependent on POSIX-RT signals for notification.
  - Linux kernel refused to implement, glibc implemented in userspace using pthreads.
  - Linux kernel implements a Linux-specific API.
- Semantics of operations like “cancel” are unclear for a file server.
  - Both SMB1 and SMB2 can cancel operations. SMB1 never does (at least for I/O). Unclear about SMB2.
- Older clients (pre-Windows 7) only issue one outstanding I/O per thread. Few programs

The dark ages - glibc AIO

- glibc implementation has two problems.
  - As smbd changes effective uid, it can get into a situation where it has no permissions to deliver the notification signal.
    - Smbd solved this via a horrible internal hack.
  - glibc forces multiple outstanding I/O requests on the same file descriptor to be synchronous.
    - This is **NEVER** what you want.
    - Path fixing this was not accepted.
- Other systems (e.g. Solaris) don't have these issues.

Driving AIO with libsmb



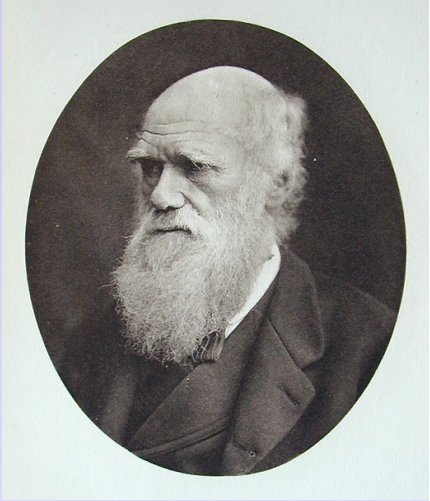
Driving AIO with libsmb

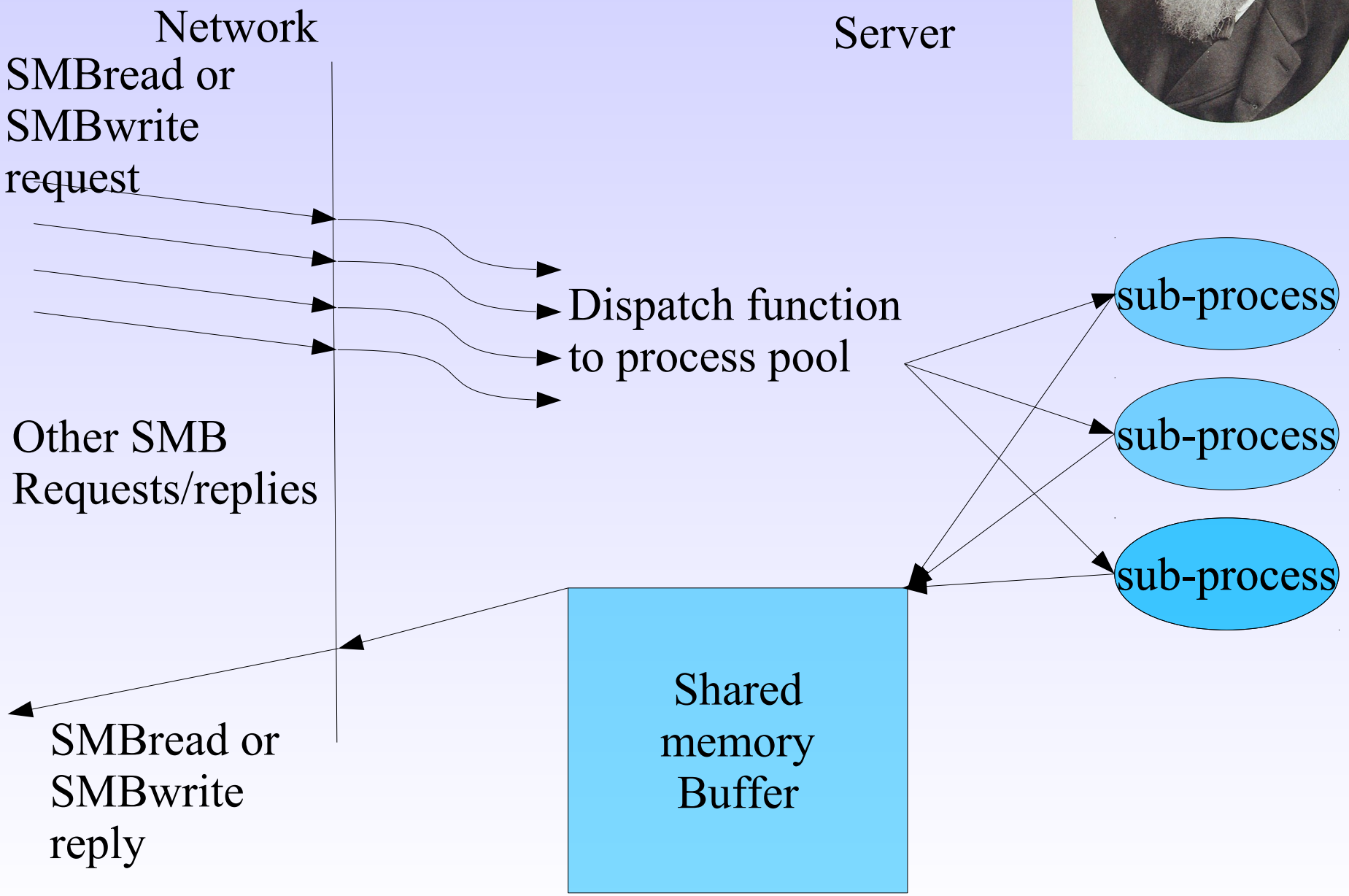
- First implemented by Volker Lendecke – who proved you could saturate gigabit wire with SMB1 traffic.
  - Idea is to pipeline multiple outstanding I/O requests up to the server declared max mux value.
  - As one completes at the server, and is received by the client, the client dispatches the next one in the queue, keeping the pipeline full.
- Outstanding performance improvements.
  - One of the initial touted benefits of SMB2.
  - Windows Vista onwards implements a

SMB2 and the new Windows redirectors

- Windows Vista onwards now includes an SMB1 redirector that will do pipelining of I/O requests (or maybe it's an improved explorer shell).
- SMB2 is designed to do multiple outstanding I/O, and can cope with quite large read/write sizes.
  - Although SMB2.0 is still restricted to 64k.
- Supporting AIO is a requirement for any servers servicing these systems to get good performance.
  - Depending on disk spindles available, CPU etc. etc.

Enter Charles Darwin – Volker and vfs_aio_fork()

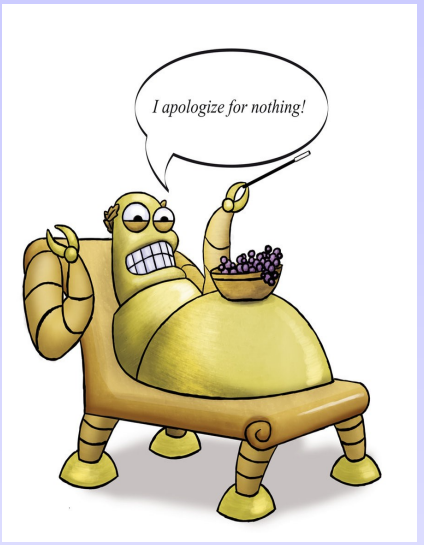


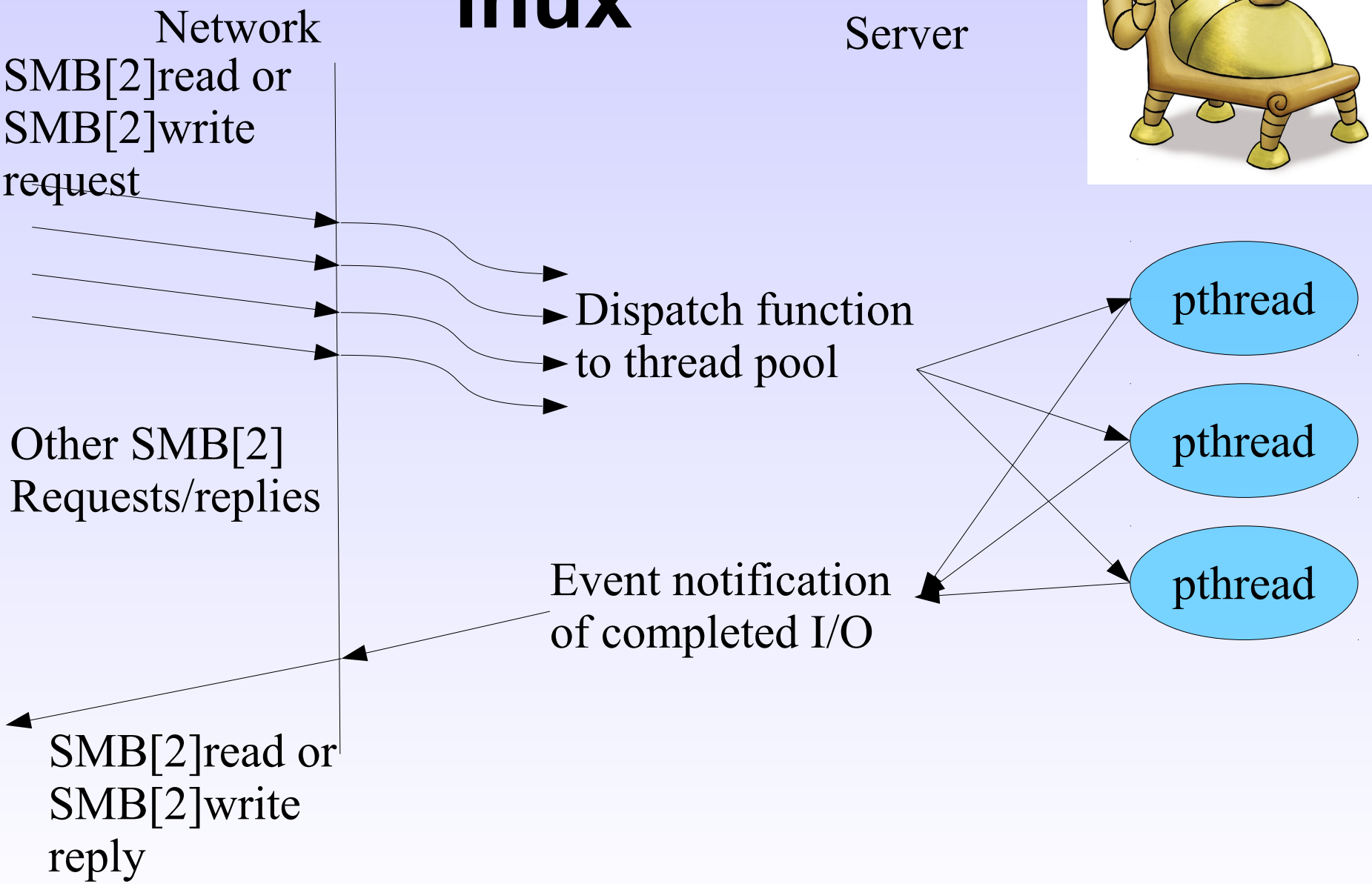


vfs_aio_fork()

- vfs_aio_fork is a very clever piece of code that hides a process worker pool behind the POSIX AIO interface.
  - Uses pipes not signals for I/O completion notification.
  - Passes file descriptor to operate on to child process via sendmsg() to create a new fd in the child.
- Module is resource intensive.
  - Only really efficient on systems like Linux where creating processes is a relatively lightweight activity.
  - Does introduce another memcpy, from the

The Renaissance – vfs_aio_pthread/vfs_aio_linux





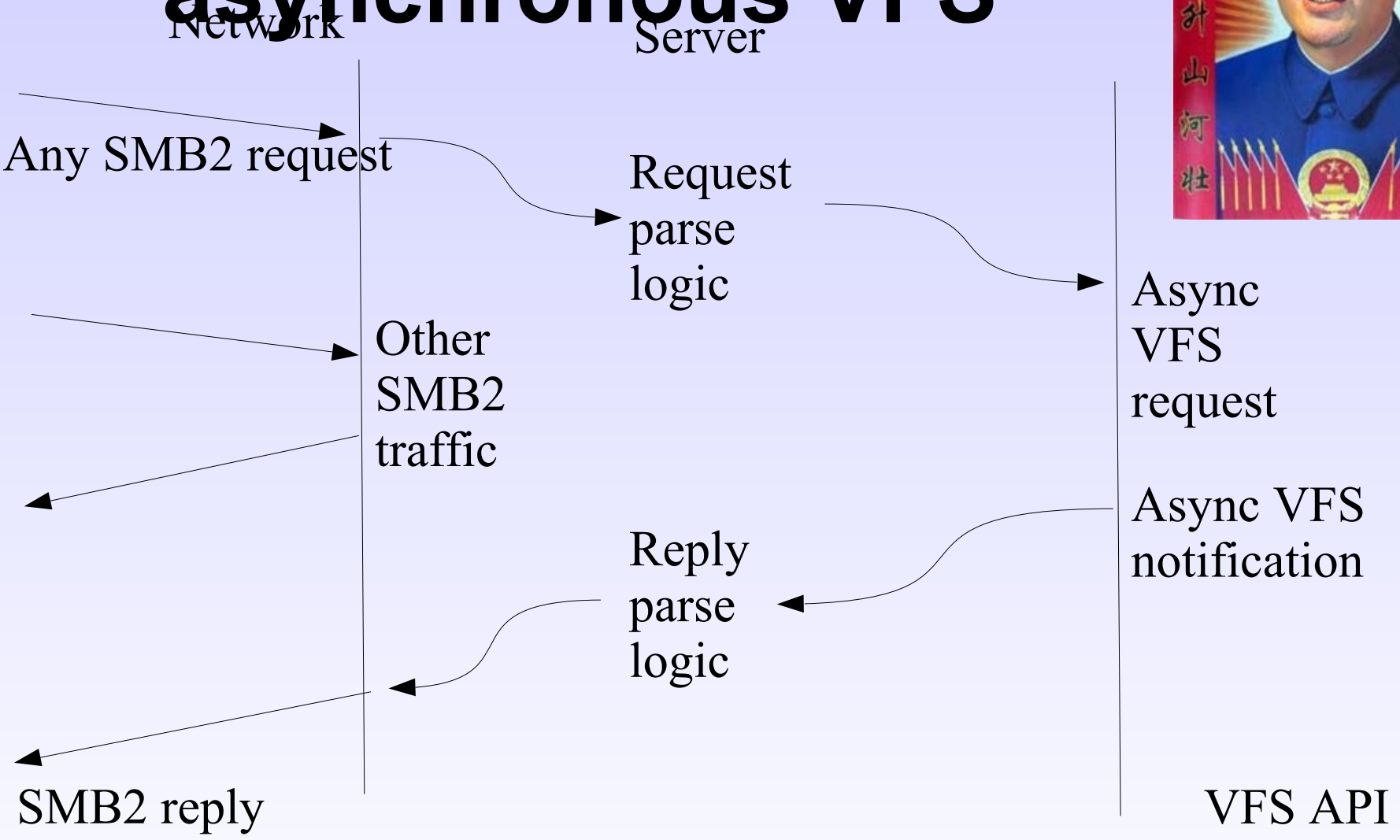
vfs_aio_pthread

- Module structure is shamelessly stolen from Volker's vfs_aio_fork code.
- Instead of processes, vfs_aio_pthread uses a pthread pool library (also written by Volker) to provide the back end asynchronous operation.
  - Maximum pthreads in the process pool is set quite large (128), as performance depends on keeping as many disk spindles busy as possible.
  - Idle threads are destroyed after 1 second, making this module capable of coping with large bursts of I/O requests, but keeping resource use low.

vfs_aio_linux

- Module structure is shamelessly stolen vfs_aio_pthread, which inherits from Volker's vfs_aio_fork code.
- Instead of threads, vfs_aio_linux uses the native Linux AIO system calls to provide back end asynchronous operation.
  - Theoretically (module still under test) this should be the most resource efficient way of providing asynchronous operation. Everything done asynchronously at the system call level.

Let a thousand flowers bloom – tevent and asynchronous VFS



tevent and the asynchronous VFS

- Now we have a boilerplate asynchronous programming API/state machine – tevent, we can start splitting up the VFS into XXX_send() / XXX_recv() call pairs that complete asynchronously.
  - This will make a “simple” VFS module more complex, but providing multiple examples should help.
  - Core logic for much of the file server will have to be refactored – not a complete rewrite, more a code reorganization (but we have to do this anyway to add more SMB2+ features).
  - Eventually the VFS won't resemble POSIX

Where are the numbers ?

- This is the slide with graphs showing how turning on AIO modules doubles Samba performance, uses less resources and generally makes life better.
  - Unfortunately this is not that slide.
- Completely depends on available system resources.
  - An example. Testing at an OEM who uses a low-powered CPU, adding AIO for write added 10% to performance, adding AIO for read lost 10% performance.
    - Turns out zero-copy sendfile() is much more important on that box.
- Performance is very dependent on disk

Questions and Comments ?

Email: jra@samba.org