

The past, present and future of Samba messaging

SambaXP 2015 Göttingen

Volker Lendecke

Samba Team / SerNet

2015-04-20

Parallelism based on messages

- ▶ Wikipedia: An actor is a computational entity that, in response to a message it receives, can concurrently:
 - ▶ send a finite number of messages to other actors;
 - ▶ create a finite number of new actors;
 - ▶ designate the behavior to be used for the next message it receives.
 - ▶ There is no assumed sequence to the above actions and they could be carried out in parallel.
- ▶ Many languages embed message passing these days
 - ▶ Erlang, Google go, Akka for JVM
- ▶ Some problems with threaded programming vanish with message-based parallelism
 - ▶ Others of course appear :-)

Why Inter *Process* Communication?

- ▶ Erlang, Go and others provide intra-OS-process messages
 - ▶ Threading based on OS threads plus VM threads
 - ▶ All in one memory space
- ▶ Dependency on the virtual machine
- ▶ Distribution to processes possibly more reliable
 - ▶ A crash in one OS process has less effect on others
- ▶ Better NUMA affinity
 - ▶ Isolated memory maps
- ▶ Process architecture has helped Samba to go clustered
- ▶ Interoperability to other languages

Unix Signals

- ▶ One of the oldest Unix IPC mechanism
 - ▶ Asynchronous delivery
 - ▶ Almost no information transferred
- ▶ Difficult to use together with threads
 - ▶ If possible, receive signals in one thread only
- ▶ When used, not much can be done in a signal handler
 - ▶ Posix lists only 135 functions as aync signal safe
- ▶ Watch out for EINTR result of most syscalls

Shared Memory

- ▶ Same memory pages visible in multiple processes
 - ▶ Blurs the distinction between threads and processes
- ▶ Fastest IPC mechanism
 - ▶ No kernel involvement for data transfer
 - ▶ Good for transferring mass data
- ▶ Limited use for message passing
 - ▶ No good signalling mechanism
- ▶ Needs coordinated access
 - ▶ Locks, Mutexes

fcntl locks

- ▶ Advisory byte range locks
 - ▶ Shared (F_RDLCK) or exclusive (F_WRLCK) locks
 - ▶ Just an IPC mechanism: fcntl locks do not block read/write
- ▶ Weird semantics regarding duplicated file descriptors
 - ▶ Closing any fd on a file loses locks on dup'ed fds
- ▶ Very bad scaling
 - ▶ fcntl locks maintained in a linked list
 - ▶ Posix suggests deadlock detection → locks are not per-file
 - ▶ Linux has one big global spinlock for all fcntl lock operations
 - ▶ Thundering herd when thousands of waiters are unblocked
- ▶ Automatic cleanup at process exit
 - ▶ Lock waiters are not notified that a process crashed

Process Shared Robust Mutexes

- ▶ `pthread_mutex_t`: pthread API to implement critical sections
 - ▶ Originally intended for multiple threads within a single process
- ▶ Implementation under Linux with atomic operations
 - ▶ No syscall in the non-contended case
 - ▶ Waiting for a locked mutex uses a syscall
- ▶ `PTHREAD_PROCESS_SHARED`: Mutexes in shared memory
 - ▶ Downside: If a mutex holder crashes, nobody can clean up
- ▶ `PTHREAD_MUTEX_ROBUST`: EOWNERDEAD
 - ▶ When a mutex holder dies, a subsequent `pthread_mutex_lock` gets EOWNERDEAD
 - ▶ Linux and Solaris only

Samba Messaging

- ▶ Samba has a multi-writer key/value hash table: TDB
 - ▶ Shared Memory
 - ▶ Coordination via fcntl locks
 - ▶ Where available: Process Shared Robust Mutexes
- ▶ Very fast for heavy concurrent small record updaters
- ▶ Messaging based on tdb:
 - ▶ Every process has a record in a tdb as a mailbox
 - ▶ Signalling via SIGUSR1
- ▶ Simple, but bad under high load
 - ▶ fcntl load brings system down
 - ▶ With robust mutexes it is okay

Unix Domain Datagram Sockets

- ▶ "UDP" on the local box
 - ▶ Contrary to UDP, AF_UNIX DGRAM sockets are reliable
- ▶ One socket per participating process
 - ▶ Limited message size, sender must fragment
- ▶ FD passing possible via sendmsg()
- ▶ Asynchronous send into full queue:
 - ▶ Poll with nonblocking send: High load by senders
- ▶ Blocking connected send scales well under Linux
 - ▶ No thundering herd, contrary to thousands of writers into a pipe
 - ▶ Well tuned for syslog via /dev/log
- ▶ Access to the socket dir can become a bottleneck

Replace ctdb messaging

- ▶ All cluster communication goes through ctdbd
 - ▶ Single Process, Single Threaded
 - ▶ Only 1 CPU used for inter-node messaging
- ▶ Samba assumes reliable messaging
 - ▶ Stream Socket, i.e. TCP between nodes
 - ▶ TCP connection setup too expensive per message
- ▶ One proxy process per peer
 - ▶ Open a normal Unix DGRAM socket
 - ▶ Forward to another node
 - ▶ Sender process decides which proxy to use

Unix Domain Stream Sockets

- ▶ Datagram Sockets not overload-safe on FreeBSD
 - ▶ ENOBUFS returned under overload
 - ▶ Unlike Linux, FreeBSD can't do graceful blocking send
- ▶ Every message involves namespace operations
 - ▶ Not truly scalable, requires a read lock on the socket directory
- ▶ File Descriptors are cheap
 - ▶ epoll/kqueue designed to scale
- ▶ $N * M$ Stream Sockets between processes

- ▶ How to pass stream socket fds?
 - ▶ Every smbd listens on a stream socket
 - ▶ connect() does not behave nicely under overload
- ▶ tmond provides a central hub
 - ▶ Every messaging process connects() once
- ▶ Fresh stream to a peer: socketpair() and sendmsg() via tmond
- ▶ Process Monitoring via tmond
 - ▶ Locks in user space: Wait for a process to go away, retry
 - ▶ General replacement for ctdb_watch_us/ctdb_unwatch

Questions?

`vl@samba.org / vl@sernet.de`