CTDB Performance

Amitay Isaacs
amitay@samba.org

Samba Team
IBM (Australia Development Labs, Linux Technology Center)

CTDB Project

Motivation: Support for clustered Samba

- Multiple nodes active simultaneously
- Communication between nodes (heartbeat, failover)
- Distributed databases between nodes

Features:

- Volatile and Persistent databases
- Cluster-side messaging for Samba
- IP failover and load balancing
- Service monitoring

Community:

- `http://ctdb.samba.org`
- `git://git.samba.org/ctdb.git`, `git://git.samba.org/samba.git`

Overview

- Current Status
- Performance Issues
  - Parallel database recovery
  - Improving database recovery
  - Socket handling
  - Database performace

Current Status

CTDB Releases

- 2.5.6 (February 2016) - 84 patches
  - Support volatile databases in tmpfs
  - Fix vlan interface monitoring
  - Numerous resource leak fixes

CTDB Releases

- 2.5.6 (February 2016) - 84 patches
  - Support volatile databases in tmpfs
  - Fix vlan interface monitoring
  - Numerous resource leak fixes
- End of development in ctdb tree!

Developers

Contributions in 2015

295	Martin Schwenke
242	Amitay Isaacs
21	Volker Lendecke
16	Michael Adam
6	Christof Schmitt
6	Stefan Metzmacher
3	Mathieu Parent
3	Rajesh Joseph
2	Günther Deschner
2	Thomas Nagy
1	David Disseldorp, Jelmer Vernooij
1	Jose A. Rivera, Led
1	Paul Wayper, Ralph Boehme

Developers

Contributions since Jan 2016

150	Martin Schwenke
102	Amitay Isaacs
6	Volker Lendecke
2	Günther Deschner
2	Michael Adam
1	Christof Schmitt
1	Jose A. Rivera
1	Karolin Seeger
1	Robin Hack
1	Steven Chamberlain

Parallel database recovery

Parallel database recovery

Why parallel database recovery?

Parallel database recovery

Why parallel database recovery?

Observation

- Clustered samba running with SMB workload
- A node goes down (overload, admin action, ...)
- CTDB starts recovery, starts freezing databases on all nodes
- Fails to freeze database repeatedly, bans **culprit** node
- Eventually CTDB bans all the nodes in the cluster

Parallel database recovery

Why parallel database recovery?

Observation

- Clustered samba running with SMB workload
- A node goes down (overload, admin action, ...)
- CTDB starts recovery, starts freezing databases on all nodes
- Fails to freeze database repeatedly, bans **culprit** node
- Eventually CTDB bans all the nodes in the cluster

Cause

- Samba is holding a lock on a record
- Samba needs another record lock
- Samba asks CTDB to migrate the record
- The *dmaster* node goes down
- Deadlock!

Parallel database recovery

Why parallel database recovery?

Parallel database recovery

Why parallel database recovery?

- CTDB database recovery was serial

Parallel database recovery

Why parallel database recovery?

- CTDB database recovery was serial
  - Freeze all databases
  - Recover databases one by one
  - Unlock all databases

Parallel database recovery

Why parallel database recovery?

- CTDB database recovery was serial
  - Freeze all databases
  - Recover databases one by one
  - Unlock all databases

Problems

- Causes deadlock!

Parallel database recovery

Why parallel database recovery?

- CTDB database recovery was serial
  - Freeze all databases
  - Recover databases one by one
  - Unlock all databases

Problems

- Causes deadlock!
- Recovery daemon can stay busy for long time

Parallel database recovery

Why parallel database recovery?

- CTDB database recovery was serial
  - Freeze all databases
  - Recover databases one by one
  - Unlock all databases

Problems

- Causes deadlock!
- Recovery daemon can stay busy for long time

Solution

- Recover each database independently and in parallel

Parallel database recovery

How to develop parallel database recovery?

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb
- Current client code not fully async

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb
- Current client code not fully async
- . . . relies on nested event loops

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb
- Current client code not fully async
- . . . relies on nested event loops

Motivation

- Need async code to exercise parallelism

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb
- Current client code not fully async
- . . . relies on nested event loops

Motivation

- Need async code to exercise parallelism
- . . . based on `tevent_req`

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb
- Current client code not fully async
- . . . relies on nested event loops

Motivation

- Need async code to exercise parallelism
- . . . based on `tevent_req`
- Need new communication framework

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb
- Current client code not fully async
- . . . relies on nested event loops

Motivation

- Need async code to exercise parallelism
- . . . based on `tevent_req`
- Need new communication framework
- Improve protocol handling

Parallel database recovery

How to develop parallel database recovery?

- It's all **client** code
- No libctdb
- Current client code not fully async
- . . . relies on nested event loops

Motivation

- Need async code to exercise parallelism
- . . . based on `tevent_req`
- Need new communication framework
- Improve protocol handling
- Improve testability

Parallel database recovery

Where is protocol code?

Parallel database recovery

Where is protocol code?

- Protocol structures distributed in various header files

Parallel database recovery

Where is protocol code?

- Protocol structures distributed in various header files
- Protocol marshalling embedded in the implementation

Parallel database recovery

Where is protocol code?

- Protocol structures distributed in various header files
- Protocol marshalling embedded in the implementation

New design

Parallel database recovery

Where is protocol code?

- Protocol structures distributed in various header files
- Protocol marshalling embedded in the implementation

New design

- All protocol structures – `protocol/protocol.h`

Parallel database recovery

Where is protocol code?

- Protocol structures distributed in various header files
- Protocol marshalling embedded in the implementation

New design

- All protocol structures – `protocol/protocol.h`
- Marshaling for protocol structures
  - `len()`, `push()`, `pull()`

Parallel database recovery

Where is protocol code?

- Protocol structures distributed in various header files
- Protocol marshalling embedded in the implementation

New design

- All protocol structures – `protocol/protocol.h`
- Marshaling for protocol structures
  - `len()`, `push()`, `pull()`
- Marshaling for protocol elements
  - `push()`, `pull()`

Parallel database recovery

Where is protocol code?

- Protocol structures distributed in various header files
- Protocol marshalling embedded in the implementation

New design

- All protocol structures – `protocol/protocol.h`
- Marshaling for protocol structures
  - `len()`, `push()`, `pull()`
- Marshaling for protocol elements
  - `push()`, `pull()`
- Trying to get it right!

Parallel database recovery

Developing new communication framework

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97
`common/reqid.c`	89	72

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97
`common/reqid.c`	89	72
`common/pkt_read.c`	190	260
`common/pkt_write.c`	101	370

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97
`common/reqid.c`	89	72
`common/pkt_read.c`	190	260
`common/pkt_write.c`	101	370
`common/comm.c`	404	843

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97
`common/reqid.c`	89	72
`common/pkt_read.c`	190	260
`common/pkt_write.c`	101	370
`common/comm.c`	404	843
`protocol/protocol_*.c`	8865	3674

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97
`common/reqid.c`	89	72
`common/pkt_read.c`	190	260
`common/pkt_write.c`	101	370
`common/comm.c`	404	843
`protocol/protocol_*.c`	8865	3674
`client/client_*.c`	7352	?

Parallel database recovery

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97
`common/reqid.c`	89	72
`common/pkt_read.c`	190	260
`common/pkt_write.c`	101	370
`common/comm.c`	404	843
`protocol/protocol_*.c`	8865	3674
`client/client_*.c`	7352	?
`ctdb2.c`	6699	?

Developing new communication framework

common/db_hash.c	268	138
common/srvid.c	269	97
common/reqid.c	89	72
common/pkt_read.c	190	260
common/pkt_write.c	101	370
common/comm.c	404	843
protocol/protocol_*.c	8865	3674
client/client_*.c	7352	?
ctdb2.c	6699	?
fake_ctdbd.c		2284

Developing new communication framework

`common/db_hash.c`	268	138
`common/srvid.c`	269	97
`common/reqid.c`	89	72
`common/pkt_read.c`	190	260
`common/pkt_write.c`	101	370
`common/comm.c`	404	843
`protocol/protocol_*.c`	8865	3674
`client/client_*.c`	7352	?
`ctdb2.c`	6699	?
`fake_ctdbd.c`		2284
	24059	7738

Parallel database recovery

Writing parallel database recovery code

Parallel database recovery

Writing parallel database recovery code

- `ctdb_recovery_helper.c` 2809

Parallel database recovery

Writing parallel database recovery code

- `ctdb_recovery_helper.c` 2809
  - Start freeze of all databases
  - Once a database is frozen, recover that database
  - Thaw that database

Parallel database recovery

Writing parallel database recovery code

- `ctdb_recovery_helper.c` 2809
  - Start freeze of all databases
  - Once a database is frozen, recover that database
  - Thaw that database
- Phew!

Parallel database recovery

Writing parallel database recovery code

- `ctdb_recovery_helper.c` 2809
  - Start freeze of all databases
  - Once a database is frozen, recover that database
  - Thaw that database
- Phew!

Next steps

Parallel database recovery

Writing parallel database recovery code

- `ctdb_recovery_helper.c` 2809
  - Start freeze of all databases
  - Once a database is frozen, recover that database
  - Thaw that database
- Phew!

Next steps

- Replace CTDB tool code (`ctdb2.c`)

Parallel database recovery

Writing parallel database recovery code

- `ctdb_recovery_helper.c` 2809
  - Start freeze of all databases
  - Once a database is frozen, recover that database
  - Thaw that database
- Phew!

Next steps

- Replace CTDB tool code (`ctdb2.c`)
- Replace all test code (`tests/src/*.c`)

Improving database recovery

Improving database recovery

Improving database recovery

How is single database recovered?

Improving database recovery

How is single database recovered?

- `PULL_DB` control to collect database records from all nodes
- Combine database records
- `PUSH_DB` control to send database records to all nodes

Improving database recovery

How is single database recovered?

- `PULL_DB` control to collect database records from all nodes
- Combine database records
- `PUSH_DB` control to send database records to all nodes

Problems

- `PULL_DB` and `PUSH_DB` use a single marshall buffer

Improving database recovery

How is single database recovered?

- `PULL_DB` control to collect database records from all nodes
- Combine database records
- `PUSH_DB` control to send database records to all nodes

Problems

- `PULL_DB` and `PUSH_DB` use a single marshall buffer
- What is the database size is large? (`MAX_TALLOC_SIZE`)

Improving database recovery

How is single database recovered?

- `PULL_DB` control to collect database records from all nodes
- Combine database records
- `PUSH_DB` control to send database records to all nodes

Problems

- `PULL_DB` and `PUSH_DB` use a single marshall buffer
- What is the database size is large? (`MAX_TALLOC_SIZE`)
- What's wrong with sending 1GB of data in a single packet?

Improving database recovery

Improving database recovery

New control `DB_PULL`

Improving database recovery

New control DB_PULL

- Recovery helper sends control DB_PULL with *srvid*
- Ctdbd sends chunked database records with *srvid*
- Recovery helper collects all records received with *srvid*
- Ctdbd sends reply to DB_PULL with number of records

Improving database recovery

New control DB_PULL

- Recovery helper sends control DB_PULL with *srvid*
- Ctdbd sends chunked database records with *srvid*
- Recovery helper collects all records received with *srvid*
- Ctdbd sends reply to DB_PULL with number of records

New controls DB_PUSH_START and DB_PUSH_CONFIRM

Improving database recovery

New control DB_PULL

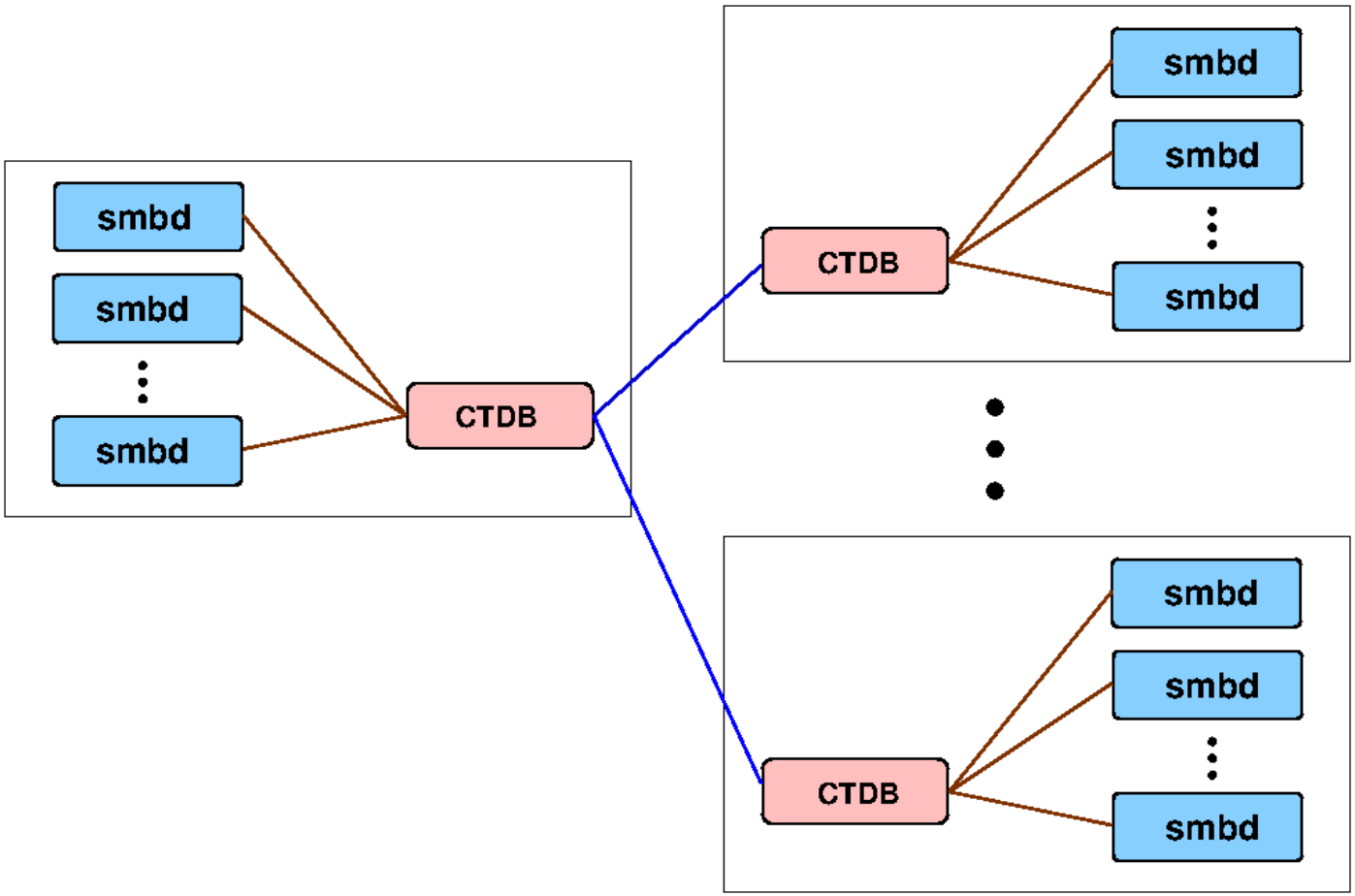
- Recovery helper sends control DB_PULL with *srvid*
- Ctdbd sends chunked database records with *srvid*
- Recovery helper collects all records received with *srvid*
- Ctdbd sends reply to DB_PULL with number of records

New controls DB_PUSH_START and DB_PUSH_CONFIRM

- Recovery helper sends control DB_PUSH_START with *srvid*
- Ctdbd starts listening for messages with *srvid*
- Ctdbd replies to DB_PUSH_START
- Recovery helper sends chunked database records with *srvid*
- Recovery helper sends control DB_PUSH_CONFRIM
- Ctdbd sends reply to DB_PUSH_CONFIRM with number of records

Socket handling

Socket handling



Socket handling

Lots of sockets and fds

Socket handling

Lots of sockets and fds

- TCP connections (few)
- Unix domain connections (thousands!)
- Child pipe fds (tens, sometimes hundreds)

Socket handling

Lots of sockets and fds

- TCP connections (few)
- Unix domain connections (thousands!)
- Child pipe fds (tens, sometimes hundreds)

Problems

Socket handling

Lots of sockets and fds

- TCP connections (few)
- Unix domain connections (thousands!)
- Child pipe fds (tens, sometimes hundreds)

Problems

- Single process single thread `ctdbd`

Socket handling

Lots of sockets and fds

- TCP connections (few)
- Unix domain connections (thousands!)
- Child pipe fds (tens, sometimes hundreds)

Problems

- Single process single thread `ctdbd`
- Scheduling of fds dependent on event system (`epoll`)

Socket handling

Original approach

```
static void queue_io_read(struct ctdb_queue *queue)
{
    if (ioctl(queue->fd, FIONREAD, &num_ready) != 0) { return; }

    to_read = MIN(sz_bytes_req, num_ready);
    nread = read(queue->fd, data + queue->partial.length, to_read);
    queue->partial.length += nread;

    if (nread < sz_bytes_req) { return; }
    num_ready -= nread;

    pkt_size = *(uint32_t *)data;
    pkt_bytes_remaining = pkt_size - queue->partial.length;
    to_read = MIN(pkt_bytes_remaining, num_ready);
    nread = read(queue->fd, data + queue->partial.length, to_read);
    queue->partial.length += nread;

    if (queue->partial.length < pkt_size) { return; }
    queue->callback(data, pkt_size, queue->private_data);
}
```

Socket handling

Original approach

```
static void queue_io_read(struct ctdb_queue *queue)
{
    if (ioctl(queue->fd, FIONREAD, &num_ready) != 0) { return; }

    to_read = MIN(sz_bytes_req, num_ready);
    nread = read(queue->fd, data + queue->partial.length, to_read);
    queue->partial.length += nread;

    if (nread < sz_bytes_req) { return; }
    num_ready -= nread;

    pkt_size = *(uint32_t *)data;
    pkt_bytes_remaining = pkt_size - queue->partial.length;
    to_read = MIN(pkt_bytes_remaining, num_ready);
    nread = read(queue->fd, data + queue->partial.length, to_read);
    queue->partial.length += nread;

    if (queue->partial.length < pkt_size) { return; }
    queue->callback(data, pkt_size, queue->private_data);
}
```

Socket handling

Original approach

Socket handling

Original approach

- Single fd at a time (triggered by tevent)

Socket handling

Original approach

- Single fd at a time (triggered by tevent)
- Read one packet at a time

Socket handling

Original approach

- Single fd at a time (triggered by tevent)
- Read one packet at a time
- Fair scheduling of `epoll_wait()`

Socket handling

Original approach

- Single fd at a time (triggered by tevent)
- Read one packet at a time
- Fair scheduling of `epoll_wait()`
- Affects TCP sockets

Socket handling

Original approach

- Single fd at a time (triggered by tevent)
- Read one packet at a time
- Fair scheduling of `epoll_wait()`
- Affects TCP sockets
- Real-time priority

Socket handling

Original approach

- Single fd at a time (triggered by tevent)
- Read one packet at a time
- Fair scheduling of `epoll_wait()`
- Affects TCP sockets
- Real-time priority

Problem

- Pending data on TCP sockets

New approach

```
#define QUEUE_BUFFER_SIZE       (16*1024)
static void queue_io_read(struct ctdb_queue *queue)
{
    if (ioctl(queue->fd, FIONREAD, &num_ready) != 0) { return; }

    if (queue->buffer.data == NULL) {
        queue->buffer.data = talloc_size(queue, QUEUE_BUFFER_SIZE);
        queue->buffer.size = QUEUE_BUFFER_SIZE;
    }

    navail = queue->buffer.size - queue->buffer.length;
    if (num_ready > navail) { num_ready = navail; }

    if (num_ready > 0) {
        nread = sys_read(queue->fd, queue->buffer.data + queue->buffer.length,
                         num_ready);
        queue->buffer.length += nread;
    }

    queue_process(queue);
}
```

New approach

```
#define QUEUE_BUFFER_SIZE       (16*1024)
static void queue_io_read(struct ctdb_queue *queue)
{
    if (ioctl(queue->fd, FIONREAD, &num_ready) != 0) { return; }

    if (queue->buffer.data == NULL) {
        queue->buffer.data = talloc_size(queue, QUEUE_BUFFER_SIZE);
        queue->buffer.size = QUEUE_BUFFER_SIZE;
    }

    navail = queue->buffer.size - queue->buffer.length;
    if (num_ready > navail) { num_ready = navail; }

    if (num_ready > 0) {
        nread = sys_read(queue->fd, queue->buffer.data + queue->buffer.length,
                         num_ready);
        queue->buffer.length += nread;
    }

    queue_process(queue);
}
```

Socket handling

New approach

```
#define QUEUE_BUFFER_SIZE       (16*1024)
static void queue_io_read(struct ctdb_queue *queue)
{
    if (ioctl(queue->fd, FIONREAD, &num_ready) != 0) { return; }

    if (queue->buffer.data == NULL) {
        queue->buffer.data = talloc_size(queue, QUEUE_BUFFER_SIZE);
        queue->buffer.size = QUEUE_BUFFER_SIZE;
    }

    navail = queue->buffer.size - queue->buffer.length;
    if (num_ready > navail) { num_ready = navail; }

    if (num_ready > 0) {
        nread = sys_read(queue->fd, queue->buffer.data + queue->buffer.length,
                         num_ready);
        queue->buffer.length += nread;
    }

    queue_process(queue);
}
```

Socket handling

New approach

Socket handling

New approach

- Single fd at a time (triggered by tevent)

Socket handling

New approach

- Single fd at a time (triggered by tevent)
- Read multiple packets at a time

Socket handling

New approach

- Single fd at a time (triggered by tevent)
- Read multiple packets at a time
- Fair scheduling of `epoll_wait()`

Socket handling

New approach

- Single fd at a time (triggered by tevent)
- Read multiple packets at a time
- Fair scheduling of `epoll_wait()`
- TCP sockets no longer affected

Socket handling

New approach

- Single fd at a time (triggered by tevent)
- Read multiple packets at a time
- Fair scheduling of `epoll_wait()`
- TCP sockets no longer affected
- Real-time priority

Socket handling

New approach

- Single fd at a time (triggered by tevent)
- Read multiple packets at a time
- Fair scheduling of `epoll_wait()`
- TCP sockets no longer affected
- Real-time priority

Problem

- CTDB daemon can stay busy between `epoll_wait` calls

Socket handling

New approach

- Single fd at a time (triggered by tevent)
- Read multiple packets at a time
- Fair scheduling of `epoll_wait()`
- TCP sockets no longer affected
- Real-time priority

Problem

- CTDB daemon can stay busy between `epoll_wait` calls
- . . . Handling event took 345 seconds!

Socket handling

There's no winning . . .

Socket handling

There's no winning . . .

- `QUEUE_BUFFER_SIZE` is heuristic and workload-dependent

Socket handling

There's no winning . . .

- `QUEUE_BUFFER_SIZE` is heuristic and workload-dependent
- Multiple event contexts?

Socket handling

There's no winning . . .

- `QUEUE_BUFFER_SIZE` is heuristic and workload-dependent
- Multiple event contexts?
- Avoid processing thousands of fds in a single process

Socket handling

There's no winning . . .

- `QUEUE_BUFFER_SIZE` is heuristic and workload-dependent
- Multiple event contexts?
- Avoid processing thousands of fds in a single process
- Need better communication infrastructure!

Socket handling

There's no winning . . .

- `QUEUE_BUFFER_SIZE` is heuristic and workload-dependent
- Multiple event contexts?
- Avoid processing thousands of fds in a single process
- Need better communication infrastructure!
  - New proxy design using Volker's ~~`unix_msg`~~, `tmond`?

Socket handling

There's no winning . . .

- `QUEUE_BUFFER_SIZE` is heuristic and workload-dependent
- Multiple event contexts?
- Avoid processing thousands of fds in a single process
- Need better communication infrastructure!
  - New proxy design using Volker's ~~`unix_msg`~~, `tmond`?
  - Avoid re-inventing wheel – zeromq, . . .

Database performance

Database performance

Current database models

volatile	persistent

Database performance

Current database models

volatile	**persistent**
distributed data	replicated data

Database performance

Current database models

volatile	**persistent**
distributed data	replicated data
single copy	multiple copies

Database performance

Current database models

volatile	**persistent**
distributed data	replicated data
single copy	multiple copies
data loss on failure	loss-less

Database performance

Current database models

volatile	**persistent**
distributed data	replicated data
single copy	multiple copies
data loss on failure	loss-less
per-node per-db per-chain mutex	clusterwide per-db mutex

Database performance

Current database models

volatile	**persistent**
distributed data	replicated data
single copy	multiple copies
data loss on failure	loss-less
per-node per-db per-chain mutex	clusterwide per-db mutex
disk backed	disk backed

Database performance

Current database models

volatile	**persistent**
distributed data	replicated data
single copy	multiple copies
data loss on failure	loss-less
per-node per-db per-chain mutex	clusterwide per-db mutex
disk backed	disk backed
cluster-wide traverse	local traverse

Database performance

Current database models

volatile	persistent
distributed data	replicated data
single copy	multiple copies
data loss on failure	loss-less
per-node per-db per-chain mutex	clusterwide per-db mutex
disk backed	disk backed
cluster-wide traverse	local traverse
shared access	client-server access

Database performance

Volatile databases

Database performance

Volatile databases

- CTDB is involved in migrating a record

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB
  - Use robust mutexes instead of fcntl locks

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB
  - Use robust mutexes instead of fcntl locks

	fcntl	**mutexes**
single record	305k	555k

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB
  - Use robust mutexes instead of fcntl locks
  - ...

	fcntl	**mutexes**
single record	305k	555k

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB
  - Use robust mutexes instead of fcntl locks
  - ...
- Unless there is contention!

	fcntl	**mutexes**
single record	305k	555k

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB
  - Use robust mutexes instead of fcntl locks
  - ...
- Unless there is contention!

	fcntl	**mutexes**
single record	305k	555k
contention	165k	300k

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB
  - Use robust mutexes instead of fcntl locks
  - ...
- Unless there is contention!
- `CTDB_DBDIR=tmpfs`

	fcntl	**mutexes**
single record	305k	555k
contention	165k	300k

Database performance

Volatile databases

- CTDB is involved in migrating a record
- Record access is local and CTDB is not involved
- Scalability dependent on performance of TDB
  - Use robust mutexes instead of fcntl locks
  - ...
- Unless there is contention!
- `CTDB_DBDIR=tmpfs`

	fcntl	**mutexes**
single record	305k	555k
contention	165k	300k
tmpfs	176k	312k

Database performance

Persistent databases

Database performance

Persistent databases

- Every update is a cluster-wide transaction

Database performance

Persistent databases

- Every update is a cluster-wide transaction
- Scalability dependent on performance of `g_lock`

Database performance

Persistent databases

- Every update is a cluster-wide transaction
- Scalability dependent on performance of `g_lock`

	fcntl	**mutexes**
g_lock	1600	3000

Database performance

Persistent databases

- Every update is a cluster-wide transaction
- Scalability dependent on performance of `g_lock`
- `fdatasync()`!

	fcntl	**mutexes**
g_lock	1600	3000

Database performance

Persistent databases

- Every update is a cluster-wide transaction
- Scalability dependent on performance of `g_lock`
- `fdatasync()`!

	fcntl	**mutexes**
g_lock	1600	3000
persistent	10	10

Database performance

Persistent databases

- Every update is a cluster-wide transaction
- Scalability dependent on performance of `g_lock`
- `fdatasync()`!
- Concurrent transactions on different databases

	fcntl	**mutexes**
g_lock	1600	3000
persistent	10	10

Database performance

Inventing new database models

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead
  - Storing CTDB state information - e.g. tickles

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead
  - Storing CTDB state information - e.g. tickles
- Persistent, clusterwide per-db per-chain mutex
  - Avoid single transaction per database restriction

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead
  - Storing CTDB state information - e.g. tickles
- Persistent, clusterwide per-db per-chain mutex
  - Avoid single transaction per database restriction
  - Useful for updating single keys

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead
  - Storing CTDB state information - e.g. tickles
- Persistent, clusterwide per-db per-chain mutex
  - Avoid single transaction per database restriction
  - Useful for updating single keys
- Persistent, lazy replication of data
  - Avoid single (or limited multiple) point(s) of failure

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead
  - Storing CTDB state information - e.g. tickles
- Persistent, clusterwide per-db per-chain mutex
  - Avoid single transaction per database restriction
  - Useful for updating single keys
- Persistent, lazy replication of data
  - Avoid single (or limited multiple) point(s) of failure
  - Storing persistent file handles

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead
  - Storing CTDB state information - e.g. tickles
- Persistent, clusterwide per-db per-chain mutex
  - Avoid single transaction per database restriction
  - Useful for updating single keys
- Persistent, lazy replication of data
  - Avoid single (or limited multiple) point(s) of failure
  - Storing persistent file handles
- Volatile, partially replicated data
  - Avoid single (or limited multiple) point(s) of failure

Database performance

Inventing new database models

- Persistent, in-memory
  - Avoid `fdatasync()` overhead
  - Storing CTDB state information - e.g. tickles
- Persistent, clusterwide per-db per-chain mutex
  - Avoid single transaction per database restriction
  - Useful for updating single keys
- Persistent, lazy replication of data
  - Avoid single (or limited multiple) point(s) of failure
  - Storing persistent file handles
- Volatile, partially replicated data
  - Avoid single (or limited multiple) point(s) of failure
  - Storing persistent file handles

Questions / Comments