CTDB node and IP management

Matt Wu

2008-03-26

Contents

1 Introduction

This document describes CTDB analysis and improvement design on CTDB flexible node/ip management. As part of Lustre pCIFS project, CTDB improvement focuses in failover support to all pCIFS clients.

2 Document Scope

2.1 What is in

- introducation of CTDB and pCIFS
- detailed analysis of CTDB
- design of CTDB flexible node management
- design of CTDB ip reassignment
- ideas on CTDB improvment (for future)

2.2 What is not in

- Samba and CIFS protocol internals
- Lustre pCIFS project for windows
- timed out requests resending by pCIFS
- IB transport analysis

3 What is CTDB

CTDB is a database implementation, providing TDB-like APIs to Samba or other applications for temporary context data management. It relies the underlying clustered filesystem to manage TDB database files, since TDB uses FLOCK to protect database access.

As TDB database is shared to all nodes in the cluster, CTDB provides failover to all CTDB clients (like Samba). When a Samba/CTDB node hangs, another node will takeover the dead node's IP address and restore all TCP connections. The failover process is completely transparent, so the Samba client won't notice the failover process and will stay alive with the new cluster configuration.

More information is available at http://ctdb.samba.org

ctdb architecture:

ctdb failover:

the second CTDB/Samba node tookover the ip address of the first node after the first node hang. all the context of CIFS client 1 and tcp connections were restored in second CTDB/Samba node.

4 What is pCIFS

Lustre pCIFS client provides parallel i/o support to Lustre servers shared by Samba. With pCIFS, data i/o is to be dispatched smartly to Lustre OST nodes while the metadata operations will be kept untouched and go directly to Lustre MDS server.

pCIFS client is actually a Samba client. While Samba acts as CTDB client since CTDB provides TDB service to Samba. And Samba also exports Lustre client (mountpoint) out for all Samba clients. see the picture below:

pCIFS architecture:

5 CTDB analysis

5.1 how to start ctdb

Please refer http://wiki.samba.org/index.php/CTDB_Setup for details.

5.2 ctdb source tree

```
client          /* routines for CTDB client to talk to CTDB daemon */
common          /* common routines shared by client and daemon */
config          /* configuration samples and event script callback */
    events.d    /* event scripts */
doc             /* document, help manual */
lib             /* infiniband support routines */
include         /* common header files */
lib
    events      /* events lib */
    popt        /* command parameters process lib */
    replace     /* substitution of lib routines *
    talloc      /* memory allocation management*/
    tdb         /* local TDB implementation */
    util        /* miscellaneous general routines */
    packaging
server          /* core code of CTDB daemon */
takeover        /* routines to handle ip-takeover */
tcp             /* tcp/socket generic routines */
tests           /* bench util */
tools           /* ctdb control utility */
web             /* help manual*/
```

5.3 fundament

5.3.1 events

The events library provides an asynchronous handling mechanism upon select or epull (preferable). Everyone can register an event object plus a callback to monitor any event either that a file becomes readable or writable or that a timer expires. When the event is triggered, the callback routine, called "handler", will be executed to deal with the event.

Here's a typical scenario using events:

```
int main ()
{
    /* open a file or network socket */
    int fd = ...;
    /* allocate talloc context to trace all memory allocations*/
    TALLOC_CTX  ctx = talloc_new(NULL);
    /* initialize event context */
    struct event_context ev = event_context_init(ctx);
    /* add an event to detect data arrival, the handler routine
       data_arrival_handler will be called to read data from fd */
    struct fd_event *fde = event_add_fd(ev, ctx, fd, EVENT_FD_READ|
                                        EVENT_FD_AUTOCLOSE,
                                        data_arrival_handler, fd);

    /* add a timer to cancle previous event if time expires */
       struct timed_event *te = event_add(ev, ctx, timeval_from_current,
                                          cancel_data_request, fde);
    /* start events process engine, it will loop forever if there's
       events registerred and quit when the event list is empty. For
       this example we'll get it return after canceling read request */
    event_loop_wait(ev);
    /* clean up all memory or opened handles */
    talloc_free(ctx);
    return -1;
}
void cancel_data_reqest(
    struct event_context *ev,
    struct timed_event *te,
    struct timeval, void *p)
{
   ASSERT(p != NULL);
   /* destory the fd event. talloc_free will call the correspond
      destructure callback to cleanup the internal members of struct
      fd_event. the destructor routine is registered by event_add_fd */
   talloc_free(p);
}
```

5.3.2 tdb & ctdb

tdb is a light-weight database system. It provides a full data semantic operations, like database open / create /close / traverse, record creation / fetch / erase / replace and transaction management. tdb depends on flock (fcntl) as it's dirtributed lock system. There's just the reason we have to specify the clientoption "-o flock" when mounting Lustre as CTDB host cluster file system.

tdb is used widely, for example e2fsprogs uses tdb to management inode and dentry cache which could be too big to be stored in system memory.

here's an example of using tdb to store enumed all pci devices installed in system:

```
int main ()
{
    TDB_DATA key, data, ret;
    struct pci_device {
        int vid; /* vendor id */
        int did; /* device id */
        int cnt; /* number of devices */
        char desc[255]; /* description */
    } dev;
    int i = 0;
    /* open or create a tdb database file */
    struct db_context *db = tdb_open("devices.db", 0, TDB_CLEAR_IF_FIRST,
                                     O_RDWR | O_CREAT | O_TRUNC, 0600);
    while (enum_pci_device(i, &dev) == 0) {
        /* initialize key */
        key.dptr = &dev; key.dsize = offset_of(struct pci_device, desc);
        /* check whehter there's a record for this device */
        ret = tdb_fetch(db, key);
        if (ret.dptr) {
            /* record exists, we just need inc device count */
            struct pci_device *pd = ret.dptr;
            pd->cnt++;
            tdb_store(db, key, ret);
            free(ret.dptr);
        } else {
            /* insert this record into tdb database */
            data.dptr = &dev;
            data.dsize = sizeof(dev);
            dev.cnt = 0;
            tdb_store(db, key, data, TDB_INSERT);
        }
    }
    /* show all the pci devices */
    tdb_traverse(db, show_device, db);
    /* close tdb database*/
```

```
    tdb_close(db);
    return 0;
}
```

Samba doesn't use TDB api in such a direct way, instead, it uses a wrapper set of TDB/CTDB routines. For example Samba uses locking.tdb to store file sharing modes for all open instances. When a file is to be opened, Samba will call access_share_mode to query the lock information from CTDB and then check whether or not there are any conflicts between this open requests and all existing opened instances.

```
void access_share_mode ()
{
    /* open or create session database */
    struct tdb_context *lock_db = tdb_open("session.tdb",
                                    lp_open_files_db_hash_size(),
                                    TDB_CLEAR_IF_FIRST|TDB_DEFAULT,
                                    O_RDWR | O_CREAT, 0644);
    /* now parepare to query file's locking info */
    struct file_id fid;

    /* initialize file_id, map_device_id will hash file sysem name
       or fsid, for a cluster sharing the same fs name, the file
       on different node will be treat as identical. actually CTDB
       provides several methods to map device (hash it) and they are
       the same in principle. */
    fid.devid = map_device_id(fs_dev); fid.inode = inode;
    /* query a file's locking information from database */
    TDB_DATA key;
    key.dptr = &fid;
    key.dsize = sizeof(fid);
    /* ctdb_fetcd_locked will call ctdb_fetch and remain the record
       as locked if the record exists. If the record doesn't exist,
       it will call ctdbd_migrate to request DMASTER of this record
       and get a valid but empty record as a return */
    struct db_record * record = lock_db->fetch_locked(lock_db, NULL,key);
    if (record == NULL) {
        /* error occur */
        return ERROR_SHARING_VIOLATION;
    }
    /* then we can docode the sharing mode info from this record.
       when the operation is done, we need save data back to the
       record and unlock it, actually this operations is done in
       a descturector handler for Samba */
    record->store(record, new_data, TDB_REPLACE);
    /* unlock the record and free it, record destructor will call
```

```
        tdb_chainunlock to unlock before freeing record */
    talloc_free(record);
    /* close database and free db context */
    talloc_free(lock_db);
}
```

5.3.3 ctdb request types

CTDB uses different requests to communicate between CTDB daemons and CTDB clients.

1. request by CTDB user/client: CTDB_REQ_CALL/CTDB_REPLY_CALL. it requests a data record in ctdb database CTDB_REPLY_ERROR: reply error information if case it fails to operate the tdb record
2. CTDB_REQ_DMASTER: current dmaster wants to give up the dmaster to another node, thus it sends CTDB_REQ_DMASTER t othe LMASTER (lock master). CTDB_REPLY_DMASTER: the lmaser will send the assignment command the the furture DMASTER.
3. CTDB_REQ_KEEPALIVE: it represents heartbeating message between CTDB nodes
4. CTDB_REQ_CONTROL/CTDB_REPLY_CONTROL: various control commands, mainly called between recovery client and CTDB daemons.
5. CTDB_REQ_MESSAGE: this is a special type of request. both client and daemon can register a service/message handler for a specified message id. Then all handlers (of both daemon and client) will be triggered when the event happens.

```
/* dynamic list for services handlers */
struct ctdb_message_list {
    struct ctdb_context *ctdb;             /* pointer to ctdb */
    struct ctdb_message_list *next, *prev; /* double link list */
    uint64_t srvid;                        /* service id ? */
    ctdb_message_fn_t message_handler;     /* message handler */
    void *message_private;
}
The service id can be any of the following values:
1, CTDB_SRVID_ALL: means any type of message id
2, CTDB_SRVID_RECOVERY: to notify the recovery process started
3, CTDB_SRVID_RECONFIGURE: CTDB cluster is just resotred from recovery
4, CTDB_SRVID_RELEASE_IP: to release the ip address
5, CTDB_SRVID_NODE_FLAGS_CHANGED: to notify a node flags are changed
6, CTDB_SRVID_BAN_NODE: user just bans a node
7, CTDB_SRVID_UNBAN_NODE: user unbans a node
```

5.4 daemon startup

this part is to describe the CTDB daemon startup process.

5.4.1 general overview

daemon startup process generally does 5 tasks:

1. initialize core structure ctdb_context
2. start daemon unix_socket listening clients requests
3. start tcp or IB transport listen on other CTDB daemon nodes
4. start recovery process to monitor whole CTDB cluster
5. start monitor events to detect all nodes status

5.4.2 core structures

```
struct ctdb_context {
    struct event_context *ev;   /* context for event engine */
    uint32_t recovery_mode;     /* recovery state: CTDB_RECOVERY_NORMAL /
                                   CTDB_RECOVER_ACTIVE */
    uint32_t monitoring_mode;   /* CTDB_MONITORING_ACTIVE or /
                                   CTDB_MONITORING_DISABLED */
    TALLOC_CTX *monitor_context; /*CTDB_FREEZE_NONE,CTDB_FREEZE_PENDING,
                                   CTDB_FREEZE_FROZEN */
    struct ctdb_tunable_tunable; /* values of all adjustable tunable
                                    parameters */
    enum ctdb_freeze_mode freeze_mode; /* all databases are frozen ? */
    struct ctdb_freeze_handle *freeze_handle;
    struct ctdb_address address; /* ip address of this node */
    const char * name;           /* ip_address:port */
    const *db_directory;         /* common directory to store sharable
                                    tdb databases in the cluser */
    const char* transport:       /* "tcp" or "ib" */
    const char * logfile;        /* logging file */
    char *node_list_file;        /* nodes configuratin  file name */
    char *recovery_lock_file;    /* a common file in the cluster,
                                    used as a recovery lock */
    int recovery_lock_fd;        /* the opened descriptor */
    uint32_t vnn;                /* this node's vnn */
    uint32_t num_nodes;          /* total nodes number */
    unit32_t num_connected;      /* total nodes connected */
    unsigned flags;              /* flags */
    struct idr_context *idr;     /* unique 16-bit ctdb request id
                                    creator */
```

```
    nint16_t idr_cnt;
    struct ctdb_node **nodes;    /* nodes array - indexed by vnn */
    char * err_msg;              /* last error message */
    const struct ctdb_methods *methods; /* transport methods: daemon
                                           listening, package handling
                                           for transport level */
    const struct ctdb_upcalls *upcalls; /* transport upcalls, routines
                                            related to ctdb_context */
    void *private_data;          /* private structure to transport
                                    staff: it's in sruct ctdb_tcp for
                                    transport tcp connections */
    struct ctdb_db_context *db_list; /* list header for all tdb
                                        databases */
    struct ctdb_message_list *message_list; /* SRVID message list */
    struct ctdb_daemon_data daemon; /* ctdb daemon core structure */
    struct ctdb_statistics statistics; /* requests statistics */
    struct ctdb_vnn_map * vnn_map; /* the real relationship between
                                      vnn <--> ctdb_nodes. in case
                                      failover happends, several VNN
                                      might pointer to a same nodes */
    uint32_t num_clients;          /* number of connected client */
    uint32_t recovery_master;      /* vnn of the recovery master */
    struct ctdb_call_state * pending_calls; /* outgoing calls */
    struct ctdb_takeover takeover; /* takeover related sturcture */
    struct ctdb_tcp_list *tcp_list;/* TCP/IP connectons to this node
                                      context of all Samba clients */
};
```

5.4.3 program logic

brief flow chart of ctdb startup:

```
main () {
    1)initialize parameters and tunables from cmdline and config files
    2)initialize event context:event_context_init()
    3)initialize ctdb_context structures with tunable values transport:
tcp nodelist initializing (ctdb_node, ctdb_vnn_map, private/public ip
addresses) in ctdb_cmdline_init()
    4)ctdb_start_daemon()
}
ctdb_start_daemon() {
    1) create unix socket daemon and listen on it to accept client
    2) lock all the existing databases (i.e. freeze)
    3) set event callback: ctdb_accept_client on the created socket fd
    4) ctdb_main_loop()
}
```

```
ctdb_main_loop() {
    1) ctdb_tcp_init(): initialize transport structure, allocate ctcp
    2) ctdb_tcp_initialise(): initialize transposrt and create a tcp
       daemon to listen CTDB events from other CTDB nodes
    3) ctdb_release_all_ips to release all possible public ips on the
       public ethernet
    4) start a user script event (startup event) and set the callback
       routine: ctdb_start_transport
    5) event_loop_wait(): will loop forever and process pended and
       incoming requests
}
ctdb_start_transport() {
    1) ctdb_tcp_start: allocate ctdb_tcp_node for every node and try
       to connect to all the nodes expcet itself and then set events
       callbacks(ctdb_node_connect_write). when the socket is becoming
       writable, ctdb_node_connect_write will be triggered and it
       will set package queue and then call ctdb_node_connected to
       change the flags to connected of the connected node: remove
       the CTDB_NODE_DISCONNECTED flag to mark the node is active
    2) ctdb_start_recoverd(): start recovery monitor process
    3) ctdb_start_monitoring():start monitor events
}
ctdb_start_recoverd() {
    1) it will start a CTDB client and monitor all the nodes. this
       part will be described in detail later in recovery process.
       general it will do 3 tasks:
       a) check whether recovery master exists. If not, then do election
          to elect a recovery master out among all nodes of the cluster
       the following two tasks are to be done only by recovery master:
       b) monitor the whole CTDB cluster and start a recovery process
          when the nodes status is not consistent.
       c) do ip takeover when there's a node not responding
}
ctdb_start_monitoring() {
    1) set a timed event to call ctdb_check_for_dead_nodes to check
       the status of other nodes (except itself). If the remote node
       has sent requests to this node, it just marks the remote node
       as CONNECTED (calling ctdb_node_connnected). If there's no any
       information (like keep_alive request) from the remote node in
       a certain interval, CTDB assumes the remote node is dead, then
       calls ctdb_node_dead will mark the node as DISCONNECTED and
       cancels all the pending requests related to this node. the node
       status can be restored to connected if it's alive and sending
       packages without any intervention of recovery process.
    2) set another timed event to call ctdb_check_health.
       ctdb_check_health calls user event script to make sure the
```

```
        current node is active and then ctdb_health_callback is called
        if script upcall returns. if the script returns failure, a flag
        NODE_FLAGS_UNHEALTHY need to be masked to current node. Otherwise
        NODE_FLAGS_UNHEALTHY will be cleared. when node flags is changed,
        ctdb_daemon_send_message will be called too to notify an event of
        CTDB_SRVID_NODE_FLAGS_CHANGED to all connected nodes. all the
        connected nodes including the sender are to receive this message,
        then corresponding handlers are to be triggered to response:
        a) one is flag_change_handler on daemon side to modify the flags
           of the corresponding node. if the node is the just sender and
           the node is marked as disconnected, it will release all the
           public ip addresses, and in turn a recovery process is needed.
           This part will be described later.
        b) another handler is monitor_handler in recovery client process.
           monitor_handler will decide whether or not to do ip takeover.
}
```

5.5 nodes management

In this part I'm trying to describe a whole lifecycle of a ctdb_node and how ctdb node is managed by CTDB.

Currently ctdb_node is kept in a fixed-size array in ctdb_context, and no gap is expected. every CTDB node keeps a complete copy of all other nodes's status and monitors all nodes changes all the time.

5.5.1 ctdb_node structures

```
struct ctdb_node {
    struct ctdb_context *ctdb;
    struct ctdb_address * address; /* ip_address : port */
    const char * name;  /* ip_address:port */
    void *private_data; /* private to transport, pointer to stru
                           ctdb_tcp_node for tcp transport */
    uint32_t vnn;       /* virtual node number ? from 0 to (nodes
                           number - 1), number/order identifier */
    uint32_t flags;     /* states */
    /*used by node monitoring routine: ctdb_check_for_dead_nodes */
    uint32_t dead_count;/* number of internals, how much time pasts
                           from the monitor receives the previous
                           message from this node. when dead_count
                           beyonds the limit, it will be treat ad a
                           dead node */
    unit32_t rx_cnt;    /* received packages from other nodes during
                           the checking internal */
    uint32_t tx_cnt;    /* transmitted packages to other nodes during
                           the checking interval.*/
```

```
    /* a list of controls pending to this node. we can time then
       out quickly when a node becomes dead. */
    struct daemon_control_state * pending_controls;
    /* public address of this node, if user specifies */
    const char *public_address;  /* ip_address:port */
    uint8_t public_netmask_bits; /* subnet mask of public ip address */
    /* the node number that has taken over this node's public address,
       -1 means never */
    int32_t takeover_vnn;
}
struct ctdb_context {
    ...
    uint32_t vnn;             /* current node's vnn */
    uint32_t num_nodes;       /* total nodes number */
    unit32_t num_connected;   /* total nodes connected */
    ...
    struct ctdb_node **nodes; /* node array in the whole cluster */
    ...
    struct ctdb_vnn_map * vnn_map;
}
```

5.5.2 ctdb node flags

```
1, NODE_FLAGS_DISCONNECTED 0x00000001 /* node isn't connected */
2, NODE_FLAGS_UNHEALTHY 0x00000002    /* monitoring says node isn't
                                         healthy */
3, NODE_FLAGS_PERMANENTLY_DISABLED 0x00000004 /* administrator has
                                                 disabled node */
4, NODE_FLAGS_BANNED 0x00000008 /* recovery daemon has banned the node */
5, NODE_FLAGS_DISABLED (NODE_FLAGS_UNHEALTHY|
                        NODE_FLAGS_PERMANE NTLY_DISABLED)
6, NODE_FLAGS_INACTIVE (NODE_FLAGS_DISCONNECTED|NODE_FLAGS_BANN ED)
```

5.5.3 ctdb_node initialization

1)ctdb_node creation

a) during startup, ctdb_set_nlist loads all nodes settings from CTDB config files and then call ctdb_add_node one by one. ctdb_add_node will reallocate ctdb_context->nodes array and initialize it's own node slot, such like ctdb_node->name, ctdb_node->vnn, etc. all ctdb_node will be set as CTDB_NODE_DISCONNECTED in default. for current node, it will clear the CTDB_NODE_DISCONNECTED flag and update ctdb_context->vnn to it's own vnn number.

b) allocate the vnn map array. the vnn_map arrary describes which node is represented by which vnn. normally it's represented by itself.

but when failover happens, there will be node who represents more than one nodes.
2) ctdb_tcp_node creation
tcp node is to be allocated and initialized in ctdb_tcp_initialise. ctdb_tcp_initialise calls ctdb_tcp_add_node to construct a ctdb_tcp_node structure for every node.

5.5.4 tcp connection

ctdb_start_transport (here it only covers tcp transport) is to build all tcp connections between CTDB nodes. it calls ctdb_tcp_start to allocate ctdb_tcp_node for every node and start connecting to all nodes but itself, then set events callbacks (ctdb_node_connect_write). when the socket description becomes writable, ctdb_node_connect_write will be triggered, then it will call ctdb_node_connected to change the flags to connected of the connected node: removing the CTDB_NODE_DISCONNECTED flag. afet this stage, all the nodes will be have the DISCONNECTED flag cleared, i.e. all nodes are active and connected. In the remote side, ctdb_listen_event accepts the connection request issued by ctdb_tcp_node_connect and keeps all the context in a local context (struct ctdb_incoming).
We can conclude that between every two nodes pair there are two connections: one is only for traffic sending and the other is only for traffic receiving. The former is managed by ctdb_tcp_node, the latter is managed by ctdb_incoming structure.

5.5.5 node flags state

1) callbacks to monitor node flag changes
a) ctdb_check_for_dead_nodes: check other nodes and mark it as dead when there's no response in a specified inernal. it also marks the node as active if it recently gets messages from the node.
b) ctdb_check_healthy: check the health status of the node itself
c) flag_change_handler: handling flags change notification from other nodes (normally sent by ctdb_check_healthy)
2) user interfere to BAN or UNBAN a node
user can ban or unban any node in a CTDB cluster with ctdb control util. user's request will be handled by the recovery process instead of the CTDB daemon. both ctdb control util and ctdb recovery process are logic clients of the ctdb daemon:

control util	<- unix_socket ->	ctdb daemon	<- unix_socket ->	recovery process

during startup, ctdb recovery process registers callbacks ban_handler for CTDB_SRVID_BAN_NODE and unban_handler for CTDB_SRVID_UNBAN_NODE.

```
When a node is to be banned, the recovery process will send a flag
change request to the node to be banned and then setup the banned_nodes
array in ctdb_recoverd structure. ctdb_unban_node just does the reverse
of ctdb_ban_node.
the daemon of the node to be banned will receive the request from ctdb
recovery process and call ctdb_control_modflags to notify all CTDB nodes
of the coming change. a cluster recovery is required when an active node
is banned.
3) nodes refresh during recovery process
ctdb recovery master will try to update all other nodes's vnn map and
nodes flags to it's own copy to keep a consistent state during recovery
process.
at that time all database are frozen, and all Samba requests will just
be pended.
```

5.6 daemon and client connection

This part is to describe how CTDB daemon manage the connection issued by CTDB client (Samba).

5.6.1 core structures

```
struct ctdb_conext {
    ...
    struct ctdb_daemon_data daemon;
    ...
    uint32_t num_clients;
    ...
}
struct ctdb_daemon_data {
    int sd;         /* unix socket listening for client's requests */
    char *name;     /* unix socket name: /tmp/ctdb.socket */
    struct ctdb_queue *queue; /* requests queue, network i/o */
}
struct ctdb_client {
    struct ctdb_context *ctdb;
    int fd; /* file descriptor to unix socket connected to daemon */
    struct ctdb_queue * queue;       /* request queue */
    uint32_t client_id;              /* client id*/
    struct ctdb_tcp_list * tcp_list; /* tcp connections between this
                                        node and it's Samba clients */
}
```

5.6.2 connection logic

seq	node	behavior
1	daemon	ctdb_start_daemon is being called to open a unix socket (/tmp/.ctdb_sockets) and register a callback (ctdb_accept_client) to monitor the incoming requests
2	client	a client should ctdb_socket_connect first to talk to ctdb daemon. ctdb_socket_connect first connects to daemon unix socket. (/tmp/.ctdb_sockets)
3	daemon	ctdb_accept_client is triggered: allocate ctdb_client structure, increment ctdb_context->num_clients, and allocate a client id, then allocat the ctdb_queue for data process engine
4	client	set up the client side ctdb_queue for i/o

5.7 request process engine

5.7.1 core structures

```
struct ctdb_partial {
    uint8_t * data;              /* data buffer */
    uint32_t length;             /* length of valid data */
};
struct ctdb_queue_pkt {
    struct ctdb_queue_pkt *next, *prev;
    uint8_t *data;               /* data buffer not sending */
    uint32_t length;             /* remained valid data */
    uint32_t full_length;        /* full length of original package
                                    including buffer alignment */
};
struct ctdb_queue {
    struct ctdb_context *ctdb;
    struct ctdb_partial partial; /* buffer for incoming packages */
    struct ctdb_queue_pkt *out;  /* queue of all ongoing packages */
    struct fd_event *fde;        /* tcp/unix socket events object */
    int fd;                      /* tcp/unix socket handle description */
    size_t alignment;            /* package alignment */
    ctdb_queue_cb_fn_t callback; /* incoming package handler */
};
```

5.7.2 request process logic

here we use an example to explain the request process logic. we imagine there are two nodes: node 0 and node 1. on every node there's a client, like Samba.

```
a client can only talk to it's daemon via the unix socket. when it
wants to broadcast to other nodes, it sends the request to it's
daemon,then the daemon will connect the just node who owns the data
record. the connection is basing tcp or ib between daemon nodes.
```

roles description:

node 0 (requestor)			node 1 (dmaster)			
client 0	<- unix socket ->	daemon 0	<- tcp network ->	daemon 1	<- unix socket ->	client 1

request handling flow:

1	client 0 (node 0)	1. client issues a request CTDB_REQ_CALL and queues the request to out_queue list in ctdb client queue (the queue is ownered by ctdb_context.daemon). 2. queue_io_handler is triggered and calls queue_io_write to write the data in queue to unix socket
2	daemon 0 (node 0)	1. queue_io_handler is triggered when data comes. then queue_io_read will read data from unix socket to the buffer of queue->partial.data and then calls ctdb_daemon_read_cb. 2. ctdb_daemon_read_cb validates the request package and pass it to daemon_incoming_packate. 3. daemon_incoming_package will dispatch the package to the corresponding request engine. for CTDB_REQ_CALL, it's daemon_request_call_from_client. 4. daemon_request_call_from_client will first initiate a new request to be sent to the real dmaseter of the target record. then calls ctdb_daemon_call_send_remote to allocate a ctdb_call_state to represent the original request and place ctdb_call_state into ctdb_context's pending_calls list and idr tree, and pass the request to ctdb_queue_packet. then finally, ctdb_tcp_queue_pkt is called to handel the real package queuing. 5. ctdb_tcp_queue_pkt queries the ctdb_tcp_node structure and calls ctdb_queue_send to queue the request on it. 6. then again queue_io_handler calls queue_io_write to do the actual network sending to remote CTDB daemon.

3	daemon 1 (node 1)	1. if the tcp connection isn't built yet, ctdb_listen_event will construct the connection and maintain it with a ctdb_incoming structure. 2. queue_io_handler_io calls queue_io_read to read package from tcp socket. and then it will finally arrive to ctdb_tcp_reade_cb this time instead of daemon_request_call_from_client. 3. ctdb_tcp_read_cb calls the ctdb level upcall: ctdb_recv_pkt. ctdb_recv_pkt calls ctdb_input_pkt. then ctdb_input_pkt will transfer the handling to the corresponding engine. it's ctdb_request_call for this case. 4. ctdb_request_call will handle the real data process (tdb database record fetch) and call ctdb_queue_package to queue the reply packge into the out_queue.
4	client 1 (possible) (node 1)	1. if it's a SRVID message request, the daemon might need call the client's message handler if this client registers it's handler for this type message.
5	daemon 0 (node 0)	1. then again queue_io_handler is called to transport the data to the remote peer upon TCP. 2. now this daemon gets the reply packge form the dmaster. queue_io_handler calls queue_io_read to read package from network, and then ctdb_tcp_read_cb is called. and finally ctdb_reply_call is called to handle the orignal request from client. 3. ctdb_reply_call first looks up the original request's state record (ctdb_call_state) from idr_tree, then copies the returned data to the original package and calls daemon_call_from_client_callback to put the reply package to daemon's queue. in the end, unlink the call state from ctdb_context's pending_calls list. 4. queue_io_handler will handle the package and send it to client

6	client 0 (node 0)	1. now on client side, queue_io_handler and queue_io_read are started and read data from the unix socket. and finally ctdb_client_read_cb will call ctdb_client_reply_call to copy results to original request's buffer and mark the original request's status as CTDB_CALL_DONE.

5.8 recovery monitor process

every ctdb daemon will start a recovery process during startup. but only the recovery master it can trigger a cluster recovery. firstly the recovery process on every node is to check the status of recovery master.

generally, the recovery process has 4 major tasks:

1. elect recovery master: it's a common task of every node
2. only recovery master will continue and do the followings
3. assure all active nodes agrees on the recovery master
4. assure all nodes are not in recovery mode
5. assure all the nodes' vnn map/node flags are consistent among ctdb cluster
6. if there's any inconsistency, start a cluster recovery
7. perform ip takeover when there's node banned or dead

5.9 election process

5.9.1 overview

the election is to elect a recovery master among all ctdb active nodes. the recovery master's role is important to monitor the ctdb cluster and play a recovery when the state is not consistent.

the election process will be triggered when ctdb cluster starts up or the recovery master node dies. and every ctdb node call issue an election request when it detects errors exist in cluster.

5.9.2 code logic

the recovery process is started by ctdb_start_recoverd(). It calls mointor_cluster() to check the recovery master status and issues election request via calling force_election if there's no an active recovery master.

recovery process is truely a client of ctdb daemon. so the communication model is the similar to client request process frame.

node 0 (request recovery master)			node 1 (normal node)			
recovery 0	<- unix socket ->	daemon 0	<- tcp ->	daemon 1	<- unix socket ->	recovery 1

election logic:

1	recovery 0	1. recovery process of this node that detects errors (no recovery master exists), will call force_election to issue a cluster election 2. it issues a SRVID message (CTDB_SRVID_RECOVERY) to all nodes electing itself as the new recovery master 3. the request will be passed to daemon 0
2	daemon 0	daemon receives the request from tcp network, and then will send this message to its recovery client since the recovery client already registers it's ownership on these messages
3	daemon 1	search the message handler and request it's recovery client to process the message
4	recovery 1	election_handler is to be called check the election package and check whether or not it agrees node 0's opinion. the earlier who starts the recovery process and the bigger vnn nuber how has will win the election. if this node doesn't agree the election request, it will issue another election request with itself assumed as the new recovery master
5	daemon 0	daemon 0 also receives the request sent in step 2 and it will call recovery 0 to handle this request
6	recovery 0	the original request doesn't request any reply. it just waits for an assumed timeout internal and checks again whether all ctdb nodes arrive to a final agreement on recovery master election. if there's no such an agreement, then issue a new election request: goto step 1

5.10 recovery process

5.10.1 core structure

```
struct ctdb_recoverd {
    struct ctdb_context *ctdb;
    uint32_t last_culprit;             /* the node causes recovery */
    uint32_t culprit_counter;          /* recovery times caused by
                                          this unstable node */
    struct timeval first_recover_time; /* last recovery time */
    struct ban_state **banned_nodes;   /* all banned nodes */
    struct timeval priority_time;      /* startup time, used for
                                          recovery master election */
};
```

5.10.2 recovery process

1. set recovery mode to active on all ctdb nodes
2. update vnn map and generation number

3. collect the database information from all nodes
4. copy the database information to all other nodes
5. copy the vnnmap and node flags to all other nodes
6. clear non completed records in all databases
7. do ip takeover
8. set recovery mode to normal mode
9. broadcast CTDB_SRVID_RECONFIGURE to notify all clients ctdb cluster is restored

5.11 ip takeover

currently all ip address release and takeover are done by recovery master . there are two interfaces to handle IP release and takeover.

5.11.1 core structure

```
struct ctdb_takeover {
    bool enabled;
    const char *interface;    /* ethernet interface for public address */
    const char *event_script; /* user script */
    TALLOC_CTX *last_ctx;
};
```

5.11.2 ip release

- control id: CTDB_CONTROL_RELEASE_IP
- handler: ctdb_control_release_ip
- description: the node who receives this request will first release the ip address from public interface via an upcall user script, then it will notify an event of CTDB_SRVID_RELEASE_IP to Samba daemon to force it exit. finally broadcast all Samba clients' ip addresses to all connected nodes.

5.11.3 ip takeover

- control id: CTDB_CONTROL_TAKEOVER_IP
- handler: ctdb_control_takeover_ip
- description: first upcalls event script to assign the new ip address and then try to restore all the broken connections to this node

5.11.4 tcp_list state maintaining

1. Samba: main() calls open_sockets_smbd to accept connections from CIFS clients
2. after child connection is built, message_ctdb_init is called to initialize ctdb client and register CTDB_SRVID_RELEASE_IP handler (to force client exit in case ip is to be released), send CTDB_CONTROL_TCP_CLIENT to ctdb daemon
3. ctdb daemon will handle the request of CTDB_CONTROL_TCP_CLIENT.
   it will add a tcp_list into ctdb_client's tcp_list, then broadcast CTDB_CONTROL_TCP_ADD to all connected nodes.
4. all ctdb nodes will get the request of CTDB_CONTROL_TCP_ADD and call ctdb_control_tcp_add to add a tcp_list into ctdb_context's tcp_list. so Samba client's ip address are stored on every CTDB node now.
5. when a client exits, ctdb_client_destructor will issue CTDB_CONTROL_TCP_REMOVE to notify the client's removal to all CTDB nodes.

5.11.5 takeover flow during recovery

when recovery master determines that ip takeover is needed, it will call ctdb_takeover_run to do ip takeover:

1. notify all other nodes except the selected node to release the ip address (calling ctdb_control_release_ip)
2. if the "dead" node isn't dead yet, Samba client will be forced to exit (callback routine msg_release_ip handles it)
3. then look up a suit node in nodes array within the same subnet as a candidate to takeover the dead node's ip address and command the selected node to take over the ip (calling ctdb_control_takeover_ip)
4. the selected node receives the message and upcalls the event script to assign new ip. after ip assignment succeeds, takeover_ip_callback is triggered to handle all Samba clients' connections rebuilding: send ACK to Samba clients to keep all the connections alive, though the Samba server is already changed to a new node.

6 ip reassignment design

pCIFS needs a more flexible ip release and reassign mechanism to address the ip reassign from one CTDB node to another when the secondary Lustre node replaces the dead node after Lustre failover. the new feature of ip reassign can

assure pCIFS i/o to be banlanced among different nodes rather than a single node with several ip addresses.

ip reassignment is actually an ip takeover operation. we are planning to this task done by recovery master to ease the unnecessary recovery handling. let's take a Lustre failover case for example:

1. a Lustre node dies
2. CTDB and Lustre failover will be triggered.
3. CTDB failover is a quick process, should be completed in no time. so the ip will be taken over by another CTDB node.
4. Lustre secondary node will be started by then heartbeating or other monitor program.
5. Lustre cluster restores from recovery, the new node will join CTDB cluster too (to be described in next chapter)
6. the new node is ready to service as a CTDB node, then requests the original ip (CTDB_SRVID_REQUEST_IP) from recovery master
7. recovery master receives the ip request and then call all nodes to release the specified ip address and then let the new node to takeover the ip.

if one node dies unexpectedly, the recovery process can deal all the cases:

1. if the new node dies, recovery master can reassign the ip addresses to another
2. if any other node ides, current takeover process won't be bothered and will continue to a final success
3. if recovery master dies, then there will be a new election and recovery process. during the recovery process, the new node will be treated as the owner of public ip address, so there won't be any conflicts.

CTDB_SRVID_REQUEST_IP is a new request type to current CTDB, to be handled (only) by recovery master. the request package should contain the public ip address and vnn number of the requestor node. no reply package is needed to the request.

the recovery process need register a callback to handle this request:

```
static void monitor_cluster(struct ctdb_context *ctdb) {
   ...
   /* register a message port for ip requestor */
   ctdb_set_message_handler(ctdb,
                            CTDB_SRVID_REQUEST_IP,
                            request_ip_handler,
                            rec);
```

```
   ...
}
static void request_ip_handler(
                struct ctdb_context *ctdb,
                uint64_t srvid,
                TDB_DATA data,
                void *private_data) {
    struct ctdb_recoverd *rec = talloc_get_type(private_data,
                                                struct ctdb_recoverd);
    struct ctdb_context *ctdb = rec->ctdb;
    struct ctdb_request_ip *r = (struct ctdb_request_ip *)data.dptr;
    int    rc, vnn;

    /* query nodes map */
    rc = ctdb_ctrl_getnodemap(ctdb,...)
    /* release the specified public ip address */
    for (i=0;i<nodemap->num;i++) {
        ...
        rc = ctdb_ctrl_release_ip(ctdb, ..., &ip);
        ...
    }
    /* let's the requstor takeover the ip */
    rc = ctdb_ctrl_takeover_ip(ctdb, ..., r->vnn, &ip);

    /* find a suit node to takeover the ip in case the requstor
       fails to take over it */
    for (vnn = (r->vnn + 1) % (total_vnn_num); rc != 0; ...) {
        rc = ctdb_ctrl_takeover_ip(ctdb, ..., vnn, &ip);
    }
    return rc;
}
```

7 nodes management design

7.1 overview and requirements

currently all nodes are loaded from CTDB config files and stay fixed in ctdb_context. removing a node is easy, since the recovery process can reconfigure the whole ctdb cluster. but there's no way to add a new node into a working CTDB cluster. then the CTDB cluster will become smaller and never get a chance to grow, so Lustre parallel i/o service will be finally centralized to several nodes. thus bottlenecks will emerge among the remained nodes. that's why pCIFS needs this feature to dynamically add a new node into a working ctdb cluster:

generally, there are at least two cases to import a new CTDB node:

1) when adding a new node (Lustre OST server) into current cluster

2) the standby node restores after a Lustre failover.

when adding a new node into current CTDB cluster, we might need reassign the public ip address from one CTDB node to another. in this chapter we only discuss how to add/remove a node, the ip reassign issue is already discussed in previous chapter.

the design also need address the following two major issues:

1) possible races between concurrent processes (like CTDB daemon and recovery process)

2) how to deal with the remaining half in case a recovery starts during node addition/removal

7.2 adding a node

7.2.1 new request types

- CTDB_SRVID_NODE_ADD: send to recovery master to request new node addition
- CTDB_SRVID_NODE_ADDED: to be broadcasted to all connected nodes to notify of the successful node additinn
- CTDB_CONTROL_NODE_ADD: send to daemon from recovery process to add/set a new node to (ctdb_context)

7.2.2 automatic transport connection

transport connection and request dispatching are highly tied to one of core structures: ctdb_tcp_node. this structure is allocated during startup in ctdb_tcp_initialise (transport initialization). transport connections to any other CTDB nodes are built in ctdb_tcp_start (called by ctdb_start_transport) during startup.

when a node is added, we need replay the startup process to allocate a new ctdb_tcp_node and initialize it's i/o queue, then build a connection to the newly added node. the whole procedure can be done manually when we detects new node addition or let it done in ctdb_queue_packet. the other modification is to let ctdb_tcp_start only initialize one node's connection instead of all.

```
static int ctdb_tcp_start(struct ctdb_context *ctdb,
                          struct ctdb_node *node)
{
        /* startup connection to this node - will happen on next loop */
        struct ctdb_tcp_node *tnode = talloc_get_type(
                         node->private_data, struct ctdb_tcp_node);
        if (!ctdb_same_address(&ctdb->address, &node->address)) {
            event_add_timed(ctdb->ev, tnode, timeval_zero(),
                            ctdb_tcp_node_connect, node);
        }
        return 0;
```

```
}
void ctdb_queue_packet(struct ctdb_context *ctdb,
                       struct ctdb_req_header *hdr)
{
   ...
   node = ctdb->nodes[hdr->destnode];
   /* initialize node's transport engin if it's not initialized */
   if (NULL == node->private_data) {
       if (ctdb->methods->add_node(node)) {
           ctdb_fatal(ctdb, "Unable initialize transport node\n");
       }
       if (ctdb->metods->start(ctdb, node)) {
           ctdb_fatal(ctdb, "Unable build transport connection\n");
       }
   }
   if (hdr->destnode == ctdb->vnn) {
       ctdb_defer_packet(ctdb, hdr);
   } else {
       node->tx_cnt++;
       if (ctdb->methods->queue_pkt(node,
                                    (uint8_t *)hdr,
                                    hdr->length)
           != 0) {
           ctdb_fatal(ctdb, "Unable to queue packet\n");
       }
   }
}
```

7.2.3 code logic

1. when starting a new ctdb node, we need specify a delegate parameter with the ip addresses of any nodes in current CTDB cluster which we want the new node to join in. for example:
   ctdbd –delegate=192.168.0.1,192.168.0.2 ...
   the first delegate node is 192.168.0.1, the second candidate is 192.168.0.2

2. the new node will do normal startup and load it's own ip addresses and other settings from config files.

3. in monitor_cluster of it's recovery process, it will try to connect to the delegate node instead of doing normal monitoring.

4. then send a request of CTDB_SRVID_NODE_ADD to the delegate, with it's own settings packed into the request package.

5. the new node will wait for response upon a timeout event

6. the delegate will transfer the request to the recovery master. then the delegate's task is done.

7. the recovery master gets the node insertion request from the delegate node and get the service handler (handler_node_add) called to process this request. the handler callback will first check current nodes map whether or not the new node already exists. if the node already exists it just return

8. add the new node structure into the daemon's ctdb_context (CTDB_CONTROL_NODE_ADD). the daemon will allocate a new ctdb_node structure and reallocate ctdb_context->nodes pointer array

9. recovery copies all nodes map and vnn map to the new ctdb nodes (SET_NODESMAP/SET_VNNMAP)

10. the new node will update the latest nodes/vnn map from recovery master, then build transport connections to these nodes

11. recovery master copies all databases to the new target node

12. the new node update it's tdb databases

13. now the new node addition is completed. recovery master then broadcasts CTDB_SRVID_NODE_ADDED to all active nodes

14. recovery master can go to the normal recovery monitor process

15. all nodes include the new, recovery master and other nodes will receive the CTDB_SRVID_NODE_ADDED message. then handler routine handler_node_added will add the new node into ctdb_context's array and then initialize a tcp connnection to the new node. for recovery master, the node is already added, so this task will be skipped. for the new node, the handler callback is only to cancel the timeout event (step 5).

16. if the new node gets timeout, it will try next delegate node in parameter list

17. after the node successfully joins into the CTDB cluster, it need issue a new request of CTDB_SRVID_REQUEST_IP to takeover it's own ip address from another CTDB node.

7.2.4 issues

1. possible access conflict between concurrent processes (daemon, recovery, client)
   we use CTDB_SRVID_XXX request instead of CTDB_CONTROL_XXX, etc. the handling of CTDB_SRIV_XXX request is done in recovery process, then the normal recovery monitor process is to be blocked. and any request from recovery process to daemon will block daemon's thread too. this mechanism can serialize most of the critical operations synchronously. the communication between CTDB daemon and CTDB client (ex: Samba) won't induce races to what we concern during node addition.

2. if recovery master dies before step 13, then new CTDB node will time out. it will try to connect again or try next delegate node. it might time out again while election is done during CTDB cluster, but it won't bother.

3. if one normal CTDB node dies, a cluster recovery will be performed after the new node is added.

4. if the new node dies, cluster recovery will be triggerred too. but we can avoid such a costing recovery before step 13 by removing the new node smartly. we could also let CTDB recovery to handle the dead nodes cleanup.

7.3 removing a node

7.3.1 overview and requirements

this part can be optional since we can remove a node easily via BAN or just unplug it from network etc. implementation of node removal is to make interfaces set complete. we can refer this part as a forcibly removing method, comparing to BAN or unplug. actually after all public ips and client connections migrated to other nodes, this node only acts as an observer rather than a member of the whole CTDB cluster.

7.3.2 requests to remove a node

- CTDB_SRVID_NODE_REMOVE: be broadcasted to all (from any node) to forcely remove a node
- CTDB_CONTROL_NODE_REMOVE: send to daemon by it's recovery process to remove a node from ctdb_context

7.3.3 code logic

1. user issues CTDB_SRVID_NODE_REMOVE with ctdb utility on any node

2. all CTDB nodes will call handler_node_remove to process this request in the recovery process

3. recovery process will block normal monitor process and call CTDB_CONTROL_NODE_REMOVE to daemon

4. daemon will cleanup all the contexts of the node (ctdb_node, ctdb_tcp_node, queue, pending requests), then return to recovery process

5. the recover process is to be waken and then do normal monitor process

8 future improvement

8.1 structure re-arrangement

ctdb_context is commonly used by both client and daemon. it's better to keep only all the common structures in ctdb_context and pick out the specific members for ctdb_daemon, ctdb_client or ctdb_recoverd

8.2 recovery mechanism enhancement

1. store context/state in tdb database instead of tcp/broadcasting, since tdb database should be more reliable than transport messages (without replies).for example, client tcp connection session can be stored in a database instead of the ctdb_context structure.

2. dead node cleanup during recovery process: clean all dead nodes rather than mark it as disconnected

8.3 minor bugs

1. ctdb->ev re-initialization in ctdb_cmdline_init and ctdb_start_daemon. the former is overridden.

2. timed events won't work if there's no fde objects.

```
static int std_event_loop_wait(struct event_context *ev) {
    ......
    /* should check both ev->te and std_ev->fd_events */
    while (std_ev->fd_events && std_ev->exit_code == 0) {
        if (std_event_loop_once(ev) != 0) {
            break;
        }
    }
    ......
}
```

9 References

1. http://ctdb.samba.org

2. http://arch.lustre.org/index.php?title=CTDB_with_Lustre