Open Software Foundation
Request For Comments: 76.0
May 1995

D. Clevenger (Transarc)

MULTI-THREADING THE DFS REPLICATION SERVER

1. INTRODUCTION

The goal of this work is to modify the DFS replication server, `repserver`, to allow for more than one thread of control for managing local replicas. As a particular `repserver` is responsible for managing more local replicas (because more replication sites are added to the associated server), the administrator will be able to increase the number of `repserver` threads, keeping the level of service at some desired level.

The current `repserver` only allows for one thread of control, so, as more load is placed on the `repserver`, it quickly becomes unable to meet the demands for its service. This is particularly undesirable since the DFS replication scheme is highly dependent on replicas being updated in a timely fashion.

2. CURRENT IMPLEMENTATION

The current `repserver` has some of the features necessary to make it multi-threaded. It has some locks defined for global data structures, but in many cases the use of locks is inadequate to perform correctly in a multi-threaded environment. Furthermore, the basic flow of control is not conducive to having multiple threads operating concurrently. For these reasons the number of threads is statically set to 1. Each thread follows an algorithm roughly like this:

```
for (;;) {
    unsigned int    t[5];  /* Array of time values */

    MakeCleanKills();
    t[0] = RenewTokens();
    t[1] = StartImporting();
    t[2] = ForceKeepAlives();
    t[3] = DoWillCalls();
    t[4] = ExpireVolChanges();

    sleep(min(t[0..4]) - current_time);
}
```

Many of the individual actions here (`MakeCleanKills()`, `RenewTokens()`, etc.) follow a structure something like this:

```
ReadLock(rep_table_lock);
foreach replica in rep_table {
    WriteLock(replica.lock);
    if (replica needs attention) {
        Update some replica state;
        Unlock(replica);
        SomeRPC(replica.server, replica);
        WriteLock(replica.lock);
        Update some more replica state;
    }
    Unlock(replica.lock);
}
Unlock(rep_table_lock);
```

One problem with this model is that the threads will tend to ''pile up'' trying to get the write lock on a replica structure that some other thread is servicing. Furthermore, this will make it very likely that some other thread will lock the replica structure as soon as the original thread drops the lock to make an RPC. Finally, the `rep_table_lock` is held across almost every RPC. Even though the `rep_table_lock` is usually only read-locked, this seems bad.

Another major problem with the current implementation is that there are no locks protecting the shared connection structures, so there is no chance of the current `repserver` operating correctly with multiple threads.

3. NEW IMPLEMENTATION

To remedy the problems of the current implementation, we are proposing a new threading model with the following basic properties:

(a) A pool of *N* `rep` threads for per-replica processing, where *N* is specified by the `-mainprocs` command-line switch to the `repserver`.

(b) A `reaper` thread that will destroy and/or clean up moribund replica filesets and structures.

(c) A `background` thread for keep-alive and will-call processing.

We want to get away from the current model where the service thread loops through the list of all replicas looking for work. To accomplish this, we will implement a priority queue where each replica structure (representing one local replica) is stored in the queue according to time at which it next requires service. Each of the rep threads will block trying to get a replica from the priority queue, and will be unblocked when it comes time to service a replica. When unblocked each thread will be responsible for servicing the needs of only one replica.

3.1. Priority Queue

The priority queue will be implemented as an abstract data type that is not particularly specific to the needs of the `repserver`. If its functionality proves useful, it could easily be used in other applications. The interface definition follows:

```
typedef void * rep_queue_cookie_t;

/*
 * The repserver specific data for a queue item
 */
typedef struct rep_queue_item {
    struct rep_queue_item *rqi_next;
    time_t                 rqi_deadline;
    unsigned               rqi_flags;
    void                   *rqi_data;
} rep_queue_item_t;

/*
 * The definition of a priority queue
 */
typedef struct rep_queue {
    pthread_mutex_t        rq_mutex;
    pthread_cond_t         rq_hasData;
    struct rep_queue_item *rq_head;
} rep_queue_t;
```

The are two functions used for initializing the priority queue package. The first,

```
void
repq_InitICL(
    struct icl_set *iclSetP
);
```

initializes the ICL set for debugging the `repq` module.

The second,

```
void
repq_Init(
    rep_queue_t *rqP
);
```

initializes the priority queue specified by `rqP`.

When a new entry is to be initially inserted onto a queue,

```
rep_queue_cookie_t
repq_Enter(
    rep_queue_t *rqP,
    void        *data,
    time_t       deadline
);
```

allocates and places an entry on the queue for the first time. The queue specified with the `rqP` is maintained in earliest-to-latest-deadline order. The internal queue item stores the `data` pointer and `deadline` and returns a `rep_queue_cookie_t` for passing to other priority queue routines.

To permanently remove an item from the priority queue, use

```
void
repq_Delete(
    rep_queue_t        *rqP,
    rep_queue_cookie_t  cookie
);
```

If the entry represented by `cookie` is in use, the deletion is deferred until the next call to `repq_Put()`.

The item at the head of the queue is retrieved with

```
void *
repq_Get(
    rep_queue_t        *rqP,
    rep_queue_cookie_t *outCookieP
);
```

if its deadline has passed; otherwise, `repq_Get()` sleeps until an item's deadline occurs. The internally stored data pointer is returned, and a pointer to the queue item is placed in the out parameter `outCookieP`.

If the caller wants to put an item back on the priority queue, then

```
void
repq_Put(
    rep_queue_t       *rqP,
    rep_queue_cookie_t cookie,
    time_t             deadline
);
```

places the item back on the queue with a new deadline. The actual deadline is set to the earlier of `deadline` and any deadline set with `rep_ResetDeadline()` since the item was retrieved with `repq_Get()`. If the item represented by `cookie` was marked for deferred deletion, it is deleted at this time instead.

An item's deadline may be reset by calling

```
void
repq_ResetDeadline(
    rep_queue_t       *rqP,
    rep_queue_cookie_t cookie,
    time_t             newDeadline);
```

with a new deadline. This routine can be called at any time, regardless of whether the item represented by `cookie` is currently on the queue or not. The actual deadline is only set to the new value if the current deadline is later than `newDeadline`. If the item represented by `cookie` is currently on the queue and now has an earlier deadline, it may be necessary to reorder the queue.

Using the priority queue interface described above, each `rep` thread will now follow an algorithm that looks abstractly like this:

```
for (;;) {
    time_t        t[3];  /* Array of time values */

    replica = repq_Get(rep_queue, &cookie);
    WriteLock(replica.lock);
    if (replica destroyed) {      /* Someone did a rmsite */
        Unlock(replica.lock);
        repq_Delete(rep_queue, cookie);
        WriteLock(rep_table_lock);
        Remove replica from rep_table;
        Unlock(rep_table_lock);
        Release(replica);         /* Once for rep_queue */
        Release(replica);         /* Once for rep_table */
        /*
         * The last release will enter replica on the
         * death_queue, another priority queue that is
         * managed by the reaper thread.
         */
    }
    else {
        t[0] = RenewTokens(replica);
        t[1] = StartImporting(replica);
        t[2] = ExpireVolChanges(replica);
        Unlock(replica.lock);
    }
    /*
     * If the replica was deleted the next action time is
     * irrelevant, since this call will just free up the
     * queue item.
     */
    repq_Put(repq_queue, cookie, min(t[0..2]));
}
```

3.2. Host Connection Management

One of the main deficiencies with the current design is in its management of connection structures. To remedy this, a new module will be introduced to manage these shared objects in a safe and correct manner. The interface definition for this module is as follows:

```
/*
 * Connection types
 */

typedef enum rep_host_conn_type {
    REPH_CONN_TYPE_FS,          /* Fileserver connection type */
    REPH_CONN_TYPE_FT           /* Ftserver connection type */
} rep_host_conn_type_t;

/*
 * A server connection
 */

typedef struct rep_host_conn {
    /*
     * The reference count is protected by rhc_mutex.
     */
    pthread_mutex_t      rhc_mutex;
    int                  rhc_refCount;

    /*
     * These members are not really protected by any lock, but
     * they are guaranteed not to change as long as there is a
     * reference to this structure.
     */
    rep_host_conn_type_t rhc_type;
    handle_t             rhc_handle; /* RPC binding handle */
} rep_host_conn_t;

/*
 * A host structure for cacheing server connections
 */

typedef struct rep_host {
    /*
     * The following members are protected by the global
     * hostTableLock.
     */
    struct rep_host     *rh_next;
    unsigned             rh_refCount;

    /*
     * The following members are not really protected by any
     * lock, but  they are guaranteed not to change as long as
     * there is a reference to this structure.  Note that the
     * host table itself holds a reference.
     */
    struct afsNetAddr   rh_addrs[ADDRSINSITE];
    unsigned            rh_numAddrs;
    char                rh_name[MAXKPRINCIPALLEN];
```

```
    /*
     * Fileserver connection information.
     * The following members are protected by rh_fsMutex.
     */
    pthread_mutex_t     rh_fsMutex;
    unsigned            rh_fsFlags;
    pthread_cond_t      rh_fsSetContextDone;
    rep_host_conn_t    *rh_fsConnP;
    time_t              rh_fsLastSuccess;
    unsigned            rh_fsHostLifeGuarantee;
    unsigned            rh_fsHostLeadTime;
    afsUUID             rh_fsUUID;
    /*
     * Ftserver connection information.
     * The following members are protected by rh_ftMutex.
     */
    pthread_mutex_t     rh_ftMutex;
    rep_host_conn_t    *rh_ftConnP;
} rep_host_t;

/*
 * Fileserver flag bits for rh_fsFlags
 */
#define REPH_FS_FLAG_DOING_SET_CONTEXT           0x1
#define REPH_FS_FLAG_AWAITING_SET_CONTEXT        0x2
```

The `repserver` host subsystem is initialized by calling

```
void
reph_Init(
    const char       *cellName,
    unsigned32        revokeEpoch,
    const afsNetData *revokeCallbackAddr,
    struct icl_set   *iclSetP
);
```

The `revokeEpoch` parameter is the start time of this `repserver` and is passed as one of the parameters to `AFS_SetContext()`. The `revokeCallbackAddr` parameter is also passed to `AFS_SetContext()` for token revocation RPC calls.

```
rep_host_t *
reph_GetHost(
    struct afsNetAddr *addrs,
    unsigned           numAddrs,
    char              *hostName
);
```

gets a host structure for the host with the given addresses. This function may return `NULL` if there is no memory available for the new host structure. A reference to this host structure will be held for the caller. When a new host structure is allocated, the table of host structures also holds an additional reference.

```
void
reph_PutHost(
    rep_host_t *hostP
);
```

drops a reference to a host that was returned by `reph_GetHost()`.

To retrieve a connection for a given host,

```
long
reph_GetConn(
    rep_host_t           *hostP,
    rep_host_conn_type_t connType,
    rep_host_conn_t    **connPP
);
```

gets the connection to the host represented by `hostP`. The `connType` argument specifies the type of server for the connection. The current choices are `fileserver` and `ftserver` connections. If an error occurs while trying to set up a connection, `NULL` will be returned in `*connPP` and the return code will be set to a non-zero value. After receiving a successful connection, the caller should make RPCs using the `rhc_handle` member. When finished with the connection, the caller should release it by calling `reph_PutConn()`.

A reference to the connection will be held for the caller when there is no error. Also, when a new connection is created, an additional reference is held for the host structure in `hostP`.

When an error occurs using a connection, calling

```
long
reph_ResetConn(
    rep_host_t       *hostP,
    rep_host_conn_t **connPP,
    long              connError
);
```

will attempt to reset the connection. The `connError` parameter should contain the error code from the failed operation.

The return values will be:

(a) 0 — `connError` indicates a condition which may be correctable with a reset, and the reset succeeded. The user can retry the operation that failed. When the user is finished with the connection, they should release it by calling `reph_PutConn()`.

(b) `REP_ERR_RESETFAILED` — `connError` indicates a condition which may be correctable with a reset, but the reset failed. The value of `*connPP` may be set to `NULL`. If so, do not call `reph_PutConn()` on it. Currently, the only case where `REP_ERR_RESETFAILED` is returned without setting `*connPP` to `NULL` is when `connError` is `rpc_s_auth_tkt_expired` and the attempt to refresh the local auth context failed.

(c) `connError` — The return of `connError` indicates a condition which is not correctable by resetting the connection, so the same error as was passed is returned. In this case, it is still necessary to release the connection by calling `reph_PutConn()`.

A reference to a connection is dropped by calling

```
void
reph_PutConn(
    rep_host_conn_t *connP
);
```

This reference should have been obtained from `reph_GetConn()` or `reph_ResetConn()`.

After a successful RPC call to a `fileserver`, use

```
void
reph_MergeSuccess(
    rep_host_t *hostP,
    time_t      successTime
);
```

to update the time `rh_fsLastSuccess` field of the host structure given by `hostP`. The time in `successTime` is only stored if it is more recent than the currently stored success time.

The number of hosts in the table is obtained with

```
unsigned
reph_GetNumHosts(
    void
);
```

4. TESTING

To determine whether the multi-threaded implementation meets the above goal, we plan to develop a test that will stress the `repserver` by having it manage a number of local replicas, all of which will be in constant need of updating due to frequent changes to their source filesets. The same test will be run on both the current single-threaded `repserver`, as well as the improved multi-threaded version. The test will report a running percentage that will represent the amount of time that the test replicas are kept up to date, according to their replication parameters. We expect a significant gain in the multi-threaded case.

Besides verifying that adding multiple threads helps, this test will put significant stress on the multi-threading code, so it will be useful as a debugging tool. Finally, the test will provide confidence in the new code as continuous-hours-of-operation statistics increase to reasonable levels.

5. ACKNOWLEDGEMENTS

Jeff Prem (then at Transarc, now at FORE Systems) was instrumental in early drafts of this document.

AUTHOR'S ADDRESS

Daryl Clevenger
Transarc Corp.
707 Grant Street
Pittsburgh, PA 15219
USA

Internet email: dlc@transarc.com
Telephone: +1-412-338-4426