Open Software Foundation **S. Moyer (Transarc)**
Request For Comments: 77.0
January 1996

SUPPORTING MULTI-HOMED DFS SERVERS

1. INTRODUCTION

This RFC describes a method for supporting multi-homed servers[1] in the DFS cache manager (CM). The goal of this work is to enhance DFS's fault-tolerance by enabling CMs to communicate with a server via any of a set of specified addresses.

An appropriate implementation of multi-homed server support will have (at least) the following properties:

(a) CMs handle failures in network connections transparently to the extent possible.

(b) CMs actively manage knowledge of the state of a server's network connections.

(c) CMs obey the server-preference scheme.

The following sections discuss the current CM implementation with respect to managing server communication, and propose enhancements to support multi-homed servers.

2. CURRENT CM COMMUNICATION ARCHITECTURE

The cache manager employs two primary object types for supporting CM-server communication: the **server object**, implemented by the server module `cm/cm_server.{h,c}`, and the **connection object**, implemented by the connection module `cm/cm_conn.{h,c}`. The CM represents the state of a server as a `cm_server` structure, with various connections (bindings) to that server represented as a list of `cm_conn` structures hanging off the server structure. A `cm_server` structure contains, among other state information, a single address that is used to create bindings for that server.

When the CM needs to perform a remote procedure call (RPC), one of three functions can be employed to establish a server binding: `cm_Conn()`, `cm_ConnByMHosts()`, or `cm_ConnByHost()`. The only one we will consider here is `cm_ConnByHost()`, since it is this function that actually establishes a binding, and since it is used to implement the other two.

The standard method for making an RPC in the CM is illustrated by the following fragment of pseudo code:

```
do {
    if (connp = cm_ConnByHost(serverp, ...)) {
        /* got server binding */
        result = server_RPC(connp->connp, ...);
    }
} while (cm_Analyze(connp, result, ...));
```

Essentially, a connection is requested and, if obtained, the RPC is performed. The results of the `cm_ConnByHost()` and the RPC (if any) are then passed to `cm_Analyze()`. If the RPC was not performed successfully, for whatever reason, then `cm_Analyze()` determines if it should be attempted again. Note that it is this methodology, when employed in conjunction with `cm_ConnByMHosts()`, that allows the cache manager to contact alternative servers to perform an operation; e.g., to contact an alternative fileset location server when the preferred server can not be reached.

1. A multi-homed server is a server that resides on a host that has two or more network connections.

Though the current CM communication architecture cleanly handles non-persistent errors, and the use of alternative servers when appropriate, it does not support multi-homed servers. Recall that only a single address is stored in a `cm_server` structure. The next section details enhancements that support multiple addresses per server.

3. ENHANCED CM COMMUNICATION ARCHITECTURE

To support multi-homed servers, it is proposed that the cache manager employ a third object type for CM-server communication: the **site object**, implemented by the site module `cm/cm_site.{h,c}`. A site object represents the state of a server's network connections on a multi-homed host (site).

Before the details of the site object are presented, it is useful to see how it will be employed in the generic RPC methodology illustrated above.

In the current CM architecture, certain failure conditions cause either `cm_ConnByHost()` or `cm_Analyze()` to call `cm_ServerDown()` to indicate that a server object appears to be down. The result of `cm_ServerDown()` indicates either that the server is truly down, in which case it is flagged as such, or that the server can be retried.

In the enhanced CM architecture, a communications failure will cause `cm_ServerDown()` to call the new function `cm_SiteAddrDown()` to indicate that a particular server address is down. The function `cm_SiteAddrDown()` performs a fail-over to the next best available address for that server, if one exists. Thus, in the case of a communications failure, `cm_ServerDown()` can indicate that the server can be re-tried given a successful fail-over.

3.1. Site Object Definition

The site object type will be implemented by the site module `cm/cm_site.{h,c}`, which will export the following data structures and functions.

3.1.1. Data structures

```
/* site address declaration */
struct cm_siteAddr {
  struct cm_siteAddr  *next_sa;       /* next site addr */
  struct cm_siteAddr  *next_sahash;   /* next addr in hash */
  struct cm_site      *sitep;         /* associated site */
  struct sockaddr_in  addr;           /* address */
  ushort              rank;           /* address rank */
  ushort              state;          /* address state */
};

/* site object declaration */
struct cm_site {
  struct cm_site      *next_sitehash;   /* next site in hash */
  osi_dlock_t         lock;             /* site-data lock */
  struct cm_cell      *cellp;           /* cell */
  struct cm_siteAddr  *addrListp;       /* address list */
  struct cm_siteAddr  *addrCurp;        /* best ranking up
                                           addr */
  ulong               addrUpdateTime;   /* time addr list
                                           updated */
  struct cm_server    *serverp;         /* associated server */
  ushort              state;            /* site state flags */
};

/* note: site identified via (cellp, serverp->principal) pair;
   should be replaced by a UUID in the future */
```

The CM will represent a server's host as a cm_site structure, with the state of the server's network connections represented as a list of cm_siteAddr structures hanging off the site structure. Exactly one site object will exist for each server object in the system.

Maintaining per-server address lists, and hence per-server address ranks and states, enables both the server-preference scheme (discussed later) and the communications fail-over mechanism to be implemented in a server-centric fashion. This architecture also accommodates the current model whereby server addresses are specified individually[2] and are obtained from different sources; specifically, the FLDB for file-servers/repservers and the CDS for flservers.

> *NOTE FOR FUTURE DEVELOPMENT: Maintaining per-server site information also enables a CM to communicate with multiple DFS servers of the same type residing (at least logically) on a single machine; e.g., to help support access to DFS servers on machines hidden behind a firewall.*

All cm_siteAddr structures are maintained in a hash table hashed by address so that associated site and server structures can be quickly located. The primary purpose of hashing by address is to support the server-preference scheme.

3.1.2. Functions

```
/* cm_SiteAlloc() -- allocate a new site object.
 *
 * Parameters:
 *     siteCellp - host's cell
 *     siteNamep - host's principal name
 *     serverp   - associated server object
 */
struct cm_site *cm_SiteAlloc(struct cm_cell *siteCellp,
                             char *siteNamep,
                             struct cm_server *serverp);

/* cm_SiteAddrUpdate() -- replace list of server addresses with
 *     the list provided.
 *
 *     As a side effect, sets site fields addrList and addrCur
 *     appropriately.
 *
 * Parameters:
 *     sitep    - site object
 *     addrvp   - server-address vector
 *     addrvcnt - server-address vector size
 *
 * Returns:
 *      0 - update successful
 *     -1 - invalid arg; addr list unchanged
 */
int cm_SiteAddrUpdate(struct cm_site *sitep,
                      struct sockaddr_in *addrvp,
                      int addrvcnt);

/* cm_SiteAddrUpdateAllFLDB() -- for all
 *     file-servers/repservers on the specified host, replace
```

2. File-server/repserver address lists are actually specified in pairs.

```
 *     list of server addresses with the full list in the FLDB.
 *
 *     As a side-effect, sets site fields addrList and addrCur
 *     appropriately for all relevant servers.
 *
 * Parameters:
 *     siteCellp - host's cell
 *     siteNamep - host's principal name
 *     addrp     - a known address for host
 */
void cm_SiteAddrUpdateAllFLDB(struct cm_cell *siteCellp,
                              char *siteNamep,
                              struct sockaddr_in *addrp);

/* cm_SiteAddrSetRankAll() -- for all servers of the specified
 *     type, assign rank to the specified address.
 *
 *     Note that server types SRT_FX and SRT_REP are treated as
 *     equivalent; specifying either updates address rank for
 *     both.
 *
 *     As a side-effect, sets site fields addrList and addrCur
 *     appropriately for all relevant servers.
 *
 * Parameters:
 *     addrp - address
 *     rank  - rank
 *     svc   - server type (SRT_FX, SRT_REP, SRT_FL)
 *
 * Returns:
 *      0 - server address rank set
 *     -1 - failed; address not found
 */
int cm_SiteAddrSetRankAll(struct sockaddr_in *addrp,
                          int rank,
                          int svc);

/* cm_SiteAddrDown() -- report failed communication to server
 *     address; perform address fail-over.
 *
 *     A successful address fail-over updates the site field
 *     addrCur to point to the best up address for the server.
 *
 *     An unsuccessful fail-over occurs when all server
 *     addresses are marked down; in this case the site field
 *     addrCur is set to the (down) address addrList.
 *
 * Parameters:
 *     sitep  - site object
 *     addrp  - server address
 *
 * Returns:
 *      0 - successful address fail-over
 *     -1 - unsuccessful address fail-over; all server
 *          addresses down
```

```
 */
int cm_SiteAddrDown(struct cm_site *sitep,
                    struct sockaddr_in *addrp);

/* cm_SiteAddrUp() -- mark all server addresses as up.
 *
 *     As a side-effect, sets site field addrCur to addrList.
 *
 * Parameters:
 *     sitep - site object
 */
void cm_SiteAddrUp(struct cm_site *sitep);

/* cm_SiteAddrPingBad() -- ping server on all addresses marked
 *     down.
 *
 *     Attempts to ping server on all addresses marked down to
 *     determine if the connection has been restored.
 *     Successful pings result in the address being marked up.
 *
 *     As a side-effect, sets site field addrCur appropriately.
 *
 * Parameters:
 *     sitep - site object
 */
void cm_SiteAddrPingBad(struct cm_site *sitep);
```

The above functions provide a simple interface for creating site objects and manipulating their state.

3.2. Site Object Integration

Site objects will be integrated into the existing CM code base in a straight-forward fashion, with most modifications concentrated in the server module. Below is a discussion of the significant modifications that will take place; many minor updates are required that will not be presented in this document.

3.2.1. Server module modifications

The `cm_server` structure in the file `cm/cm_server.h` will be modified as follows:

(a) `struct cm_server *nextUUID` — Remove.

(b) `struct cm_cell *cellp` — Remove.

(c) `struct sockaddr_in serverAddr` — Remove.

(d) `u_short rank` — Remove.

(e) `struct cm_site *sitep` — Add.

(f) `int failoverCount` — Add.

Data fields removed from the server structure are subsumed by equivalent fields in the site structure now referenced by `sitep`. The hash chain link `nextUUID` becomes obsolete as it will no longer be necessary for the server module to link `cm_server` structures into two hash tables, one hashed by IP address (`cm_servers[]`) and the other by server UUID (`cm_serverUUID[]`). Instead, the server module will maintain a single hash table (`cm_servers[]`) with server entries hashed by UUID; the hash link employed will be the existing `next` pointer, since it is already used throughout the CM code to scan through all server structures. The new field `failoverCount` represents the number of address fail-overs since the last successful server RPC; it is used for fail-over throttling as discussed later.

Though many of the functions in the file `cm/cm_server.c` will be modified, most are simple updates to access data items now stored in the associated `cm_site` structure. The functions requiring significant operational modifications are the following:

(a) `cm_GetServer()` — This function locates/allocates a server object. It will be modified to call `cm_SiteAlloc()` when a new server object is allocated. Server structures will be placed in a hash table hashed by UUID only.

(b) `cm_ServerDown()` — This function is called to report that a server appears to be down. As previously discussed, this function will be modified to perform address fail-over by calling the new function `cm_SiteAddrDown()` to report a failed communication. The result of address fail-over, and the value of `failoverCount`, can then be taken into account when determining if the server is ‘‘really’’ down.

(c) `CheckDownServer()` — This function pings a server marked down to determine if it has come back up. It will be modified to ping each server address in rank order until successful or all addresses have been tried.

(d) `cm_SetServerRank()` — This function is called to set the rank value of a server (server address). The function will be modified to call `cm_SiteAddrSetRankAll()` to do the actual work. A discussion of the use of address ranks in the server-preference scheme for multi-homed servers is presented in a later section.

3.2.2. Other modifications

The impact on other CM modules of employing site objects is minimal, with the major changes summarized as follows:

(a) `cm/cm_conn.c`: `cm_ConnByHost()` — This function locates/creates a connection (binding) to a server and, for file-server connections, may also perform the RPC `AFS_SetContext()`. The function will be re-named `ConnByAddr()`, and will be modified to create bindings for a specified server address, and to reset `failoverCount` after a successful `AFS_SetContext()` call.

A new `cm_ConnByHost()` will be implemented which has the same signature and semantics as the current function with that name. This function will call `ConnByAddr()` until either a server binding is obtained, or all of the server's addresses have been tried.

(b) `cm/cm_conn.c`: `cm_ConnByMHosts()` — This function attempts to get a binding for any one of a set of servers, in accordance with the server-preference scheme. It will be modified to utilize `ConnByAddr()`, rather than (the new) `cm_ConnByHost()`, so that addresses can be tried for binding in rank order across the set of all servers.

The current implementation of this function requires that sets of server references in volume and cell objects be kept sorted by rank. This requirement is made obsolete in moving to multi-homed server support with an address-oriented preference scheme.

(c) `cm/cm_rrequest.c`: `cm_Analyze()` — As previously illustrated, this function determines if an RPC should be attempted again. It will be modified to reset `failoverCount` after a successful RPC to a server.

(d) `cm/cm_cell.c`: `cm_NewCell()` — This function creates/updates a cell object, which contains, among other cell-related information, an array of pointers to server objects representing the flservers in the cell. In the current implementation, this function calls `cm_GetServer()` once for each flserver in the cell in order to (re-)establish this array. It will be modified so that for each flserver it will also call `cm_SiteAddrUpdate()` to update/create the list of server addresses.

> *NOTE: Currently, only one address per flserver is passed to* `cm_NewCell()`. *However, the machinery is in place and needs only to be enabled so that the dfsbind process can pass to the CM all flserver addresses obtained from the CDS. Enabling this code will be part of the modifications required for this project.*

(e) `cm/cm_volume.c`: `cm_InstallVolumeEntry()` — This function updates a volume object from information obtained from the FLDB (`struct vldbentry`). A volume object contains, among other information, two arrays of pointers to server objects, one for file servers and one for replication servers, with an entry in each for each machine that houses the fileset (volume). As is the case with cell objects, these array entries are defined via calls to `cm_GetServer()`. Thus this function will be modified in a similar fashion to `cm_NewCell()`. However, in addition to calling `cm_SiteAddrUpdate()`, this function may also call `cm_SiteAddrUpdateAllFLDB()` as discussed below in the section on address acquisition.

(f) `cm/cm_daemons.c`: `cm_Daemon()` — This function arranges for a thread pool to perform various background processing tasks. It will be modified to schedule a thread to periodically iterate through the list of servers, calling `cm_SiteAddrPingBad()` for each active server. This is discussed below in the section on address revival.

3.3. Operational Overview

Given the site object definition and integration discussed above, a complete operational overview of multi-homed server support can now be presented.

3.3.1. Address fail-over

Address fail-over is performed transparently within the canonical CM RPC framework:

```
do {
    if (connp = cm_ConnByHost(serverp, ...)) {
        /* got server binding */
        result = server_RPC(connp->connp, ...);
    }
} while (cm_Analyze(connp, result, ...));
```

A communication failure detected by either `cm_ConnByHost()` (in `ConnByAddr()`) or `cm_Analyze()` results in a call to `cm_ServerDown()`, which will perform address fail-over by calling `cm_SiteAddrDown()` to report the failed communication. The result of address fail-over can then be taken into account in determining if the server is to be marked down.

3.3.2. Address revival

Just as performing address fail-over is important to over-all system operation, so is determining when previously failed addresses are again available. Making a previously failed network connection available for use provides the opportunity for increased fault-tolerance, in the form of an additional connection for future fail-overs, and increased performance, in the form of an additional connection that may be ‘‘better’’ than one currently being used. The following is a description of the two methods by which address revival is performed: the first for servers marked as being down, and the second for servers marked as being up.

As part of actively maintaining server state information, the CM schedules a background thread to periodically call `cm_CheckDownServers()` which pings all servers marked down to determine if they have come back up. `cm_CheckDownServers()` iterates through the list of server objects, calling `CheckDownServer()` for each server marked down. `CheckDownServer()` will call `cm_SiteAddrUp()` to mark all the server's addresses as up, and then will ping the server at each address in rank order until either successful or all addresses have been tried.

Similarly, a background thread will be scheduled to periodically iterate through the list of server objects, calling `cm_SiteAddrPingBad()` for each active (up) server. `cm_SiteAddrPingBad()` attempts to ping a server on all addresses marked down to determine if the connection has been restored.

Together, these background threads will actively maintain each server's address-list state.

3.3.3. Address acquisition

In addition to maintaining each server's address-list state for currently known addresses, the CM must actively maintain the set of known addresses for each server. Recall that a server's address list is defined after the server object and associated site object are created. In `cm_NewCell()`, the function `cm_SiteAddrUpdate()` will be called with the list of all flserver addresses obtained from the CDS. In `cm_InstallVolumeEntry()`, the function `cm_SiteAddrUpdate()` will be called with the list of all file-server/repserver addresses obtained from the FLDB via a `vldbentry` structure.

Though the flserver address list provided in `cm_NewCell()` will represent all of the addresses specified for use by the server, this is not necessarily the case for the file-server/repserver address list provided in `cm_InstallVolumeEntry()`. The reason is that a `vldbentry` structure can contain a maximum of 16 addresses, which must be divided among all fileset instances, of which there can be as many as 16. Thus in the worst case (a fileset with 15 replicas), the `vldbentry` structure will contain one address per host. Note that this can occur in spite of the fact that the FLDB can always contain four (4) addresses per file-server machine (file-server/repserver).

To address this problem, the `cm_site` module exports the function `cm_SiteAddrUpdateAllFLDB()`, which attempts to update the address lists of the file-server/repserver on a given host from the FLDB via the `VL_GetSiteInfo()` function.[3] `VL_GetSiteInfo()` returns all addresses (up to four) specified for use by a file-server/repserver on a host. But because `cm_SiteAddrUpdateAllFLDB()` is a potentially high-latency operation, it will not be called directly by the thread executing `cm_InstallVolumeEntry()` (which could need to call it up to 15 times), but instead will be called by a worker-thread scheduled by that thread.

The CM also schedules a background thread that periodically (every hour) calls `cm_CheckVolumeNames()` to iterate through all volume objects and mark them as requiring checking. The next time the volume object is employed, it's state will be updated from the FLDB via `cm_InstallVolumeEntry()`. As a result, the CM will automatically (on-demand) track file-server/repserver address-list changes in the FLDB.

Similarly, the CM times-out cell information (after 24 hours) and refreshes it via `cm_NewCell()`. As a result, the CM will automatically (on-demand) track flserver address-list changes in the CDS.

> *NOTE: A problem with using* `VL_GetSiteInfo()` *to get file-server/repserver address information is that it requires knowing one address in the desired list. An administrator could potentially change addresses for a given file-server/repserver in the FLDB before address list completion, in which case a complete list might not be obtained until after the next time* `cm_CheckVolumeNames()` *is executed. One possible solution (among many) is to implement an flserver operation that would retrieve site information based on host principal name. In practice, this should not really be a concern.*
>
> *Future updates to the flserver protocol may obviate the need for the* `cm_SiteAddrUpdateAllFLDB()` *function, except when dealing with older versions of the flserver.*

3.3.4. Server-preference scheme

The cache manager implements a preference scheme whereby server addresses are assigned a specified or computed rank. Address rank is used to determine fail-over ordering when obtaining information from one or more servers.

Because the CM currently supports only a single address per server, assigning address ranks is equivalent to assigning server ranks. However, with multi-homed server support, the CM will exhibit the intended semantics: when attempting to contact one of a set of servers (where set size can be one), the CM will employ *addresses* in rank order.

Supporting address-preference semantics is the reason that `cm_ConnByHost()` and `cm_ConnByMHosts()` will be implemented via `ConnByAddr()`. In doing so, `cm_ConnByHost()` can try for binding all (up) addresses in rank order for a particular server, while `cm_ConnByMHosts()` can try for binding all (up) addresses in rank order across a set of servers.

3.4. Performance Issues

This section identifies several performance issues alluded to throughout this document. These are really tuning issues concerned more with ''how much'' rather than with ''how''. No specific tuning values are given; this will require some analysis and experimentation.

---

3. `cm_SiteAddrUpdateAllFLDB()` will update more than one file-server/repserver on a host if extant; e.g., to support multiple DFS servers behind a firewall.

(a) Fail-over throttling — In performing address fail-over, the server structure field `failoverCount` is used to count the number of address fail-overs since the last successful RPC to that server. This value is used to limit the number of fail-overs that can occur before a server is deemed to be dead.[4] Restricting fail-overs in this manner bounds the amount of time that is spent attempting to contact a server that is likely to be down. The fail-over count limit must be chosen to balance operation latency with fault-tolerance effort.

Note that throttling is not applied in the server module function `CheckDownServer()`, which pings a server marked down to determine if it has come back up, so that all addresses can be tried.

(b) Daemon period — To revive failed addresses for functioning servers, the CM arranges for `cm_SiteAddrPingBad()` to be called periodically for each active server. The period for this function must be chosen to balance the potential benefit with the overhead. However, since (observed) address failure should not occur often, execution overhead should normally be low. Thus a period on the order of 15 minutes is probably not unreasonable.

4. CONCLUSIONS

This document proposes a method for supporting multi-homed servers in the DFS cache manager which meets the stated objectives, namely:

(a) transparent handling of network connection failures,

(b) active management of each server's network connection state, and

(c) compliance with the server-preference scheme.

These objectives are met by introducing a *site object* into the existing CM communication architecture to represent the state of a server's network connections. It is shown that site objects can be integrated into the code base in a straightforward fashion, with most significant modifications concentrated in the server module.

5. ACKNOWLEDGMENTS

The author wishes to acknowledge the valuable comments of many of the folks at Transarc, in particular Ted Anderson, Steve Berman, Craig Everhart, Bruce Leverett, Dan Nydick, Lyle Seaman, M. C. Srivas and Bill Zumach.

AUTHOR'S ADDRESS

Steven Moyer
Transarc Corp.
707 Grant Street
Pittsburgh, PA 15219
USA

Internet email: moyer@transarc.com
Telephone: +1-412-338-2047

4. As an exception, a file-server is never deemed to be dead until the host lifetime expires.