Open Software Foundation
Request For Comments: 74.0
May 1995

S. Berman (Transarc)

SERVER PREFERENCES IN DFS

1. INTRODUCTION

The goal of this work is to modify the DFS Cache Manager to adhere to a set of user-specified preferences determining which hosts to contact for various services. This work proposes to give the DFS Cache Manager the ability to allow administrative users to direct preference for one file server over another. It also chooses which server to contact first intelligently in the absence of specified preferences. Server preferences in DFS will utilize a ''rank'' value issued to each file server with which the Cache Manager has communicated. This rank value is examined whenever a new fileset is accessed to determine the order in which servers containing read-only replicas and read-write filesets will be sought out.

2. BACKGROUND INFORMATION

In a DFS cell where some file servers contain read-write filesets and others contain read-only replicas, we wish to select the file server that is best able to provide the data to us. ''Best able'' is a qualitative term that may include such criteria as, file server cpu model and current load, network ''distance'' from the client, or even network link throughput on the route between the client and servers. Since many factors will be important in deciding server preferences, I do not address these issues; instead I propose a method for setting and abiding by server preferences, and leave the rest to the system administrator. In the case of no specified server preference, a reasonable default value can be computed based only on what the network connection is between the client and server. These values will establish an intelligent set of default preferences.

The common file network topology is one in which the read-write version of a particular fileset is located on a server host attached to the backbone network, with read-only replicas on servers on each subnet. Given a *random* server selection, the load will be distributed evenly across the fileservers, but (n–1)/n requests will probably be traversing the network backbone. We wish to dictate a default preference that would force the Cache Manager to prefer ''closer'' servers over more ''distant'' servers. In this analysis, ''distance'' is considered to be a measure of how many network routing devices a message must pass through before reaching its destination.

The primary concern is network utilization, with load balancing a secondary concern. Note that the random technique did not achieve true load balancing (''from each according to his ability, to each according to his need''), either. It merely tended to provide an equal work distribution without regard to servers' capabilities.

If other mitigating factors affect the performance of certain file servers with respect to certain clients, then these factors will have to be considered by the local system administrator, and addressed by setting server preferences on individual clients using the provided command. This is the only sensible alternative since any attempt to dynamically compute server preferences would be certain to overlook site-specific configuration details.

3. ARCHITECTURE OF PREFERENCES

The architecture of server preferences is based on replacing the random selection from the available list of servers for a particular fileset with a selection criteria based on a per-server **rank** value. Rank is an `unsigned short` value. Lower rank values indicate higher preference.

In the existing DFS Cache Manager, a host that provides file data, file location or replication services for any particular fileset is represented internally in a `struct cm_server`. The rank value is stored here. Each fileset is represented in a `struct cm_volume` which contains two arrays of pointers to `cm_server` structures. One list points to fileset data servers, and the other list points to servers to be contacted for read-only replica related services (keep-alive, new-file versions for replica, etc.).

These two lists and two associated per-server status lists get sorted in order of increasing server rank, and hence, determine where to send RPC service requests. The preferred (lower rank value) servers are tried first since they migrate to the heads of the lists.

4. NEW IMPLEMENTATION

The proposal contains specifications for the general algorithms and data structures needed for server preferences and a method for computing default ranks. There are several implementation problems to be solved here.

(a) The server rank value must be stored somewhere.

(b) Need to be able to set rank for a server without a corresponding `struct cm_server`.

(c) Need to implement a network-topology-based default ranking algorithm.

(d) Need to use rank to influence choice of server when making RPC.

(e) Need to be able to get and set rank for a server through an API.

(f) Must provide a user interface via the `cm` control program.

4.1. Storing Server Rank

The Cache Manager creates a `cm_server` entry with information about each known server when it first references that server. This entry contains a rank field which is set to an appropriate default value. The rank field will replace the random field in this structure. A user interface will allow the administratively empowered user to adjust this value. However if the user specifies a rank for a server which has no associated `cm_server` structure then we must queue the rank request for use when that server is eventually contacted for the first time and the `cm_server` structure set up for it.

4.2. Ranks for Servers Not Yet Contacted

The system must be able to handle a request to set a server's rank even in the absence of a `cm_server` structure for that server. Such an operation will require the recording of the desired rank for that server in some holding area. The server rank, type and IP address will be recorded in a structure attached to a linked list of other such requests that are considered outstanding. These structures will be allocated at time of request and freed when a newly contacted server has an IP address and type which matches one of them. The list is scanned when new entries are made to prevent duplicate requests for the same IP address and type being added.

Whenever a new server structure for a file exporter or replication server is added to the system, a function will try to establish its rank by looking for another server with the same IP address and getting the rank from there. This keeps file and replication servers at the same rank, and ensures they get sorted correctly. Failing finding another server, the system will search the list of outstanding requests for an entry with a matching IP address and service type. If no request is found it will compute a default rank for the server based on the network adjacency policy. In this way we can ensure server ranks will be set according to administrative preference, but without violating any of the locking hierarchies involved in creating new `cm_server` structures.

4.3. Computing Default Ranks

Sensible default server ranks must be determined without very much administrative information available. Therefore the choice is made to compute default ranks solely on the basis of network adjacency and connectivity parameters that can be determined by examining IP addresses and network interfaces.

Server ranks are in the range from 0 to 65534. In the absence of user-provided ranks, the Cache Manager will assign a default rank value as follows:

(a) If the server machine has the same address as (i.e., ''is'') the local host, it will be assigned a rank of 5000 plus a small random number.

(b) If subnetting is in use on an interface, and the server's address is in the range of that subnet, the server will have an associated rank of 20000, plus a small random number.

(c) Otherwise, if the server's address is in the range of the network, the server will have a rank of 30000 plus a small random number.

(d) A server located on a network which is not directly connected to this host will be assigned a default rank of 40000 plus a small random number.

(e) If no locality information can be determined in a particular kernel, the default rank will always be 40000.

(f) Ties (equal ranking) among servers which export the same fileset will be resolved randomly.

Note that any machine may potentially have more than one interface and more than one IP address. Therefore we search all interfaces and all addresses configured to find the closest network connection to any server when computing its default rank.

For these computations, the definition of ''same network'' and ''same subnetwork'' are taken to be in the context of the Internet Address protocol, where specific classes (A,B,C,D) of networks have been defined and their significant addressing digits for host, subnetwork, and network are well understood.

4.4. Using Server Rank to Implement Preferences

Implementing server preferences requires a method for using rank to influence which server gets the RPC request for a particular service. I propose to do this by manipulating a part of the `cm_volume` structure.

Currently, when a new `cm_volume` structure (for a fileset) is allocated, or an old one is reread from its place in the `FilsetItems` disk file, it is installed in the system by the function `cm_InstallVolumeEntry()` in `cm_volume.c`. This function finds all the file servers and replication servers that service this fileset, and places pointers to their `cm_server` structures into two arrays. One array is for file servers, the other for replication servers. After filling them, it currently calls a function to sort these arrays based on a random number.

In a similar fashion, each time a new cell is accessed, a `cm_cell` structure is allocated, and the FLDB servers for that cell are identified. Pointers to `cm_server` structures are then filled into the `cellHosts` array for that cell. These pointers can be sorted just as the the ones above are.

In `cm_server.c`, I propose to change the sorting function `cm_SortServers()` to use the rank field instead of random. This function already sorts the entries in a host list by numerical value, so no other changes will be needed to enforce host list sorting by rank. Note that this function will sort these arrays in ascending rank order.

The order that servers appear in these two arrays governs directly the order in which they are polled to satisfy a service request. The function `cm_ConnByMHosts()` searches the array of server pointers passed in for a connection to a server host. The search proceeds in the order of the array. Only if the server is marked ''down'' or a timeout occurs does the Cache Manager attempt to use the next server in the array. This applies for both file exporter servers and replication servers. By extension, FLDB servers are also sorted in the `struct cm_cell` for choosing which of several file location database servers to contact.

Several new functions will need to be written to adjust the rank value for both existing and new server structures. Additionally, when an administrator adjusts a server's rank, there will be a function to scan the active set of filesets to determine which server lists will need to be re-sorted based on the changed ranking. The ranks for replication servers and file servers on the same host will always be the same since these services are always used together by the Cache Manager for read-only fileset access.

4.5. Programming API

Server preferences should be like other Cache Manager configuration parameters. They will be accessed and changed via the `pioctl` interface. All `cm pioctls` use a `struct afs_ioctl` to carry parameters to and from the Cache Manager. I propose to use this structure in the standard way to add a `cm pioctl` interface.

4.5.1. Pioctl VIOCSETSRVPREFS

The Set Server Preferences `pioctl` uses an array of the following structures, passed in the `afs_ioctl.in` data area. The user must allocate this array and specify the size (in bytes) in the `afs_ioctl.in_size` field of the pioctl request.

```
struct spref {
    struct sockaddr_in  server;
                        /* socket address in network order */
    unsigned short      rank;
                        /* Server rank */
    unsigned short      flags;
                        /* Flags for cm to set on return */
};
```

This `pioctl` can only be used by the local root user, otherwise it will fail with the error code `EPERM`. The specified preferences will apply equally to replication and file servers. If neither a replication server nor a file server can be found for a specified address, the request will be queued into an outstanding request queue. Flags may be set to `CM_PREF_FLDB` to cause the preferences adjustment to apply to only File Location Database servers. Normaly this `pioctl` will return zero.

If `afs_ioctl.in_size` is larger than the supported maximum, the `pioctl` will return a non-zero return code, and the error code will be `E2BIG` and no data will be returned. All of the preferences will remain unchanged.

4.5.2. Pioctl VIOCGETSRVPREFS

This `pioctl` can be run by any user to extract the current set of preferences. `VIOCGETSRVPREFS` will accept an integer offset into the Cache Manager's list of servers, and will return addresses and ranks of servers from that point on until the data area specified by the user is filled. The `afs_ioctl.in` area should be a structure of the following type:

```
struct sprefrequest {
    unsigned short  offset;
    unsigned short  num_servers;
};
```

The offset field should contain an integer offset into the Cache Manager's list of servers. A value of zero will indicate that the returned list will begin at the beginning (i.e., offset zero). The `num_servers` field should be set to the size of (number of elements in) the `servers[]` array of the `sprefinfo` structure described below. If the `afs_ioctl.in_size` field is smaller than `sizeof(struct sprefrequest)`, `ENOENT` will be returned.

Several requests may be required in order to obtain the entire list of servers. The appropriate offset for each subsequent request will be the value returned in the `next_offset` field by the previous request, unless there was no previous request, in which case, the appropriate offset is zero. The output from this `pioctl` is a structure of the following type:

```
struct sprefinfo {
    unsigned short  next_offset;
    unsigned short  num_servers;
    struct   spref  servers[];
};
```

This structure is passed in the `afs_ioctl.out` data area of the `ioctl` structure, and must be allocated by the user. The maximum size of the structure (in bytes) will also be specified in the `afs_ioctl.out_size` field of the `pioctl` request. Note that `servers[]` is a variable length array whose size is constrained by the smaller of the `afs_ioctl.out_size` field and the `num_servers` field of the `sprefrequest` structure.

On return from this `pioctl`, `num_servers` will contain the number of elements actually returned in the array, while `next_offset` will contain the value which should be passed in the `offset` field on the next invocation of `VIOCGETSRVPREFS` in order to continue iteration through the list. Under no circumstances will the returned value of `num_servers` ever be larger than that specified in the `sprefrequest` structure, though it will often be smaller. If there are no more elements remaining in the list, `next_offset` will be zero. This suggests that the next invocation of `VIOCGETSRVPREFS` should start from the beginning of the list. The output list is comprised of servers found in the server hash table first, and next servers listed in the outstanding requests list.

If the value of the afs_ioctl.out_size field of the pioctl request is smaller than

```
sizeof(struct spref) * sprefrequest.num_servers + 2*sizeof(int)
```

server preferences information will be omitted by subsequent calls to this iterator. It will in effect, ‘‘fall through the cracks’’ between calls. This will not cause any harm to the Cache Manager, only the quality of the data returned by this pioctl will be adversely affected. Also note that additions to the list of servers maintained by the Cache Manager between subsequent calls to this iterator may cause information for some servers to be omitted (effectively, the offsets have changed). The output servers[] array has each entry’s flag value set to indicate whether the entry found was for a file exporter, replication, or FLDB server.

4.6. User Interface

The default user interface to the server preference pioctls is via the CM command suite. CM is modified so that the local root user can specify server ranks directly. The local root user will be able to set or override the current rank of a particular server by using the setpreferences subcommand of CM. The server ranks may be examined via the getpreferences subcommand. Any user may execute the getpreferences subcommand. The -fldb modifier flag is to indicate the user is interested in querying or modifying preferences for file location database (FLDB) servers instead of file data and replication servers.

```
cm setpreferences  [-server <machine rank>...]
                   [-path <filename>] [-stdin] [-fldb] [-help]

cm getpreferences  [-path <filename>] [-numeric] [-fldb]
                   [-help]
```

Only the local root user may set server preferences. The CM setpreferences subcommand will take data from any combination of specified sources, and will issue the necessary pioctls to effect the requested changes. Host names will be resolved to IP addresses using the local resolver method. The getpreferences subcommand will resolve address to names (when possible) unless the -numeric switch is given. Unresolved addresses will be in dotted decimal form. Use of the -numeric switch is highly recommended if the output is not intended for human consumption.

5. Cache Manager Code Changes

Changes to implement server preferences include new functions and data structures to manipulate server ranks and server lists, and some changes to strengthen the locking hierarchy protecting these lists.

5.1. Changes to Existing CM Functions

The cm_server structure is changed so that the random field is now called rank. When creating a server structure in cm_GetServer() the rank value is set by a new function called cm_DefaultRank(). The function for sorting server lists, cm_SortServers(), is changed to observe the rank value when sorting, and is always called with a lock held on the parent cm_volume.

A change is needed in cm_ConnByMHosts() to more aggressively require the volp->lock be held whenever looking into the volume structure’s server lists. This is necessary to use the volp->lock as the semaphore for changing these server lists. When resorting server lists we take this lock to prevent concurrent access to arrays that are changing.

Since cm_ConnByMhosts() also drops its locks while attempting to use a server structure to obtain an RPC connection, we have to go a step further to prevent the possibility of missing a server in a list that gets resorted between RPC connection attempts. This is done by use of the generation count added to the cm_volume structure. This count is incremented after each resorting. The count is read when we enter the loop in cm_ConnByMHosts() and checked prior to each iteration. If it changes, we start over knowing that a list in this cm_volume structure has been resorted.

5.2. New CM Functions and Data Structures

Two new files, cm_serverpref.c and cm_serverpref.h, will be added to the source tree. The first file will contain most of the new server preference manipulation code. However the new functions cm_FindNextServer(), cm_FindServerIP() and cm_SetServerRank() will be added to cm_server.c to provide functionality that is needed for the implementation. The reason for putting these functions in cm_server.c is that they access the server hash table directly and modularity requires that they be located in the file which contains the objects and locks that they manipulate. Similarly, the new function cm_ReSortServers() will be placed in cm_volume.c since it accesses the cm_volume structures and locks directly. Also, cm_ReSortCellSrvs() will be placed in cm_cell.c for the same reasons.

5.2.1. cm_FindNextServer()

```
struct cm_server *
cm_FindNextServer(struct cm_server  *lastHad,
                  u_short            type);
```

This function finds us the next server in the cm_servers[] (IP address) hash table after the one passed in. If NULL is passed in, then it returns the first server in the cm_servers[] hash table. This function acquires the cm_serverlock during the hash table search. The structure pointer it returns does not have a lock held or reference count incremented. It skips over cm_server structures which have their sType field set to anything other than the provided argument.

5.2.2. cm_FindServerIP()

```
struct cm_server *
cm_FindServerIP(struct in_addr  *addrp,
                u_short          type);
```

This function is similar to cm_FindServer() in that it returns a server structure pointer, but this one will find the structure given the type and the IP address. This function must be called holding the cm_serverlock for the hash table search. It does not lock the server structure, but its caller may do so providing it releases the cm_serverlock. Server structures are not currently ever deleted from the cm_servers[] hash table so this locking strategy works. The reason this function does not lock the hash table like cm_FindServer() does is that his caller may wish to hold the lock after an unsuccessful search to guarantee no race condition exists between failing to find a server and adding a request queue element.

5.2.3. cm_SetServerRank()

```
int
cm_SetServerRank(struct in_addr  *addrp,
                 u_short          rank,
                 u_short          svc);
```

This function will set the rank value for the named server and type if it is currently in the system. If not, a rank request must be added to request list. Note that any *existing* request for the same server and service type must be deleted. This function uses cm_FindServerIP() and holds the cm_serverlock while searching the hash table and adding a request. This locking prevents the possible race between adding a request and contacting the same server at the same time. This function returns zero after normal completion.

5.2.4. cm_ReSortServers()

```
void
cm_ReSortServers(struct cm_server  *serverp);
```

Whenever an existing server preference has been reset we must carefully sort all the server arrays in all volumes that contain a reference to that server. Fortunately, the cm_fcache stored in the FilesetItems file on disk do not contain Host arrays, so we can ignore them and only look at the cm_volumes[]. The function cm_ReSortServers() is a straightforward find and sort routine. It has to hold a read lock on cm_volumelock and a write lock on each specific volume structure while it is being examined/sorted. It is unfortunate that we have to hold a global lock and a local volume lock for this long, but the operation should not have to occur very often.

There are several functions which read the repHosts[] and serverHost[] arrays. These functions will have to be changed to detect that the array has been resorted by examining the new per-array generation counter.

This counter should be consulted whenever the arrays are accessed to determine if the array was resorted since last examined.

5.2.5. cm_ReSortCellSrvs()

```
void
cm_ReSortCellSrvs(struct cm_server  *serverp);
```

This function performs the same task as `cm_ReSortServers()`, but for cell FLDB servers in the `cm_cell` structure's `cellHosts[]` array. It is implemented in a similar fashion as `cm_ReSortServers()` described above.

5.2.6. cm_SortServArrays()

```
static void
cm_SortServArrays(struct cm_volume  *volp);
```

This function works like `cm_SortServers()`, but recognizes the association between the arrays `serverHost[]`, `timeBad[]` and `perSrvReason[]`. These three arrays all record data about the same collection of servers. They by necessity must be sorted together, so this function is provided to handle this special case.

5.2.7. Queued Rank Requests for Unknown Servers

The following structure definition and function prototypes describe the mechanism for storing and retrieving preference requests for servers not yet contacted by the Cache Manager. The functions listed add, remove and scan for a particular preference request.

```
typedef struct cm_ServerRank {
    struct in_addr         server; /* IP address in
                                      network order */
    unsigned short         rank;   /* Server rank */
    struct cm_ServerRank  *next;  /* Next in list */
    struct cm_ServerRank  *prev;  /* Previous in list */
} cm_ServerRank_t;

static struct cm_ServerRank  *cm_ServerRankReqs = NULL;
static unsigned long          cm_RankReqsCnt = 0;
static osi_dlock_t            cm_RankReqsLock;

void
cm_AddRankRequest(struct in_addr  *addrp,
                  u_short          rank,
                  u_short          svc);

cm_ServerRank_t *
cm_ScanRankRequests(struct in_addr  *addrp,
                    u_short          svc);

static void
cm_RemoveRankRequest(cm_ServerRank_t  *srp);
```

The list pointed to by `cm_ServerRankReqs` maintains all the user specified ranks for servers not yet in the internal `cm_servers[]` hash table. This list is consulted whenever a new server structure is allocated and needs its default rank to be set by `cm_DefaultRank()`.

5.2.8. cm_RandomRankAdj()

```
int
cm_RandomRankAdj(void);
```

This function returns a random number in the range of 0 to 15, which is used as a small, random adjustment applied to server ranks to prevent ties.

5.2.9. cm_DefaultRank()

```
u_short
cm_DefaultRank(struct in_addr  *addrp,
                u_short          type);
```

This function returns a rank value for a new server structure. It replaces the call to the random value generator in `cm_server.c`. If a rank request for the same IP address and service type exists in the request queue, then that value will be used instead. Otherwise the algorithm described earlier is used to establish a default rank based on existence of another server instance on this IP address, or the network topology.

5.2.10. New Pioctl Interfaces

Two new `pioctl` definitions are added to `kutils/ioctl.h` for preferences. These are `VIOC_SETSPREFS`, and `VIOC_GETSPREFS`. These definitions correspond to new functions located in `cm_pioctl.c`, called `cm_PSetSrvPrefs()` and `cm_PGetSrvPrefs()` respectively.

```
static long
cm_PGetServPrefs(struct cm_scache    *scp,
                 long                 function,
                 struct cm_rrequest  *rreqp,
                 char                *inDatap,
                 char                *outDatap,
                 long                 inSize,
                 long                *outSizep);
```

This is the `pioctl` subcommand that implements the user interface for getting all the server ranks. It must iterate over the `cm_servers[]` hash table and the `cm_RankRequest` queue.

```
static long
cm_PSetServPrefs(struct cm_scache    *scp,
                 long                 function,
                 struct cm_rrequest  *rreqp,
                 char                *inDatap,
                 char                *outDatap,
                 long                 inSize,
                 long                *outSizep);
```

This is the `pioctl` subcommand that implements the user interface for setting the server rank. See the user interface section for more information about what the commands that invokes these functions can do.

5.3. New OSI Functions and Data Structures

The following functions are added to the OSI layer to provide a machine independent set of boolean interfaces to determine facts about the network topology of the cell. These functions are used only in `cm_DefaultRank()`.

```
int
osi_SameHost(struct in_addr  *addrp);
```

Recognize if the address passed in is the same as the local address for any interface attached.

```
int
osi_SameSubNet(struct in_addr  *addrp);
```

Recognize if the address passed in is on the same subnet as the local address for any interface attached.

```
int
osi_SameNet(struct in_addr  *addrp);
```

Recognize if the address passed in is on the same net as the local address for any interface attached.

5.4. User Interface Functions

The following functions are added to the CM program to provide the administrative commands for getting and setting server preferences:

```
int
DoSetPrefs(struct cmd_syndesc  *aSyntax,
           char                *aRock);

int
DoGetPrefs(struct cmd_syndesc  *aSyntax,
           char                *aRock);

void
SetUpSetPrefs(void);

void
SetUpGetPrefs(void);
```

These functions are written following the format of the other CM subcommands.

6. TESTING

Testing of the Server Preference package will proceed in two stages. First a unit-test stage, which will attempt to exercise each component of the server preference module individually in user space. Second, a system test stage where using a cell with some number of servers we exercise the system and verify that server preferences are being followed.

All of the new functions in `cm_serverpref.c` can be exercised independently by use of a test program that simulates what the rest of the Cache Manager might do. A user-space program that exports the global locks and data structures that server prefs depend on will be able to test each function for correctness. Locking can be similarly tested using parallel threads that create server setting requests and access servers.

Appropriate ICL traces will be added to the server preference code as it is developed. Since no new global data structures are being added and only one field of the `cm_server` structure is changing no additional debugging interfaces will be needed. Eventually it would be helpful to add a debugging interface that would return the name of the server that is being used for a particular fileset.

After passing the unit tests, a CM kernel module with preferences will be run in a test cell. Here I will exercise the `pioctl` code and the server preference setting and getting modules. Next I will conduct scenario tests that will exercise and verify the server choices.

By replicating some filesets across three servers located on different IP subnets, the default server preference value choice will be tested. Two subnets will be required to effect this test, since there are four cases of where the host can be located (self, same subnet, same net, other net).

First, using `cm getpreferences` I can visually confirm that generated server ranks conform to the described algorithm. Next, attempting to access the replicated filesets with ICL tracing enabled will generate a trace dump that names the server that shipped the data. Again visual inspection will confirm the correct behavior of the system.

The boundary conditions of the system can also be tested using some shell scripts. Adding too many preferences, parallel `cm setpreferences` requests, and setting a request for a server at the same time as first contacting it can all be accomplished using some shell scripts and ICL trace analysis to verify correct behavior.

Lastly, a `cm setpreferences` command to decrease the rank of a server not currently preferred should result in fetching data from that server instead for the next fetch. This can also be visually confirmed in the ICL trace dump.

AUTHOR'S ADDRESS

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

Internet email: berman@transarc.com
Telephone: +1-412-338-6993