The Coda File Server

Peter J. Braam, braam@cs.cmu.edu

v0.9, 25 Dec 1998

Contents

1 Callback management in the server

This code is in `vicecb.cc`.

The file server keeps a table of hosts that are up, see the section on connection handling. Each time a call back is granted to a client, a file entry is entered in the hash table. When the file is modified, all clients holding the callback are notified.

Callback management handles a hash table for FileEntry's and for CallBackEntry's. A CallBackEntry is attached/removed to each FileEntry as needed.

A CallBackEntry contains a pointer to the host HostTable (the host (vicedep/srv.h) so that the host to notify can be found back. All structures are managed in VM. For locality of referenence, CallBacks and FileEntry are allocated in blocks. Routines are present to delete the callbacks for a Fid, or to remove all backbacks for a Venus which has failed.

2 Volumes

2.1 Volume Data Structures

When a server is initialized during its first run, a structure at a fixed RVM address is initialized, from which all data access is initiated. The structure is of type *coda_recoverable_segment* defined in `coda_globals.h`. For volume handling it holds the following information:

1. an array *VolumeList* of size *MAXVOLS* of *VolHead* structures.
2. a constant *MaxVolId* holding the maximum allocated volume id. This constant is overloaded and holds the constant *ThisServerId* in its top 8 bits.

We will now explore how the *VolHead* structures lead to other volume structures. There are many data structures linking these things together:

VolHead:

defined in `camprivate.h`. An RVM structure containing a structure *VolumeHeader* and *VolumeData*.

VolumeData:

defined in `camprivate.h`. An RVM structure, points to *VolumeDiskData* and points to *SmallVnodeLists* (with some contstants related to these, and to *BigVnodeLists*.

VolumeHeader:

defined in `volume.h`. Contains the *volumeid, type* and *parentid.*

VolumeDiskData:

defined in *volume.h*. An RVM structure holding the persistent data associated with the volume.

These RVM structures are copied into VM when a volume is accessed - we will describe this in detail. In VM we have a hash table *VolumeHashTable* for *Volume* structures, with hash key the *VolumeId*. This is used in conjunction with a doubly linked list *volumeLRU* of *volHeader* structures, which was probably created to avoid keeping all *volHeader*'s in VM, since the latter are large.

Volume:

defined in `volume.h`. This structure is the principal access point to a volume. These VM structures are held in a hash table. It contains quite a lot of information, such as a pointer to a *volHeader* (which has the cached RVM data), Device, partition, vol_index, vnodeIndex, locks and other run time data.

volHeader:

defined in `volume.h`. A VM structure sitting on a dlist, with a backpointer to a *Volume*. Contains a *VolumeDiskData* structure. This is the VM cached copy of the RVM *VolumeDiskData* structure.

Notice that in RVM volume's are identified principally by their index, which is the index in the static *VolumeList* array of *volHead* structures. Otherwise volumes are mostly accessed by their volume id. To map an index to a volumeid proceeds through the *VolumeHeader* structures held in the *VolumeList* inside the *volHead* structures.

The reverse mapping, to get an index from a *VolumeId* is done through an auxiliary hashtable *VolTable* of type *vhashtab*, defined in `volhash.cc`.

It is informative to know the sizes of all these structures:

- VolumeDiskData: 636
- VolHead: 88
- volHeader: 648
- VolumeHeader: 20
- Volume: 96

2.2 Initializing the volume package

The VInitVolumePackage sets up a lot of other structures related to volumes and vnodes.

InitLRU:

calloc a sequence of (normally 50) volHeader's, then call ReleaseVolumeHeader to put it at the head of the volumeLRU.

InitVolTable:

sets up the *VolTable*, hashing volid's to get the index in the rvm *VolumeList*.

VolumeHashTable:

Hash table used to lookup volumes by id. It stores pointers to the Volume structure

VInitVnodes:

setup of the vnode VM lru caches for small and large vnodes. Store smmary information in the VnodeClassInfoArray for both small and large vnodes. The way to reach allocated vnode arrays is through the VnodeClassInfoArray.

InitLogStorage:

go through the VolumeList and assign VM resolution log memory for every volume; store the pointer in VolLog[i]. The number of rlentries assigned is stored in:

`VolumeList[i].data.volumeInfo->maxlogentries.`

It is not yet clear if RVM resolution is using this.

CheckVLDB:

See if there is a VLDB.

DP_Init

Find server partitions.

S_VolSalvage:

This analyzes all inodes on a Coda server partition and matches data against directory contents. After this has completed, the volume *InUse* bit and *NeedsSalvage* bit are cleared (stored in the disk data).

FSYNC_fsInit:

This interface is discussed below.

Attach volumes

Now iterate through all volumes and call *VAttachVolumeById*. The details a far from clear, but the intent is to add the information for each volume to the VM hash tables.

VListVolumes

write out the `/vice/vol/VolumeList` file (VListVolumes).

Vinit

This variable is now set to 1.

2.3 Attaching volumes

This code is sheer madness.

Attaching volumes is the following process. First a *VGetVolume* is done. If this return a volume, it has found it in the hash table. If that volume is in use, it is detached with *VDetachVolume* (error here is ignored). Attaching is not supposed to happen with anything in the hashtable already.

Next the *VolumeHeader* is extracted from RVM. If the program is running as a *volumeUtility* then *FSYNC_askfs* is used to request the volume. We continue to chech the partition and call *attach2*.

Attach2 allocated a new *Volume* structure, initializes the locks in this structure and fills in the partition information. A *VolHeader* is found from the LRU list and the routine *VolDiskInfoById* reads in the RVM data - this can only go wrong if the volume id cannot be matched to a valid index. If the needs salvage field is set, we return the NULL pointer (leaving all memory allocated). Attach2 continues by checking for a variety of bad conditions, such as the volume apparently having crashed. If it is *blessed*, *in service* and the *salvage* flag is not set, then we are ready to go and put the volume in the hash table. If the volume is writable the bitmaps for vnodes are filled in.

DetachVolume is also very complicated. It starts by taking the volume out of the hash table (forcefully). This means that *VGetVolume* will no longer find it.

The *shuttingDown* flag is set to one, and *VPutVolume* is called. This frees the volume and *LWP_signal*s people waiting on the *VPutVolume* condition. [It is not clear that any process is waiting, since they normally only appear to wait when the *goingOffline* flag is set, not when the *shuttingDown* flag is set.]

2.4 State of volumes

There are numerous flags indicating the state of volumes:

inUse

Cleared by *attach2*, but only if volume is *blessed*, not *needsSalvaged* and *inService*.

inService

The latter flag has some use in the volume utility package, not much else.

goingOffline

Taking a volume *offline* (as in *Voffline* means writing the volume header to disk, the *inUse* bit turned off. A copy is kept around in VM.

Set by *Voffline*

shuttingDown

used in *VDetachVolume*.

goingOffline

Set by *VOffline*. Clear by *VPutVolume*, *VForceOffline* and *attach2*. In this case *VPutVolume* sets *inUse* to 0, and *VForceOffline* does that and sets the *needsSalvaged* flag.

blessed

Heavily manipulated by volume utilities while creating volumes, backing them up etc. Probably indicates that the volume is not in an internally consistent state.

needsSalvage

This is an error condition. Cleared by *VolSalvage*, set by *VForceOffline*.

2.5 The FSYNC interface

Requests for volume operations, such as *VGetVolume* can come from:

- The fileserver
- A volume utility
- The salvager

The FSYNC package allows a volume utility to register itself. The call *VConnectFS* is made during *VInitVolutil* and makes available an array for the calling thread, named *OffLineVolumes*. This array is cleaned up during *VDisconnectFS* and contains a list of volumes that have been requested to be offline.

The requests that can be made are *FSYNC_ON* to get a volume back online. This clears the spot in the OffLineVolumes, calls *VAttachVolume* and finally writes out changes by calling *VPutVolume*.

The other request is *FSYNC_OFF* and *FSYNC_NEEDVOLUME*. If the volume is not yet in the threads *OffLineVolumes* then a spot is found for the volume. If this is not for cloning the volume is marked as *BUSY* in the *specialStatus* field of the *Volume* structure. In *VOffline* the volume header is written to disk, with the inUse bit turned off. A copy of the header is maintained in memory, however (which is why this is VOffline, not VDetach).

The FSYNC package can also watch over relocation sites. This is not functional anymore, and should probably be removed.

2.6 Interface to the *volumeLRU*

This doubly linked list of *volHeader*'s is one of the more confusing areas in the code. This is the interface between VM and RVM data, and the invariants are not very clear.

AvailVolumeHeader(struct Volume *)

See if there is a *volHeader* available for the volume passed.

GetVolumeHeader(struct Volume *)

Assign a *volHeader* to the *Volume* structure. This routine does **not** fill in the data.

2.7 Interface to the *VolumeHashTable*.

The when a volume is needed, a call is made to *VGetVolume*. This routine is given a *volumeid* to search for and finds the *Volume* structure in the *VolumeHashTable*. If this is not found it returns *VNOVOL*.

If there are users of this volume already, then we are guaranteed that the *VolumeHeader* structure for this volume is available too. Perhaps the *header* field in the volume structure still exists, otherwise a *volHeader* must be found in the LRU, which possibly involves writing an old one back to RVM.

If the *header* field in the volume structure is not null we are ok, if it is null we would like to use the first available header in the *volumeLRU* list. This is done by checking the *back* pointer in the *volHeader*.

2.8 Creating a volume

The start of this is found in vol-create.cc in the volutil directory.

1. We first initialize the volumeid with VAllocateVolumeId.
2. VCreateVolume (vol/vutil.cc) now build a VolumeDiskData structure *vol* in VM. The recoverable resolution log vol.log is generated with recov_vol_log which does a RVMLIB_REC_MALLOC to allocate a certain number of pointers to recle's.

3 Vnodes

3.1 Vnode data structures

1. VnodeDiskObject (cvnode.h): RVM structure holding the following information:
    - type, mode, owner, mtime

- inodeNumber: for a large vnode a pointer to a directory inode in RVM for a small vnode a real disk inode number of a file.
- version vector
- vol_index
- rec_smolink nextvn: next vnode with the same vnodeindex
- rec_dlist *log: pointer to resolution log

2. Vnode (cvnode.h): VM structure holding
   - hashnext, lruNext, lruPrev, hashIndex
   - vnodeNumber
   - VolumePtr
   - VnodeDiskObject *
3. VnodeClassInfoArray: global VM variable with summary information.

4 Server RVM layout

The initial RVM layout of a coda server is simple:

```
boolean_t       already_initialized;
struct VolHead VolumeList[MAXVOLS];
struct VnodeDiskObject    *SmallVnodeFreeList[SMALLFREESIZE];
struct VnodeDiskObject    *LargeVnodeFreeList[LARGEFREESIZE];
short    SmallVnodeIndex;
short    LargeVnodeIndex;
VolumeId    MaxVolId;
long    Reserved[MAXVOLS];
```

These fields are accessed through the CAMLIB_REC macro. The SmallVnodeIndex is the index of the highest pointer in the array, pointing to a free vnode.

The RVM layout is initialized in coda_init (recovc.cc) which allocates small and large vnodes in RVM for each of the entries in the free vnode list arrays. The vnodes are zeroed out.

4.1 Vnodes for volumes

The VolumeData structure has a pointer to the arrays: smallVnodeLists and largeVnodeLists.

4.2 Resolution Logs for vnode lists.

5 VolInfo interface

This should turn into the volume location server.

```
ViceGetVolumeInfo
        VRDB.find by name
        if not found try to find by id (VRDB.find nasty casts)
        if found: call
             vrent::GetVolumeInfo for the entry
                     fills in VSGADD stuff
                     special case for canonicalize
                     fills in the replica volid's in INFO
        else
             VGetVolumeInfo
                     gets it out of the VLDB_fd
             if successful and ROVOL: calculate GetVSGAddress
             elseif RWVOL lookup the volumeinfo for a replicated volume
        return information
```

6 Configuration databases

- Volume databases (text)
- `/vice/vol/VolumeList` Holds partition and volume information for volumes for which this server is custodian. This is file is generated by **codasrv**.
- `/vice/vol/groupid` is maximum replicated volume id allocated so far.
- HOSTS (text)
  - Entries describe volume storage groups: id hostname1 hostname2 etc. Stored in /vice/db/VSGDB
  - `/vice/db/hosts` servers should be in this file. Servers found in the VSGDB are checked to be in this file before proceeding to make a volume.
  - `/vice/db/servers` contains an id for each server.

Createvol_rep works as follows: Parameters are: volname vsgaddr partition-name [groupid]

- check we are the SCM
- Check that the volname doesn't appear in AllVolumes or VRList
- Get the list of servers from the vsgaddr using the VSGDB
- verify the servers against /vice/db/hosts
- get a new replicated id for this replicated volume using the `/vice/vol/groupid` file
- use volutil to create the volume on each host, using the partition, volumename and groupid acquired before. Note that each server now assigns a volume id for this newly created volume. It dumps

7 bldvldb.sh

- get `/vice/vol/VolumeList` from each server in `/vice/db/hosts`. Names is server.list.new. First try this using ftp, then rfs, then krcp.
- if the file was acquired move it to `/vice/vol/remote/server.list`
- dump them all in `/vice/vol/BigVolumeList`
- volutil rebuilds the VLDB with `volutil makevldb /vice/vol/BigVolumeList`

Note this rebuilds the VLDB on the machine where is script is run, i.e. the scm.