Design discussions for dm Server Group function

Steve Powell

February 13, 2009

Abstract

The group management function for collections of dmServer nodes is based around the concept that a group of nodes need to be managed as a synchronised unit. This synchrony is based on the assumption of a common workload, though with potentially variant configurations for different platforms or resources.

Current group monitoring and management concepts cannot properly support synchronisation due to the lack of a centrally managed configuration state that is aware of the run-time artefacts that form the core of the workload configuration.

This document records features of the design discussions that took place trying to understand how such a requirement might be met, and what components and responsibilities would be necessary.

This work was frozen on 12 February 2009, pending re-introduction of the dmServer group synchronisation requirement.

Contents

1 Intro

There was one team meeting and two design CRC meetings that gave rise to this document. They established some aims and started to break down the classes, subsystems and responsibilities for the parts of a design to support synchronisation and central coordination of a collection (group) of dmServer nodes.

2 Initial Design Guidance

The first meeting identified some basic guidance about what a group *is* and how it might behave.

2.1 Distributed coordination of a dm Server Group

A dm Server (server) can run a *coordinator* for a collection of servers called a *server group*. The servers in the group are called *group members*. All the members of a group share the tail of a repository chain.

No server can be a member of more than one group. A coordinator cannot be a member of a group it manages. A server may manage more than one group by 'running' several coordinators. It is the coordinator's job to keep track of the members and their (shared) state.

Figure 1 shows a schematic for this.

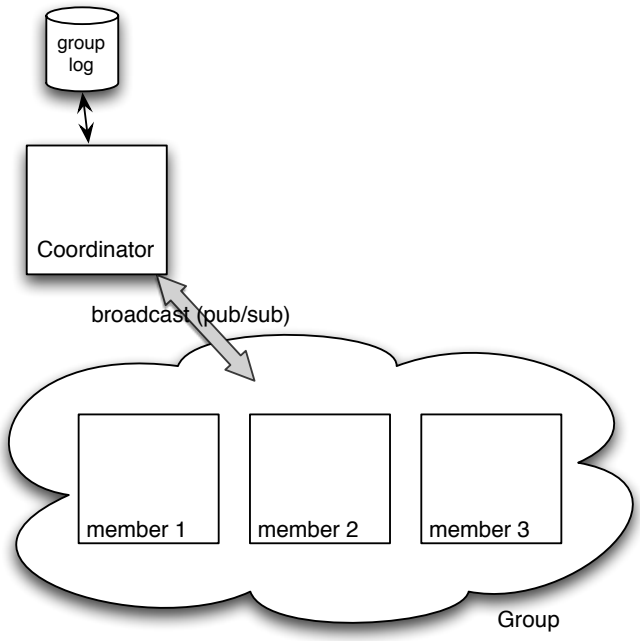

Figure 1: Relationship between coordinator and members.

2.2 Group identity

A group has a *name* by which it is identified, and group members know the name of the group to which they belong. (This is an initial configuration parameter; there are other pieces of state implied by this scheme.) The name is used to establish communication with the coordinator, and to identify logs.

The members of a group share the tail of their repository chain. That is, they may have (probably will have) one or more local repositories in their chain, but thereafter they share all subsequent repositories in their chains (and in the same order). The coordinator knows what the shared repository chain is, and uses this to maintain a common group 'state'.

2.3 Establishing a group

Before being managed by the coordinator a member must *join* the group. This involves prodding the coordinator and synchronising with the group. A current group state is passed from the coordinator and the member uses this to get in step. After that the member responds to the coordinator and may receive a series of commands to 'catch up' with the rest of the group.

2.4 Group control

The coordinator serially issues commands to the members of the group. The members are to obey these commands in the specified order and respond to the coordinator to acknowledge this. The coordinator keeps track of the desired state of the group as determined by the group commands and a presumed initial state. No attempt is made to monitor the state of a member: synchronisation is used to re-establish a member's state if it detects (or provokes) a discrepancy.

The commands and the state are encapsulated from the coordinator. These are passed in on the coordinator interface and used in logging/auditing, member synchronisation, and group state maintenance.

2.5 Leaving a group deliberately

Leaving a group deliberately is a request to the coordinator from the member. This allows the coordinator to remove information from the (active) logs for the group concerning the member. There is no other real consequence, except that if the member joins the group subsequently, the coordinator has

no knowledge of the member, of course, and resynchronisation may involve considerable change.

2.6 Leaving a group by accident

Leaving by accident (crash, comms failure, and so on) is only a(n admin) problem if the state is lost or (equivalently) some commands are missed. A gap in the series of commands is detected by the group member, or the member may decide to (re-)join in any case. The coordinator may detect (sluggish) response by the group member and provide re-sync protocol explicitly. Joining again after a period of absence is detected by the coordinator since a sequence number is passed to the coordinator when joining. This identifies where in the (log of) commands the new joiner is, so the coordinator can issue the appropriate synchronisation steps.

2.7 Discussion

When figuring this scheme out we realised that local deviations from the group state need to be allowed, and members of a group have to track the local state and the group state.

There are two ways in which a member dm Server can be configured differently from the other members of the group. One is through a local (repository) version of the configuration used in a group plan.

Members share the repository chain from some point on, that is: they have a private initial part of the repository chain (possibly just one local repository) and thereafter the chains are common (possibly just one shared 'group' repository).

When first joining the group the initial state transferred to the member might be deployed as is, but the local repository can override some of the references, for example there might be a local configuration file.

The second way is that local admin commands are issued for a single member, which are not reflected back to the coordinator.

In either case this can result in significant discrepancies from the group state. The coordinator cannot detect these, and will not attempt to correct them. (Note this obviates the need for the coordinator to track the individual states of each member.)

The group member needs to maintain two states, the (coordinated) group state and the local deviations state. We envisage that one is stored as a delta of the other. (Given that the *Runtime Artefact Model* will be needed anyway,

and that deltas are necessary parts of the single server implementation, this is not an unreasonable requirement.)

3 Use cases

To move design on, we held design meetings driven by scenarios (or use-cases) and building CRC (Class, Responsibility and Collaborator) cards.

The following scenarios were tabled to drive the CRC meetings:

1. issue a simple (non-state-change) operation to a group, for example, 'DUMP';
2. issue a state-change operations to a group, for example, 'DEPLOY artefact';
3. join a group;
4. start a group.

These are arranged in increasing order of difficulty (according to us on that day).

3.1 Simple operation

Figure 2 shows a diagram for the first scenario.

Sending a DUMP – the coordinator end

The DUMP command enters on a REST interface, which simultaneously identifies the group this command is for. It is a group command.

The REST interface first ascertains if there is a group coordinator for this group (by accessing the Group Coordinator Registry (or manager, or factory, or whatever)) which returns a coordinator instance if there is one for this group. [Is this right? Should the Rest Handler for DUMP actually do this – it knows this is a group command?]

The REST interface access the command subsystem to determine if a (group) DUMP command is supported.

If so it access and calls the *RestHandlerForDump* which is supplied by the command subsystem from a key which is generated from the REST URI interface.

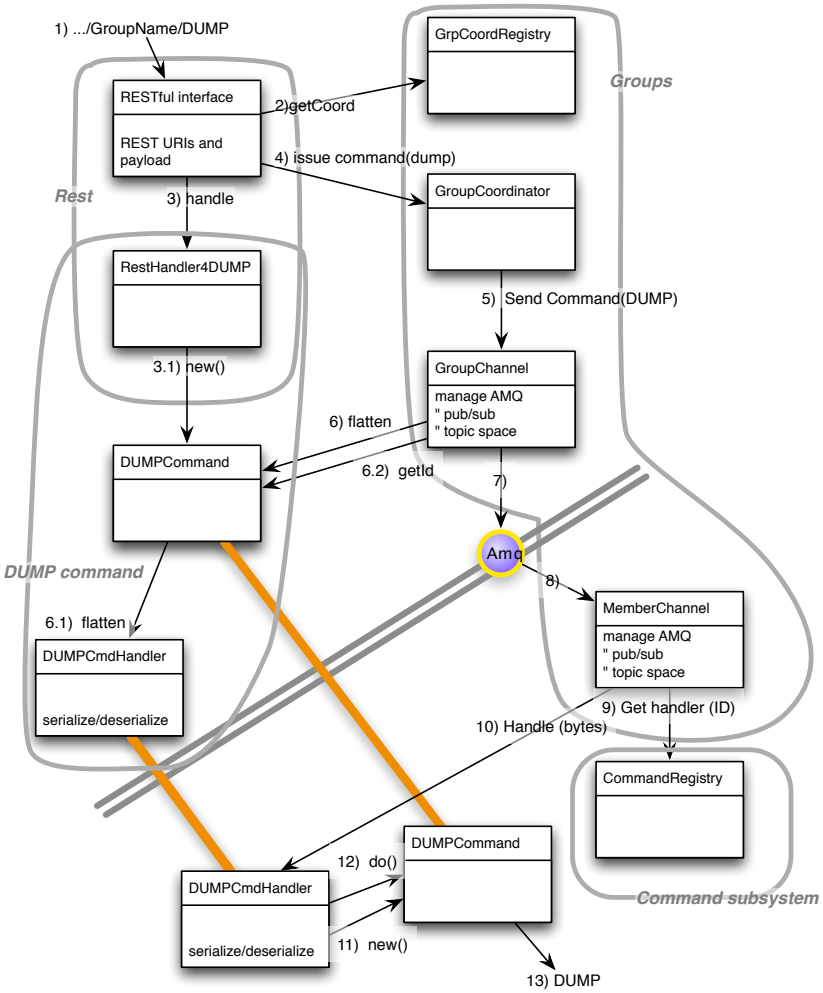

Figure 2: First CRC design for 'dump' use case.

A DUMP command instance is constructed and returned to the REST handler and thence to the REST interface which calls the (appropriate) group coordinator passing the command instance.

The coordinator sends the command to the members by calling the *Group Channel* (which is part of the coordinator state). The Channel arranges with the (group) command support for DUMP to ensure this command is transmissible over the channel (which the command does by deferring to a command handler for DUMP – it is likely that there is a generic serialisation/deserialisation handler for most commands) and the command is 'flattened'. The flattened command is then sent (in our case over pub/sub on AMQ). [Note here that the DUMP command identifier is attached to the channel

message to enable lookup in the member. This piece of design does not survive the next few days.]

Sending a DUMP – the group member end

The member channel instance receives the message (it subscribed to the topic for that group coordinator, which is specific to the group). It consults woith the command subsystem to discover the dump command from the id [which won't happen in the second incarnation].

The payload of the message (which are low-level bytes to the channel) is passed tot he DUMP command handler who de-serialises the bytes into a proper command (instruction) and passes this to the DUMP command proper. The DUMP occurs!

3.2 Revision of simple operation

After settling of minds the notion of an identifier on the command was denigrated, and the ownership of parts for subsystems was rethought.

Figure 3 shows the revised diagram.

DUMP revisions

Here we have removed the 'GetId' method call from the channel to the *DUMPCommand* and instead introduced a command handler in the member-side processing which the channel defers unpacking of the command to. The handler determines what command it is, gets the command handler from the command subsystem, and then calls the DUMP command handler with the rest of the payload bytes. The specific de-serialisation for the DUMP command then is completed.

The rest of the arrangement is more-or-less identical, except that the command subsystem now owns, and provides access to, the various parts of the support for the command, and the individual commands provide the REST interface support when they are 'enabled'.

We thus begin to develop a picture of the command subsystem and command support. The various commands (some of which are 'group' commands and some not) supply plug-able components to the command subsystem (and other subsystems possibly) which individually support group command support and local command support.

We will see in the next section that the group commands must be distinct from the local commands of the same nature (e.g. 'DEPLOY') though they

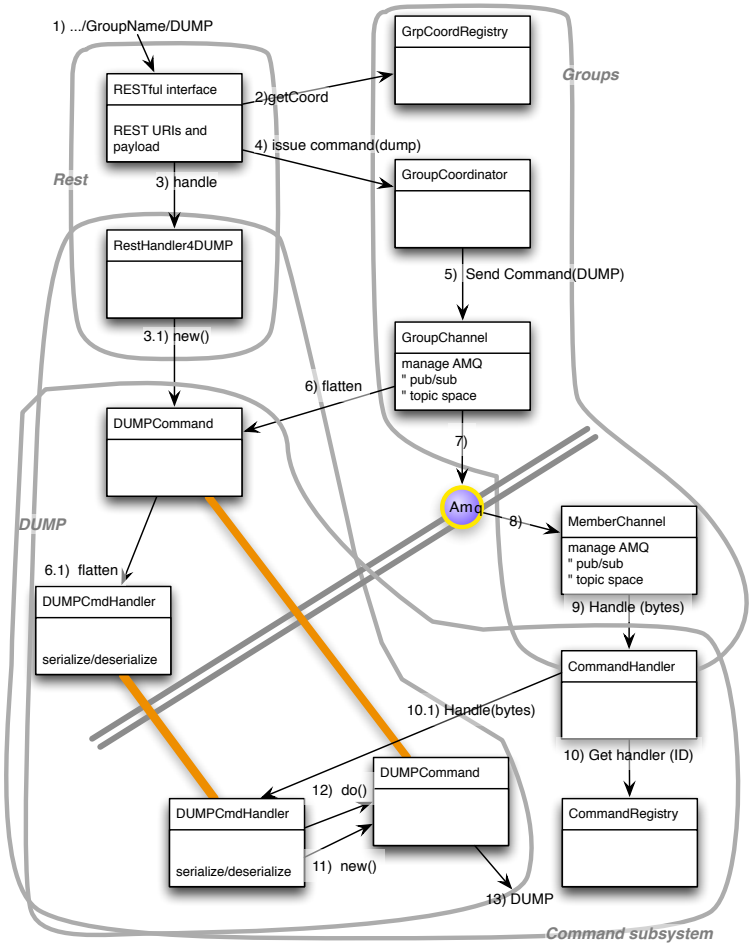

Figure 3: Revised CRC design for 'dump' use case.

are likely to share implementation.

3.3 State-change operation

By suppressing the detail of the first scenario we could explore the differences between a point-and-shoot command (like 'DUMP') which has no state-changes associated with it (?) and a state-changing operation (like 'DEPLOY') which does.

Figure 4 shows a diagram for the second scenario.

The subsystems are suppressed into grey fuzzy boxes instead of repeating the noise from the first use-case.

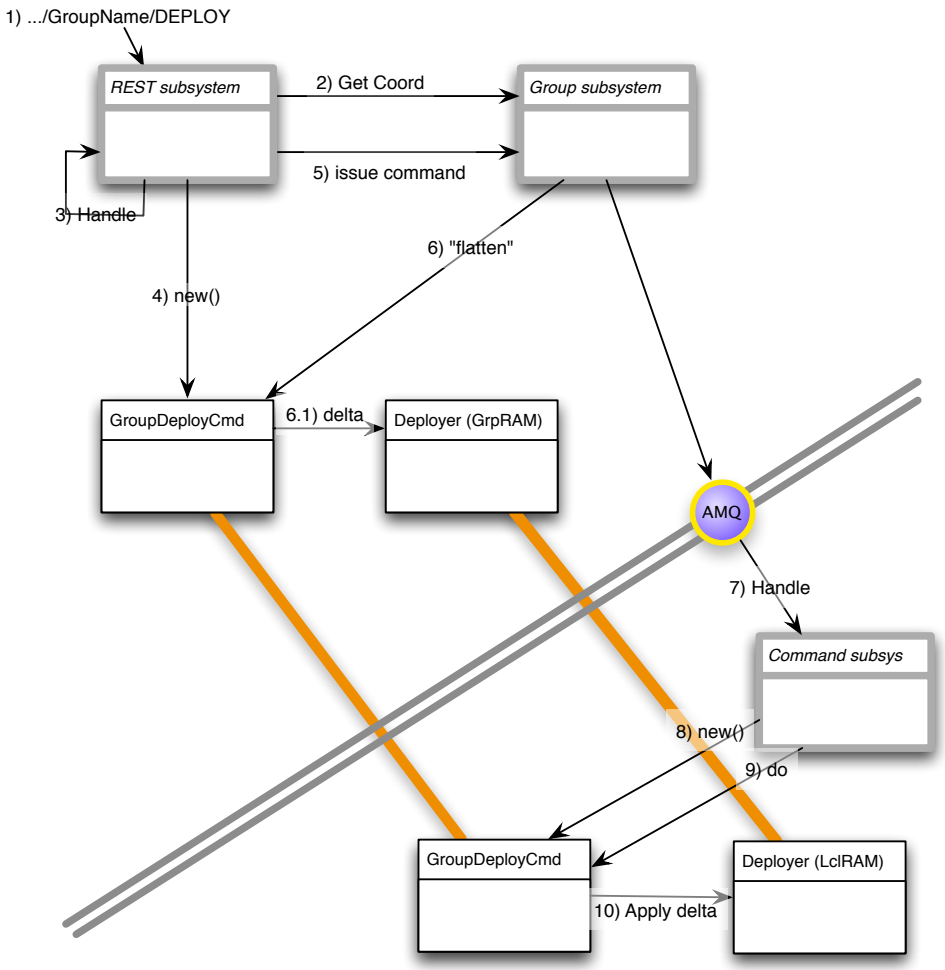

Figure 4: CRC design for 'deploy' use case.

Sending a DEPLOY – the coordinator end

The group command arrives in REST again and the coordinator located, as before. The GROUP DEPLOY command (and handler?) is then invoked by the group subsystem to "flatten" the command for sending.

here we encountered a problem – the action here is more (much more) than flatten. In fact it involves a deploy (simulation) on a copy of the Runtime-Artefact-Model we are maintaining for the group state. Various things might be logged and the group command may be tagged with a sequence identifier.

That's a lot of work for "flatten" to do, and we are not sure this is the right responsibility breakdown. However we progress.

The command is propagated (again) and the command subsystem gets to

handle it as before (we have omitted the channel stuff here, which is in the group subsystem fuzzy boxes on this diagram). The local Group Deploy command unpacks it, and when instructed to execute it does so.

A local 'group deploy' is cognisant of the membership of the dmServer, and which group it is in (though this is likely to be part of the command) and local state can be updated here. Local state may include a simulation of the group state and a delta (maintained locally) of the deviations for the local member from the group state.

In any case, after the deploy is completed (the local Deployer is involved in this) a response is sent back to the coordinator via the Command subsystem (which was handling the message). Notice that the response needs to identify the group member and the sequence number of the (DEPLOY) command that was issued.

The dealing with the response is not shown here (due to lack of time).

4 Further design considerations

Discussion was made concerning the handling of the repository chain for groups, the automatic detection of group state divergence in a group member, the ability to start/stop and refresh members from the coordinator, and the means by which this whole system might co-exist with an AMS-controlled group.

4.1 Agent-based control

Agents, in separate processes, stil provide the only feasible way of starting/restarting a dmServer member externally. This also allows extra function (for example bootstrapping/spawning dmServers without a pre-configured image).

It was therefore decided that an agent-based architecture should at least be considered.

bin

config

transient

spawning; used to start/stop; not limited to group function.

4.2 Group control with agents

multiple servers under one agent

multiple groups represented on the same platform under single agent

coordinator talks directly to the group members; actually communicates via the appropriate agent

agent process can be made minimal.

4.3 Proposed system breakdown

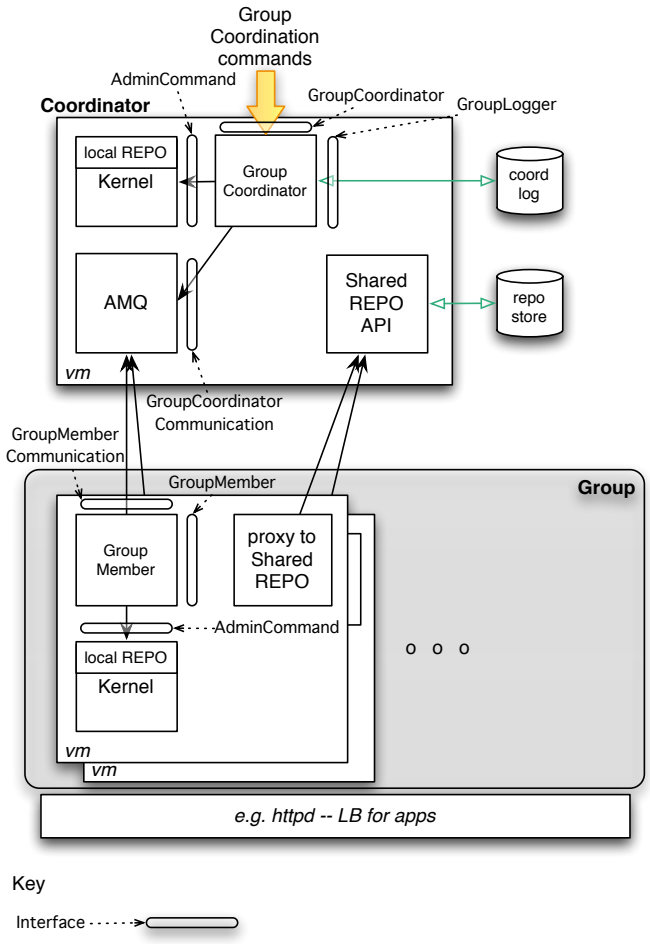

Figure 5: First cut process breakdown.

The diagram in Figure 5 shows an initial stab at breaking the problem into processes. The work was frozen before these separations were validated.

5 Freezing

The groups design is now frozen (12 February 2009) due to the following considerations:

1. tcServer AMS group support should co-exist with dmServer group support – and probably look like a natural subset of dmServer stuff;
2. monitoring of group members (as a group) is still only feasible with AMS-like support (without a lot of work);
3. the complexity of the work looks high – not possible before June;
4. the agent-based architecture suggested above is a duplication of the AMS agent stuff, and is likely to be incompatible and difficult to make simple and smooth.

For these (and some other external reasons) this project is now frozen, but remains in the commercial project set until the future direction of SpringSource server group support is better known.