University of Washington's upgrade to 2.1.2

This is a summary of the University of Washington's experience migrating from an Grouper 1.6 to Grouper 2.1.2. As you know the principal difficulty in this upgrade is the database conversion.

Some requirements of the upgrade

No interruption of service

Our group service (GWS), a RESTful API using Grouper's API as a back-end, is a core service at the UW. It is involved in a great many computer-computer and user-computer interactions---all day and night. We se about 200,000 - 300,000 queries per day, mid-quarter. More than that around the quarter breaks. Even during the night there are no hours with fewer than several thousand hits. Maybe 5% of these are updates.

So a first requirement was that this service is not to be interrupted.

New database

Our DBAs have been after us for some time to move to postgres version 9. It has some abilities that really benefit their operations.
Our sysadmins have been after us for some time to move our database off old hardward and onto new, virtual systems.
So this migration seemed like a good opportunity to accomplish both those tasks as well.

Steps in the upgrade

Update software

Updating our own code to work with the 2.1.2 API was not difficult, as the new API is mostly compatible with 1.6. Mostly we had to deal with the new requirement that subject lookups need a session. There is supposedly a setting, "subjects.startRootSessionIfOneIsntStarted=true", that defeats the new rule, but I found it not effective. Since that setting seemed like a temporary shortcut anyway, we fixed our code to always have sessions.

Couple of minor items:

1. tomcat7 uses 'DeployTask' instead of 'InstallTask' in build.xml
2. database updates in hooks don't complete unless "ddlutils.use.nestedTransactions = true"

Update database

This was the bigger problem. Our tests found a couple of problems with the upgrade procedures:

1. usdu() didn't populate the new member fields due to a bug in the subject classes. This is fixed in the most recent grouper release I think.
2. The database operations take a long time. Just dumping the database and loading it into the new system took 45 minutes.
3. I've never much liked the "stop service, make bunch of hard to undo changes, start service, hope it works" upgrade method for any service.

Enter clustered group systems

For some time I've imagined clustering independent group systems, using messaging to keep them all in agreement. Because our GWS service is RESTful the only things that need to be sent from one system to the other are resource representations. There are some difficulties to be solved before such a cluster could be put into general service, but even without those this method can provide an easy upgrade path:

1. We have our existing, Grouper 1.6, group service (gws1)
2. Create a new, Grouper 2.1.2, group service (gws2). New hardware, upgraded postgres, updated code.
3. Set up a message queue (aws1) that sends updates from gws1 to gws2
4. Set up a message queue (aws2) that sends updates from gws2 to gws1
5. Start the aws1 sender, and the aws2 sender
6. Dump gws1 and load it into gws2, start gws2
7. Start the aws1 receiver (on gws2). This brings gws2 up to date with gws1.
8. Start the aws2 receiver (on gws1). This brings gws1 up to date with any changes made on gws2.

Now we have two independent, mutually up-to-date GWS systems. The production GWS web service is itself a cluster, so each member can be dropped from the cluster, upgraded and connected to gws2, and put back into the cluster. During this migration some users will be on gws1 and some on gws2. When it's completed all our users will be on the upgraded service (gws2), without ever knowing they had moved.

Summary

Our upgrade pretty much went as outlined in the previous step. At the end we took the old system (gws1) out of service and retired its hardware.

The cluster approach shows promise as a means to a very scalable and resilient groups service. We'll be looking into this.