Managing Backward Compatibility

Managing On-wire Backwards Compatibility

Client/server and peer-to-peer communications in Geode have backward compatibility requirements. This page describes the infrastructure provided to meet those requirements. First we'll look at the Version infrastructure and then discuss client/server, peer-to-peer and persistence requirements.

Geode versions

The class Version lists all versions of the product and is used to implement on-wire backward compatibility between servers/clients and between peers.

All client/server and server/server (aka peer to peer) communication in Geode uses Data Serialization, which provides versioned streams (see VersionedDataInputStream and InternalDataSerializer as starting points) that can be queried to see what version the recipient or sender of a message is using. These are used to maintain on-wire compatibility between supported versions.

Classes that implemented `SerializationVersions` can note which versions the on-wire form of the class has changed and implement `toDataPre` /`fromDataPre` methods that Data Serialization will invoke based on the version of the sender/receiver. For instance,

```
    private static final Version[] dsfidVersions = new Version[] {
        Version.GFE_66, Version.GFE_70 };

    @Override
    public Version[] getSerializationVersions() {
      return dsfidVersions;
    }

    public void toDataPre6_6_0_0(DataOutput out) throws IOException {
      // serialize for versions prior to 6.6
    }

    public void toDataPre7_0_0_0(DataOutput out) throws IOException {
      // serialize for versions 6.6 up to 7.0
    }

    @Override
    public void toData(DataOutput out) throws IOException {
      // serialize for the current version
    }
```

This is the preferred way of implementing backward compatibility in Data Serialization, but for some changes or in superclasses you may find it easier to check for a versioned stream in the regular `toData`/`fromData` methods using `InternalDataSerializer.getVersionForDataStream`.

```
    public void fromData(DataInput in) ... {
      Version senderVersion = InternalDataSerializer.getVersionForDataStream(in);
      short bits = in.readShort();
      short extBits = 0;
      if (senderVersion.compareTo(Version.GFE_80) >= 0){
        extBits = in.readShort();
      }
      ...
```

Unit tests include AnalyzeSerializablesJUnitTest that has a set of "sanctioned" information about serializable classes and data-serializable classes. If this unit test fails it means that backward-compatibility may have been broken by your changes. The output of the test will note which classes were affected and what to do about it.

Client/server backward compatibility

Client/server backward compatibility must work for all releases. All previous versions of clients must be supported because they may be embedded in applications that users would find difficult, if not impossible, to track down and upgrade.

For clients there are command sets in the class `CommandInitializer`. When new client/server messages are needed we create a new command set and associate it with the new version. Clients send messages to servers using the client version and the server must be at the same or newer version. The client tells the server its version and the server uses the appropriate command set and deserialization behavior for that client.

Peer to peer compatibility between major releases

Geode servers are connected to one another via a peer-to-peer distributed system. Since Geode is an always-up system it supports rolling upgrade between both minor and major releases between peers.

This means that new distributed algorithms must be able to tolerate the presence in the distributed system of older versions of Geode and make allowance for how they behave. The algorithms can't assume that all peers will understand the new algorithm.

Changes to existing distributed algorithms can also be difficult to implement so that they allow both the old and new behavior of the algorithm.

Once a rolling upgrade begins you can count on three things: 1) peers using older versions of the product will no longer be able to join the distributed system, 2) no "schema" changes will take place during the upgrade, and 3) clients using the new version will not be used until the system is fully rolled to the new version.

Persistent file compatibility

Persistent file compatibility between releases is still somewhat ad-hoc. For minor versions, newer versions should be able to start up using the persistent files from old versions. For major versions, ideally we will also allow new versions to start up from the persistent files of older versions. However, for some releases we may choose to provide a conversion tool that must be run before the new version will recover the files.