VSL Design Notes

December 5, 2011

Contents

This document is intended to accompany and clarify the data model and UML files for the VSL design. It should be compact. More details explanations and ideas should be left to the *p2p_fs.pdf* file.

1 Terminology

Diff model We support flexible diff models ... e.g. binary diffing, data chunking, etc... The VSL is diff model neutral with the diff-model specific data (e.g. chunk number, etc...) stored in a implementation-specific Data Chunk Header.
UPDATE: while the above is still somewhat true we will actually favor a chunking model since a diff model would require a local fair amount of local statefulless to track updates to the file. With chunks we can just get the md5sum of each chunk in the backend and compare that with local chunking.

Auth/Perm model We are also auth-model neutral. All entries in the VSL backend have a permissions object which is implementation specific. As an example consider the following model: all data in a version is encrypted with a symmetric version key (generated for each version). The version key in turn is encrypted with the public key of any user/group that should have access and all these encrypted copies are stored in the permissions object.

Signatures All data in the VSL should be signed to avoid tampering. The signature is usually an md5 hash of the data encrypted using the version key of the version or the public key of all owning users/groups.

2 Data Model

2.1 Objects

The data model is built around two immutable objects, a *version* and a *data chunk*, as well as a mutable object, the *EntryHeader*. Although the latter can be modified we only allow "puts" not "pops" so data integrity is never an issue. Finally a special data type, *Index*, is used to store data that is subject to many small and frequent modification (e.g. an index of messages or even a directory) for which complicated version would entail too much overhead.

2.1.1 Index

An index is essentially any "put-mostly" list. While we could implement this as a special kind of file we do not because indices are accessed differently. They are subject to many small and frequent changes and creating new versions for each modification does not seem efficient. So instead the index is a single MMap entry which contains all its own version hisotry and is immune to write conflicts.

Basically an index is stored as a version tree. Data is simply put into the relevant backend entry along with a unique identifier and a version number. To update or delete data from an index a new record is added updating the status of an older record (along with the relevant version number). If two entries refer to the same record version with conflicting operations this is simply treated as a branch event.

Note there are two notions of versioning built into an index. The first is associated with each entry. Every entry that is added gets an *indexEntryID* and updates to that entry must refer to the previous version of the entry they're updating. To maintaime some sychronized notion of a timeline however there is another indexVersionID that spans multiple entries. Thus when updating an entry in an index the user must provide both the ID of the last version of the entry to be updated and the ID of the latest entry in this index (even if the latter is not the entry that is being updated). This allows us to track when two users modify different entries at the same time (which is usually fine but we may still wanna track it).

The Index header stores ownership, etc... info for the index. Having such an object requires *two* backend queries to get the index so if its not useful we may eventually drop it.

vslID	The unique vsl identifier for this entry.
permissions	A serialized object implementing a permissions system (see below). Note this only controls read-access (i.e. keys) to the data in the entry. Write access should be controlled via the storage layer.

The index's data is all stored in a single multi-map address associated with the index. The data is *unordered* and each entry is an *IndexEntry* object

indexEntryID	A unique ID generated for this entry.
prevEntryID	A reference to any older version of the same object in this index list. If this is NULL then we're adding a new object. If it is not null then this entry represents an update.
versionID	A unique ID generated for the new "version" or state of the index represented by adding this entry.
prevVersionID	A reference to the previous "version" of this index that this entry was supposed to be a modification of. We keep track of this to get some kind of a "snapshot" of the index.
permissions	A serialized object implementing a permissions system (see below). Note this only controls read-access (i.e. keys) to the data in the entry. Write access should be controlled via the storage layer.
indexEntryData	The actual data associated with this index entry.

2.1.2 EntryHeader (put-only)

An entry header provides the first point of access to an entry. It is essentially a list of version headers.

The entry header is essentially a "drop-box" where new versions associated with an entry are registered. Anyone with the correct key should be able to add a version to the list below but we do not allow version deletion or removal for now.

vslID	The unique vsl identifier for this entry.
permissions	A serialized object implementing a permissions system (see below). Note this only controls read-access (i.e. keys) to the data in the entry. Write access should be controlled via the storage layer.
versionEntry	Each entry is a version header with the following info: • vslID of version data • verionString • prevVersionString • timeStamp

2.1.3 Version (Immutable)

Versions are serializable objects containing *all* the data describing a version.

Note that version data is immutable. Once written to the system it will never be udpated again (though eventually it may be deleted).

It contains the following data:

vslID	The unique vsl identifier for this version.
versionStringd	A unique version string, preferably generated from the data for the version itself. Note for now we are keeping this seperate from the vslID but there may be reasons to have them be the same.
prevVersionString	A reference to the version preceeding this one. To allow merges we might need to allow this to be multivalued.
signature	An m5sum of the data encrypted with the version key. This prohibits tampering by the host peer.
permissions	A serialized object implementing a permissions system (see below).
chunks	A list of vslIDs reference to the data chunks associated with this version.

2.1.4 Data Chunk (Immutable)

Data chunks are also serializable objects encoding the changes associated with a version. Depending on the diff model there may be more than one Data Chunk per version and they can be stored in one or more mmap entries (ideally one for faster access but we should be flexible).

Data chunks are also immutable. They will never be updated or replaced though some day we may implement some kind of garbage collection.

vslID	The unique vsl identifier for this chunk (or list of chunks).
signature	An m5sum of the data encrypted with the version key. This prohibits tampering by the host peer.
permissions	A serialized object implementing a permissions system (see below).
chunkHeader	An implemenation specific (serializable) object providing chunk metadata. E.g. if a chunk represents a file chunk then this would contain its location (order) in the chunked file.
chunkData	An implementation specific (serializable) object proving chunk data. This could be e.g. just a chunk of a file or it could be a binary diff.

2.2 Storage

The objects above should be fully serializable (including all contained objects). They are stored in the MMAP by just serializing them and putting them in.

2.3 Structure: A VSL "Entry"

A VSL entry corresponds to a particular MMAP entry: the EntryHeader. The latter should hold a completely unodered list of all serialized versionHeader objects. The latter can be used to reconstruct the version history. They define the entry in the VSL. As should be evident they are essnetially write-only and can be "put" in any order.

More detailed information about a version can be found by following the vslID of the version in the versionHeader. This entry contains the full version info including the list of data chunks and their locations as well as version metadata.

Of course the version objects only refer to the actually data which is stored elsewhere as DataChunks. A version might either contain a list of MMAP ids corresponding to multiple DataChunks or it might contain only one.

3 Object Model

The object model is a bit more sophisticated than the Data Model but not much. We use the data assocaited with versions and chunks to create an "Entry" but converting an entry into a real piece of data (i.e. a file or whatever) is implementation specific – the VSL only knows how to store it and version it, not how to reconstruct it.

3.1 Class List

3.1.1 Core Classes

- VSL - the main class representing the system.
- vslEntry - a class representing a single entry stored in the VSL.
- vslVersion - a class representing a version of an entry.
- vslDataChunk - a class representing an immutable piece of data in the VSL.

3.1.2 Interfaces

These are interfaces that the core system interacts with. Implementations of these are necessary for the system to function.

- vslBackEnd - a backend storage layer for the VSL.
- vslBackEndEntry - a piece of data we can store in the backend.
- vslBackEndEntryMutable - a mutable entry that we can "put" to but not delete from.
- vslData - represents a data type that can be stored in a VSL such as a file, an image, a message. Implementations provide a particular chunking/diffing algorithm.

3.2 Classes

3.2.1 VSL

This is the central VSL object.

vslBackend An instance of a VSL backend such as a local MMAP or a P2P wrapper.

Entries A list of entries that this VSL knows about. They can exist in various states of synchronization.

Future add(Entry) Create a new entry in storage backend and return an ID. We return a Future to allow for asychronous writing.

Future update(Entry)

Future get(id)

4 Use Case Pseudocode

4.1 Add New File

Lets do a new file use-case, ignoring for now issues of authentication and additional data.
INPUT: file
OUTPUT: entryID

```
fileData = new vslFileData(filename)
{
   chunk file data;
   new Chunk {
      header: order, hash, length, begintoken, endtoken, timestamp
      data: chunk data
```

```
   }
   set chunks[];
}
entry = new vslEntry.new(filedata);
{
   version = new vslVersion(filedata.chunks[])
   {
      set prevVersion = null;
      /* note this unique version string is not the VSL ID */
      gen new version string;
   }
   header = new vslEntryHeader(version)
   {
      versionList.add(version);
   }
}
entry.store()
{
   versions.create()
   {
      chunkIds[] = chunks.create();
      store chunkIds;
      store version data: versionstring, chunkIds[], timestamp.
      return versionID;
   }
   header.create( )
   {
       store header data: versionList, fileName;
       return headerID;
   }
   set headerID;
   return headerID;
}
```

4.2 Get File

Here we retreive a file from the system.
INPUT: entryID
OUTPUT: file

```
entry = new vslEntry(entryID);
entry.loadHeader();
vtree = entry.buildVersionTree();
if (vtree.head().length() == 1)
{
   vslEntry.getVersion(vtree.head());
   {
      version = new Version(vtree.head().vslID());
      version.load()
      {
         /* read version entry from storage */
         version.header = version.getVersionHeader();
         /* read the data associated with all the chunks */
         foreach(chunk = version.header.chunk[])
         {
```

```
            chunk.load();
         }
      }
   }
   file = new FileData(version)
   {
      // read chunks out of version and build a file out of them
   }
   return file;
}
```

5 Issues

Although this section promises to grow without bound lets keep a running list of issues with the design so we can keep are sanity while trying to move forward.

5.1 Read performance

There are several conflicting factors affecting read performance. Ideally we want to read off as little data as we need for each update but this causes several problems. If we store all the data in as few MMAP entries as possible we save on the number of get's but we pay for it by downloading unnecessary data so looks like we need to store, e.g. chunks, seperately. Ideally we could implement some kind of locallity in the name scheme.

5.2 Chuking vs Diffing (or the needly for caching)

A significant benefit of chunking, etc... is not just data integrity but reducing network usage as in principle we should only need to download new versions of the data. But this can cause problems that a particular implementation will have to overcome. If I have a local file how that is being stored updated and stored in the VSL how do I generate a new version. I either need a local copy of the last version (to diff against), or I need to redownload the latter. This is where chunking has a large advantage over diffing. The only way to make diffing equally efficient is to keep a local cache of the last version and the version history and this becomes quite storage inefficient.