PROPOSAL FOR NEW “MAPPING ZOO” ID GENERATION SERVICE ARCHITECTURE

Version 1.0

March 9, 2010
D.Rosenstrauch
Demdex Inc.

Introduction

As explained in the “PROPOSAL FOR NEW EVENTPUBLISH AND RANKING/SCORING ARCHITECTURE” document, the new eventpublish process requires a “Mapping Zoo” service in order to supply unique ID’s for the new records being published. This document describes a proposed architecture for implementing this Mapping Zoo service.

Statement of problem

Since the new eventpublish process will be run as jobs inside of Apache Hadoop, by definition it will be run in a distributed, asynchronous fashion. And, like nearly every system run in such a distributed fashion, the eventpublish process will be faced with the classic problem of how to make its distributed processes coordinate and synchronize access to shared resources.
Fortunately, the creators of Apache Hadoop – once again following the inspiration of the engineers at Google – have developed a solution to this problem: Hadoop Zookeeper.

Hadoop Zookeeper background

Core Zookeeper functionality

Hadoop Zookeeper is a “distributed coordination” service for Apache Hadoop. In a nutshell, it provides the capability for a collection of distributed processes to read and write shared data in a “threadsafe” fashion – i.e., ensuring that the integrity of the data is preserved even when multiple processes are trying to write to it simultaneously.
Zookeeper accomplishes this by providing the user with the ability to perform CAS (“compare and set”) operations. Whereas typically performing a “setData()” operation on some object might look like this:

```
boolean success = object.setData(newData);
```

… the API for a CAS-based setData() operation might look more like this:

```
try {
       int newVersion = object.setData(newData, oldVersion);
}
catch (CASException e) {
…
}
```

What happens in a CAS operation, then, is that an update of the data only occurs if you have supplied the correct version number of the data. If you attempt to perform an update using an old version number – meaning that the copy of the data that you are holding in memory is out of

date – then the update is rejected. The code should then go and retrieve the latest version, and then re-try the same update.

This CAS functionality is a low-level synchronization primitive. But it is quite powerful, in that it can then be used as a building block to implement more complicated higher-level synchronization primitives, such as semaphores, barriers, etc.

And, more importantly from the perspective of the eventpublish process, it can be used to build a threadsafe "counter" primitive, with which we can implement the core of our eventpublish ID generation service. E.g.:

```
public void incrementCounter() throws CounterException {
	boolean counterUpdated = false;
	int numFailures = 0;
	while (!counterUpdated) {
		DataAndVersion counterAndVersion = counterService.getData();
		int counter = ((Integer)counterAndVersion.getData()).intValue();
		int version = counterAndVersion.getVersion();

		// increment the counter
		counter++;

		// save the updated counter
		try {
			counterService.setData(counter, version);
			counterUpdated = true;
		}
		catch (CASException e) {
			numFailures++;
			if (numFailures > FAILURE_THRESHOLD) {
				throw new CounterException("Unable to update counter");
			}
		}
	}
}
```

Other Zookeeper functionality

There are several other noteworthy aspects of Hadoop Zookeeper which make it a good choice for our ID generation service:

- **Zookeeper itself is a distributed system.** Zookeeper itself is a distributed, multi-node system. This makes it:
  - very reliable, as there is no single point of failure, and
  - very available and performant, since slowness or downtime on one Zookeeper node does not impact all the other nodes
- **Zookeeper is an in-memory database.** Zookeeper holds all its data in-memory. This makes it quite fast, as it does not need to perform disk seeks on reads or writes to/from its data.
- **Zookeeper is a key-value store.** Aside from the all the added synchronization and compare-and-set functionality, Zookeeper is basically just a key-value store. You can store a value into it which is associated with some key, retrieve values from it by key, overwrite existing values by key, or delete a key/value mapping.
- **Zookeeper's key name space is hierarchical.** The namespace of the keys in Zookeeper is arranged hierarchically, forming a tree. Although Zookeeper provides the ability to

traverse the tree (e.g., access all children of a parent node) we will likely not need to use this functionality in the ID generator, and instead rely solely on its key/value mapping functionality. The naming syntax for Zookeeper keys is similar to that of a unix file system. So some examples of key names might be "/apps/app1/key1", "/users/user1", etc.

Constraints on the design of the ID generation service

Zookeeper is an excellent foundation on which to build the ID generation service, for the reasons listed above. That said, there are still some considerations that need to be taken into account on the design of the service, else we run the risk of the service causing unneeded network overhead, and thus performing poorly.

1. **The ID generation service should be implemented as a library.** It would be poor design if the caller that required an ID (i.e., one of the map/reduce tasks in the eventpublish process) were required to perform all the low-level tasks of interacting directly with Zookeeper. Instead, it would be much better to create a library whose sole responsibility is to interact with Zookeeper to get/set ID numbers. The calling code, would then only interact with the (most likely very simple) API of the library, making the calling code much simpler and better designed.
2. **New ID's need to be generated in batches.** Every update to Zookeeper causes the update to get propagated out to all other nodes in the Zookeeper cluster. As a result, if we were to make a call to Zookeeper for every ID that we generate, the Zookeeper cluster will quickly get overwhelmed with network I/O and become non-performant. Therefore when a map/reduce task in the eventpublish process needs to assign an ID, it must ask for a large batch of ID's (most likely somewhere between 1000 and 10000 ID's), not just a single ID number.
3. **Unused ID's need to be "recycled".** Although, as per constraint #2 above, an eventpublish map/reduce task needs to request ID numbers in large batches, it may wind up not needing to use all of the ID's that it has been assigned. (e.g., if the task requests 10000 ID's … on the last event that it is publishing) But it will not be acceptable for the task to just throw away the remaining unused ID's. Although the ID counter is a fairly large number (a 32-bit integer, with a max value of > 4 billion) and should last us for a while, it won't last us for very long if we keep throwing away blocks of ID numbers! Therefore, the ID generator library needs to provide the capability for the caller to "push back" unused ID's that it has been assigned. These ID's should then get "recycled" back into the list of available ID's, and assigned again to other callers in subsequent ID number requests.

Proposed design

Overview

The API for the new ID generator library would look something like this:

```
public class IDGeneratorService {
	public IDSet takeIDs(IDCategory category, int numIDs) throws
IDGeneratorException {
	…
	}
```

```
        public void pushIDs(IDSet ids) throws IDGeneratorException {
        …
        }
}

public class IDCategory {
        public String getName() {
        …
        }
}

public class IDSet {
        public IDCategory getCategory() {
        …
        }

        public Iterator<IDRange> getRanges() {
        …
        }

        public int takeID() {
        …
        }

        public boolean hasMoreIDs() {
        …
        }

        public int countIDs() {
        …
        }
}

public class IDRange {
        public IDCategory getCategory() {
        …
        }

        public int takeID() {
        …
        }

        public boolean hasMoreIDs() {
        …
        }

        public int countIDs() {
        …
        }

        public int getStartID() {
        …
        }

        public int getEndID() {
        …
        }
}
```

Calling code would look something like this:

```
public void publishEvents {
…
	IDCategory  demdexIDCategory =
		new IDCategory(“/com/demdex/eventpublish/idgen/did”);
	int idSetSize = 10000;
	IDSet idSet = null;
	try {
		while (moreEventsToPublish) {
			// grab an ID range
			if (idSet == null || ! idSet.hasMoreIDs()) {
				idSet = generatorService.takeIDs(demdexIDCategory,
		idSetSize);
			}

			// assign an ID to each event and publish
			while(idSet.hasMoreIDs()) {
				int eventID = idSet.takeID();
				publishEvent(eventID);
			}
		}
	}
	finally {
		if (idSet!= null && idSet.hasMoreIDs()) {
			// push back any leftover ID’s
			generatorService.pushIDs(idSet);
		}
	}
…
}
```

Implementation Detail

Implementation of the ID generator service should fortunately not be too complex. As described earlier, Zookeeper’s core functionality is as a key/value store. The values that we will be storing into it will just be simple lists of ID number ranges, such as “5000:10000”.
(For simplicity’s sake in the following examples, let’s assume a maximum integer value of 123456789.)
Initially, we would seed the ID generator node with the full range of available integer values. So the contents of the node would look as follows:

```
/com/demdex/eventpublish/idgen/did = {
	1:123456789
}
```

Now, let’s assume that 3 map/reduce tasks each perform a “take ID’s” operation, requesting 10,000 ID’s. So task 1 gets assigned the range 1:10000, task 2 gets assigned 10001:20000, and task 3 gets assigned 20001:30000. The contents of the ID generator node after this would now look like:

```
/com/demdex/eventpublish/idgen/did = {
	30001:123456789
}
```

Now, let’s assume that after each of the map/reduce tasks completes, each of them has 1000 ID’s left over. Each task would then “recycle” its left over ID’s by making a “pushIDs” call using its list of remaining ID’s. After these 3 operations, the contents of the node would look as follows:

```
/com/demdex/eventpublish/idgen/did = {
```

```
        9001:10000
        19001:20000
        29001:30000
        30001:123456789
}
```

Note that the ID ranges are always pushed back into the list in ascending order. This ensures that the oldest "holes" in the ID sequence get "filled" as soon as possible.

The generator node would likely remain in the above state for a while – i.e., until the next time that event publishing gets run. When it next does get run, the next set of 10000 ID's to be handed out would wind up consisting of 4 non-contiguous ranges, which together add up to 10000 ID's: 3 ranges of 1000 ID's (9001:10000, 19001:20000, and 29001:30000), followed by 1 range of 7000 (30001:37000). The contents of the generator node after this operation would then be:

```
/com/demdex/eventpublish/idgen/did = {
        37001:123456789
}
```

Processing would then continue on repeatedly as described above.

The only exceptional conditions I can foresee for this algorithm are as follows:

1. **Problem:** A caller requests more ID's than are left available in the range (e.g., request 10000, but only 1000 are available). **Solution:** Hand out only as many ID's as are available (1000, in this case).
2. **Problem:** A caller requests ID's, but there are none available. **Solution:** Throw an exception to the caller.

Note:
Although there are arguments to be made for storing the ID range list in a binary format (e.g., smaller storage, less network I/O, faster parsing, etc.) I believe that it would be better to store them in a text format. The ID range data in the node will never be very large (in the 10's of kilobytes range, maximum), and so will never be a big bottleneck in terms of storage, network I/O, processing CPU, etc. And a text format would provide much better end-user clarity for anyone needing to support or debug the ID generator.