Distributed data structures over a fuzzy shared log

Advisor: Mahesh Balakrishnan

Tango provides developers with the abstraction of a replicated, in-memory data structure (such as a map or a tree) backed by a shared log. This allows only for a strong consistency model, whereas different consistency guarantees with varying efficiency tradeoffs could be offered. To achieve this, we propose replacing the totally ordered shared log with a partially ordered directed acyclic graph.

The goal of this project is to explore the usefulness of such a system. The final product will be include a dummy runtime implementation of the fuzzy shared log on a single machine, an in-memory data structure that is backed by the log, and an application that uses this structure.

Tango background

Tango aims to provide a way to store metadata by allowing transactional access to arbitrary data structures with persistence and high availability. The Tango paper notes that some services such as ZooKeeper provide persistence, high availability, and strong consistency—but only for specific data structures.

Tango objects are a class of data structures stored in memory that built on the shared log. The history of the object is stored directly in the log, and each object can have multiple views which are copies of the data structure stored by the clients. The source of truth in the system is the shared log, and the views are "soft state" that can be reconstructed by replaying the shared history. Modifications to Tango objects are made by appending updates to the history, and accesses are made by syncing the view with the history.

Using a shared log makes strongly consistent operations easier than they would be in a conventional distributed system. Such applications include remote mirroring, coordinated rollbacks, and consistent snapshots across objects. These options are available since views can be synced to any offset in the log and replayed as needed.

Directed acyclic graphs

The shared log abstraction represents a total ordering of the updates added to the log. By definition, this means that it is a set with a relation that satisfies reflexivity, transitivity and symmetry, where any two items can be compared. On the other hand, we want to represent a partial ordering of the updates so that we can allow the user the flexibility of not relying on other users writes. A directed acyclic graph is a data structure that can represent partial ordering relations. Therefore, we propose using a shared DAG as a representation of state in the system.

Consistency guarantees

Not all applications require a strong consistency guarantee. Depending on the particular application, lower latency reads or the ability to write immediately could be preferred. We would like to enable our users to select what level of consistency works for them. Rather than have the consistency level be specified at development time, we would like to support this at the API level. By having something like a Service Level Agreement specified within the update_helper and query_helper, the client could configure the guarantee that works best in the situation. The SLA specified would then inform where in the DAG to place the update. The motivation for such a system is that different procedures in an application can have different guarantee requirements.

SLA requirements we may support include:

strong returns the value of the last preceding Put(key) performed by any client

read-my-writes returns the value written by the last preceding Put(key) in the same session or returns a later version; if no Puts have been performed to this key in this session, then the Get may return any previous value as in eventual consistency.

bounded(t) returns a value that is stale by at most t seconds. Specifically, it returns the value of the latest Put(key) that completed more than t seconds ago or some more recent version.

causal returns the value of a latest Put(key) that causally precedes it or returns some later version. The causal precedence relation $<$ is defined such that op1 $<$ op2 if either (a) op1 occurs before op2 in the same session, (b) op1 is a Put(key) and op2 is a Get(key) that returns the version put in op1, or (c) for some op3, op1 $<$ op3 and op3 $<$ op2

Any of these consistencies could be supported with the DAG structure, as long we implement an efficient way to stream through the structure.

Note that these potential consistency requirements come from the *Consistency-Based Service Level Agreements for Cloud Storage* paper.

Deliverables

Emulator of the fuzzy log

I will write an emulator of the fuzzy log that exposes our object API. The details of the API are not fully decided, but at minimum, it will specify the *update_helper*, *query_helper* and *SLA* requirements.

The *update_helper* will accept a buffer that will be added onto the DAG. The *query_helper* will read from the DAG and provide the object with the updates via the *apply* upcall. These functions are similar to the requirements of a Tango object.

To allow for flexible consistency, the *SLA* requirement requested by the system object must be interpreted by the fuzzy log. Depending on whether the request came from the *update_helper* or the *query_helper*, it would have to find or append to the correct part of the DAG.

It would be out of the scope of this project to fully implement the DAG. Rather, I will build an application over a dummy fuzzy log to denonstrate its usefulness.

Distributed data structure

To support the example application—a distributed text editor—I will create a data structure over the fuzzy log. This would be a buffer view that the application could use to interact with the file. The application in this case could tolerate weaker consistency requirements than are provided by Tango.

When editing different parts of the file, for instance, eventual guarantees would suffice among machines. However, when editing the same parts of a file, strong guarantees would be needed. To effectively implement this data structure, careful consideration will have to be given to the choice of SLA during different editing procedures.

Application layer

As a stretch goal for this project, I will build a text editor that uses the text buffer view. The text buffer view will not behave exactly like a text buffer, since the application will need to make decisions as to what edits to group together to commit to the data structure. Depending on what level of abstraction we build into the data structure, a couple possibilities of implementation exist.

If the data structure requires significantly more management than a normal text buffer, then it may make sense to write a text editor from scratch. In that case, I would use the Qt C++ library to build the interface for doing so.

Otherwise, if the data structure requires minimal management, then I would create a vim plugin that supports our text buffer view.

References

Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, Aviad Zuck. *Tango: Distributed Data Structures over a Shared Log*

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, Hussam Abu-Libdeh. *Consistency-Based Service Level Agreements for cloud storage*