SyncFree Technology White Paper

Bounded Counters: maintaining numeric invariants with high availability

Valter Balegas, U. Nova de Lisboa

28 October 2016

Why Bounded Counters?

Having a presence on the Internet is essential for companies to sell theirs products and services. When events of interest for a large audience occur, for instance, a new smartphone from a major company is available for purchase, or a big franchise movie tickets go on sale, services often become overloaded with costumers that have interest on those things. It is in these situations that services platforms are put to test.

It is common to use data replication to improve the scalability and availability of storage systems. Many existing storage systems already provide replication out-of-the-box, but typically, they only provide eventual consistency across replicas. In these systems, a read operation might not reflect the most recent updates executed in the system, which might be problematic when multiple application servers execute updates based on the state that they observe.

The Bounded counter is a new data-type that allows improving the consistency properties that can be maintained in eventual consistency systems. The problem that the bounded counter solves is to guarantee that the value of a counter does not ever exceed some limit value, for instance that the value does not become negative, when multiple updates for the same counter are executed in different replicas.

Use of Bounded Counters

In the example of selling smartphones, the company might only have a limited quantity of devices in different geographical regions and want to ensure that the number of smartphones sold in each region does not exceed a certain limit. It is impossible to guarantee that constraint in systems that only ensures eventual consistency, because different clients might be able to buy the last units available concurrently: the application servers checks that the operation can execute in the local replica; the operations execute locally and succeeds, returning immediately to the client; when operations are propagated and execute in the remote replica, it might occur that the value of the counter becomes negative, which violates the defined constraint.

It might seem that it is not problematic to oversell a few extra units of the device, however, our experimentation shows that with an increase of clients trying to update a counter, the diverge also increases, which translates into having more oversold units when the number of users trying to acquire the device increases.

The bounded counter prevents the value of the counter from violating the constraint, by limiting the number of operations that can execute in each replicas before synchronizing the changes with other replicas. The design of the bounded counter allows the integration of this novel data-type in existing systems that support a convergent data-model, with little effort from the programmer.

Bounded counters are part of the Just-Right Consistency approach, ensuring availability while minimising synchronisation to precisely match application requirements. We refer the reader to the companion white paper: " Just-Right Consistency, or How to tailor consistency to application requirements."

Bounded-Counter design

The Bounded-Counter is a new CRDT that supports a numeric constraint for preventing the value of the counter from exceeding some limit. For simplicity, we assume that the constraint ensures that the value of the counter is always greater or equal to zero, but the data-type allows to set value. To ensure the constraint, the bounded counter stores the number of decrement operation executions that are allowed in different logical partitions. Typically, it is defined one partition for each replica, but the programmer might specify partitions with a different granularity.

The interface of the data-type supports *increment(val,prtId)* and *decrement(val,prtId)* operations, as normal counters, and a new *transfer(origId, destId)* operation to transfer permissions to execute decrements from partition with identifier *origId* to partition with identifier *destId*. Increment operations always succeed, as they can never violate the constraint. Decrement operations are safe when *prtId* has enough permissions to execute the operation, but when *prtId* does not have enough permissions, the operation fails with a error indicating that the operation is not safe. In this case, the replica that wants to execute the operation must increase the permissions of *prtId*, by requesting some replica to transfer permissions to that partition.

The data type works as any normal CRDT data-type and has state-based and operation-based implementations. The programmer is responsible for ensuring that operations for each partition are executed in sequence. Many systems already support operations serializability, which allows implementing this functionality. The programmer must also provide some policy to transfer resources between partitions.

Bounded-Counter implementation

We provide standalone implementations of the Bounded Counter in Erlang and Java. We have experimented adding the data-type to the Riak KVS, by extending the KVS with a middleware that serializes the execution of operations for the different partitions and handles permissions transference between different replicas in the system. The performance of the data-type was comparable to the performance of the counters that are provided by Riak.

Programmers willing to support bounded counters on their databases can use our prototype as reference for implementing the data type and the transference policy.

Bounded-Counter use-cases

The bounded counter design is useful in a number of situations. Also, the same design principle can be applied to other data-types to provide other constraints. We indicate a few situations where bounded counters are useful:

- Ensure that the Stock of products does not become negative;
- Ensure that tickets for an event are not oversold. Either with assigned numbers or undistinguished;
- Ensure that the number of prints of an advertisement does not exceed the budget that the client requested;
- Ensure that the account balance plus the credit of a costumer does not become negative;
- Implement a generic distributed lock that can be shared, or is exclusive, to enforce arbitrary constraints.