How to build a distributed counter

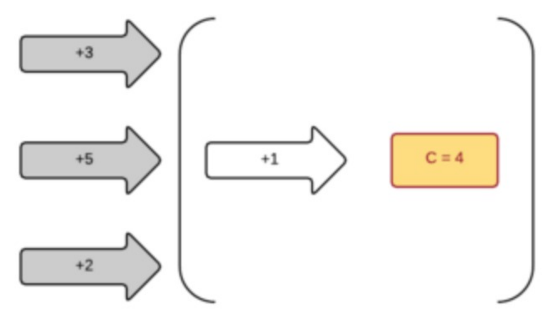
Posted by **Damien Bailly** on Wed, Nov 4, 2015

An introduction to CRDTs

A counter how simple is that?

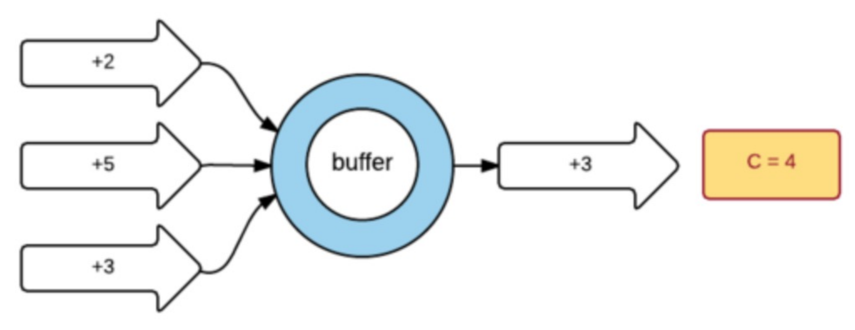
Surely, it's a value that is incremented by a given number and it yields a new value. Simple!... until multiple "users" try to increment the counter simultaneously. How can we handle counting in a distributed manner?

Easy! Use synchronisation to ensure that the counter is updated once at a time. That works, but we're enforcing sequencing of the operations and we lose parallelisation. How can we improve?

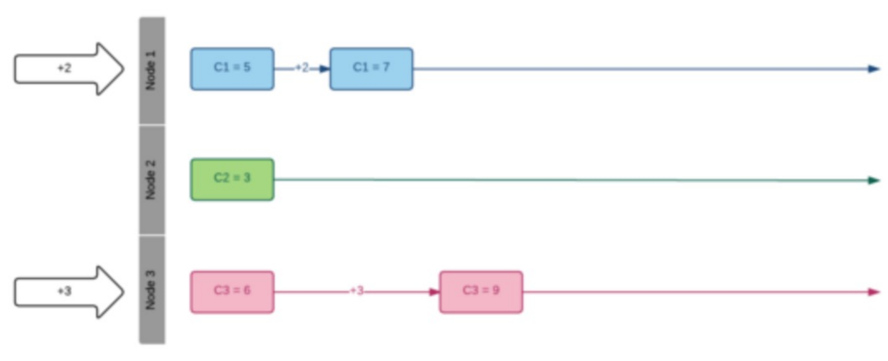


Yes get rid of locks! Use CAS (compare and swap) it improves throughput globally. Some "users" may be starving while they keep trying to update the counter under heavy load. So, it's better but still not ideal.

Alternatively we can have a single "user" incrementing the counter while the others push the increment requests into a queue. It does improve throughput but the counter itself is not really distributed.

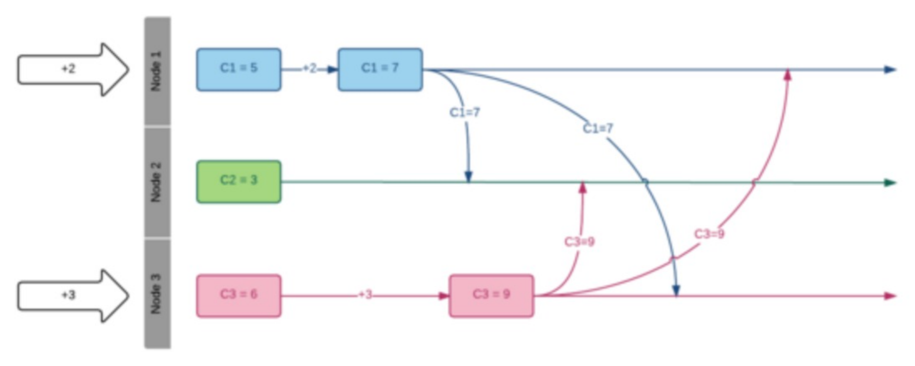


Now let's make this counter really distributed and split it: instead of one "writer" for the whole system let's have each user write to its own counter. Now each user (or node) can write when it wants as it only changes its very own value. And we can have parallel increments.



Ok that looks really distributed now but how to get the counter's value? Well we need to sum up all the individual counters.

So every node needs to know every other node's value to get a "global" view of the counter. Let's have the nodes broadcast their values.



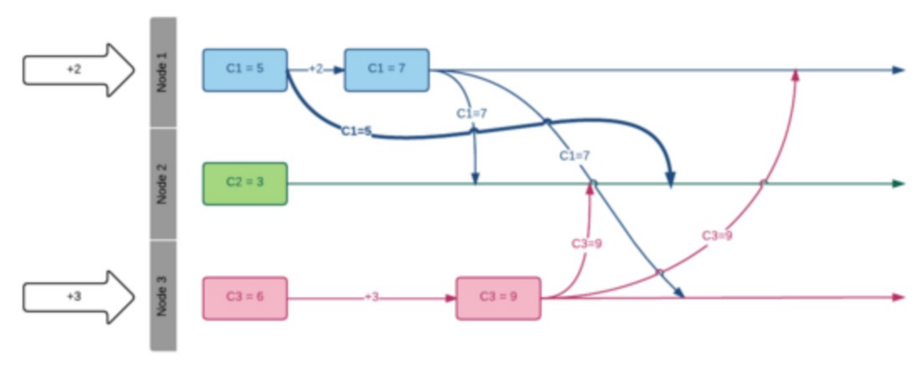
Interesting, look's like we're getting somewhere but a distributed world is not 100% reliable: messages can get lost, or be delayed, ...

Solving message lost is not a trivial task Let's consider

- On the sender side nodes can regularly broadcast their state.
- On the receiving side we need a way to find out what to do with duplicate messages. Again it's not too hard as a node can compare the received value with the value it already knows for this node. It the values are the same it's a duplicate message otherwise it's a new one so update the node's value.

One last thing to solve: delayed messages.

This is the tricky part - how to know if the received message is stale?



Well in a distributed world you can't (unless you rely on some sort of clock to order the events). In order to solve this issue we're going to impose a restriction on our system: let's assume that a counter can only move forward (no decrement allowed).

Now it's easy to find out what to do with the received messages: just keep the max between the known and the received value.

At this point every node knows what to do with the received messages and we have strong eventual consistency because as soon as a node has received all the events (in any order) it knows the current state of the counter and this state is the same for every node. There are no conflicts, all nodes eventually converge to the same value.

Actually these type of distributed data structures are well known and are called CRDT (for Conflict-free Replicated Data Type).

In our case we broadcast the local state (the counter value) so our counter lives in the world of state-based CRDT or CvRDT (for convergent CRDT). In fact we need 3 things to define a CvRDT:

- a set (e.g. positive integers)
- a partial order (e.g. values keep increasing)
- a binary operation that is associative, commutative and idempotent (e.g. the Max function)

These 3 properties form something called a semi-lattice and this is just what we need to define a CRDT. Associativity allows grouping in any order, commutativity allows to process messages in any order and idempotency allows to replay messages.

Another good thing with CRDT is that they do compose. Remember our counter limitation: no decrement. Well we can fix it by using a second counter which follows the same rules in order to track the decrements. Then the global value will be the sum of the increments minus the sum of the decrements.

CRDT are also fault-tolerant as only one node needs to be alive for the system to work. Other nodes will be able to automatically recover when they come back to life.

Now that we have a distributed counter let's quickly explore what other data types we can have:

- we can have flags (boolean) and registers
- we can have sets (more on that later)
- once we have sets we can have maps
- once we have sets we can have graphs (a graph is a set of vertices and a set of edges)
- once we have graphs we can have trees

Let's look at the set type. We need to impose a restriction (like we did for our counter) to solve conflicts. Let's make our set insert-only. That's a perfect fit for a CRDT!

Every node maintains its own local set, broadcasts its content and the "global" set is just the union of all the local sets.

Can we combine 2-sets (like we did for the counter) to have a fully operational set (with the remove operation)? Almost!

We just need a rule to decide what to do when a node removes an item that was added by another node. A common rule is that the "add" always wins but the opposite would be perfectly valid too. The key is that we need a rule that every node follows.

Another problem with state CRDT is that it broadcasts its state. It is fine when the state is an integer, it might not be as good for a set of potentially large objects. In such case we can broadcast the operations only (e.g "add X"). Such CRDT are called CmRDT (for Commutative CRDT) and are equivalent to CvRDT.

Crafting a data type to obey these rules can be tricky (especially for operation based CRDT). Quite often it relies on version vectors or vector clocks to ensure local ordering of the operations.

CRDT are used in the Riak database, on the SoundCloud platform (with Roshi), fit very well with akka's actor model and have been widely studied over the past years.

The aim of this post was to give a brief introduction to CRDTs and see how powerful and simple they are. However not every problem can be solved with CRDT, but if you can twist your problem so that it fits into the CRDT's world you'll benefit from all the simplicity and power of these data types.

Topics: Akka, Roshi, Riak

Valentin Tihhomirov 12/4/2016, 5:08:33 PM
How does the buffer improve CAS throughput? Isn't queue implemented using CAS? IMO, incrementing a global value under common lock is much easier than inserting a value into a global FIFO.

Reply to *Valentin Tihhomirov*

Comments

First Name*

Last Name*

Email*

Comment*

protected by reCAPTCHA
Privacy - Terms

Submit Comment

Recent Posts

This week in #Scala (04/06/2018)
This week in #Scala (21/05/2018)
This week in #Scala (14/05/2018)
New Relic Instrumentation for Kafka
This week in #Scala (07/05/2018)

Posts by Topic

Scala (439)
Akka (361)
Reactive (185)
DevOps (180)
AWS (155)
see all

Posts by Author

Jan Machacek (250)
Petr Zapletal (201)
Laura Bria (139)
Chris Cundill (67)
Mark Harrison (37)
see all

Subscribe to Email Updates

First Name*

Last Name*

Email*

Notification Frequency*

Instant Daily Weekly
Monthly

Subscribe

Subscribe via RSS