Distributed Computing
https://doi.org/10.1007/s00446-017-0322-2

Scalable eventually consistent counters over unreliable networks

Paulo Sérgio Almeida[1] · **Carlos Baquero**[1]

Received: 5 September 2014 / Accepted: 9 December 2017

Abstract
Counters are an important abstraction in distributed computing, and play a central role in large scale geo-replicated systems, counting events such as web page impressions or social network "likes". Classic distributed counters, strongly consistent via linearisability or sequential consistency, cannot be made both available and partition-tolerant, due to the CAP Theorem, being unsuitable to large scale scenarios. This paper defines Eventually Consistent Distributed Counters (ECDCs) and presents an implementation of the concept, Handoff Counters, that is scalable and works over unreliable networks. By giving up the total operation ordering in classic distributed counters, ECDC implementations can be made AP in the CAP design space, while retaining the essence of counting. Handoff Counters are the first Conflict-free Replicated Data Type (CRDT) based mechanism that overcomes the identity explosion problem in naive CRDTs, such as G-Counters (where state size is linear in the number of independent actors that ever incremented the counter), by managing identities towards avoiding global propagation and garbage collecting temporary entries. The approach used in Handoff Counters is not restricted to counters, being more generally applicable to other data types with associative and commutative operations.

Keywords Conflict-free Replicated Data Types · Distributed counters · Eventual consistency

1 Introduction

A counter is one of the most basic and important abstractions in computing. From the small-scale use of counter variables in building data-types, to large-scale distributed uses for counting events such as web page impressions, banner clicks or social network "likes", the presence of counters is pervasive. Even in a centralized setting, the increment operation on a counter is problematic under concurrency, being one of the examples most used to illustrate the problems that arise if a sequence of load, add one and store is not atomic. In a distributed setting things are even more difficult, due to the absence of shared memory, possibly unreliable communication (message loss, reordering or duplication), network partitions or node failures.

If one has a strongly consistent distributed database with support for distributed transactions, counters can be trivially obtained. Unfortunately, such databases are not appropriate for large-scale environments with wide-area replication, high latency and possible network partitions. A naive counter obtained through a "get, add one, and put" transaction will not scale performance-wise to a wide-area deployment with many thousands of clients.

The CAP theorem [5,12] says that one cannot expect to have Consistency, Availability, and Partition-tolerance together; one must choose at most two of these three properties. Therefore, to have an always-available service under the possibility of partitions (that in world-wide scenarios are bound to happen from time to time), distributed data stores such as Dynamo [9], Cassandra [22] and Riak [21] have been increasingly choosing to go with AP (availability, and partition-tolerance) and give up strong consistency and general distributed transactions, in what has become known as the NoSQL movement.

This work was partially supported by SMILES within project "TEC4Growth Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01- 0145-FEDER-000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF); EU FP7 SyncFree project (609551), and EU H2020 LightKone project (732505).

✉ Carlos Baquero
cbm@di.uminho.pt

Paulo Sérgio Almeida
psa@di.uminho.pt

[1] HASLab, INESC TEC, Universidade do Minho, Braga, Portugal

Published online: 19 December 2017

With no support for strong consistency and distributed transactions, and an API mostly based on simple get and put operations, obtaining a "simple" counter becomes a problem. NoSQL data stores like Cassandra have been trying to offer counters natively with an increment operation in the API, something that has revealed a non-trivial problem, involving many ad hoc evolutions. This issue in Cassandra is well summarized by [14]: "The existing partitioned counters remain a source of frustration for most users almost two years after being introduced. The remaining problems are inherent in the design, not something that can be fixed given enough time/eyeballs."

CRDTs (Conflict-free Replicated Data Types [27]) allow obtaining provably correct eventually consistent implementations of data types such as counters. The idea is that each node keeps a replica that can be locally queried or operated upon, giving availability even under partitions, but providing only eventual consistency: queries can return stale values, but if messages go through, then all nodes will converge to the correct value. CRDT-based data types trivially tolerate message loss, duplication or non-FIFO communication.

CRDTs are not, however, a silver bullet. A problem that can easily arise in CRDTs is scalability. One practical situation where this occurs, and that motivated our focus on CRDT scalability, is illustrated by the initial design of the advertisement counting system for banner control in Rovio's Angry Birds mobile game [1]. A game that at its prime listed 50 millions of users. If there is one replica per participating entity, each one with a unique identity, as many CRDTs keep maps with these ids as keys, these maps will keep growing over time, as more entities participate; this results in each replica having size proportional to the total number of entities that ever participated in the system, which is not scalable. Notice that we are not talking about one map entry per node in the cluster, a number that could be expected to be in the hundreds, but instead one map entry per edge client or short-lived thread handling a client, and thus facing hundreds of thousands of entries or more.

One possible approach is to restrict replicas to a small number of servers, excluding clients from the participating entities, in what concerns the CRDT. This will solve the scalability problem and allow unreliable communication between servers, but will not solve the fault-tolerance problem in the client-server interaction. This is because a basic problem with counters is that the increment operation is not idempotent; therefore, an increment request by a client (which itself does not keep a replica) cannot just be re-sent to the server in case there is no acknowledgment. This problem is well recognized by practitioners [23].

If one looks at theory of distributed counting, a substantial amount of work has been done, namely Software Combining Trees [13,33], Counting Networks [2], Diffracting Trees [28] and Counting Pyramid [32] (and their variants). However, all these works address a strongly consistent definition of counter, as a data type that provides a single "fetch-and-increment" operation in the sense of [29]. Although a powerful abstraction, capable of generating globally unique sequence numbers, it is indeed too powerful to be implemented while providing availability under unreliable communication. The focus of these works is scalability (mainly avoiding contention or bottlenecks) and not fault-tolerance. While some aspects like wait-freedom are addressed, to tolerate failures of processes, tolerance to message loss or component failure (e.g., a balancer in the case of counting networks) is not addressed. This means that these mechanisms are more suitable for tightly-coupled, low-latency, failure-free environments, such as multiprocessors, serving as scalable alternatives to lock-based counter implementations, as discussed by [17].

The contributions of this paper are: formally defining *Eventually Consistent Distributed Counters* (ECDCs), an alternative to classic distributed counters, towards achieving availability and partition tolerance; exposing the scalability issues that may arise when applying the pure CRDT approach and the availability issues when restricting CRDTs to server-side; introducing *Handoff Counters*, an implementation of the ECDC specification, providing eventually consistent counters that are simultaneously reliable, available and partition-tolerant, and scalable in the number of entities (both active and already terminated).

The main ingredients of our approach are: a replication mechanism that is non-symmetric (contrary to standard CRDTs where all replicas converge to the same state); hierarchical tiered topology, allowing different roles (e.g., many pure transient clients, a few permanent persistent servers) and availability through alternative communication paths; node identity containment (contrary to current CRDTs, node ids are not propagated to the whole network, being only temporarily stored in the state of some neighbor nodes); a mechanism to perform a reliable handoff of a value between two nodes, possibly using a third party, to achieve tolerance to partitions or transient node failures.

Standard approaches for fault tolerance for strong consistency (e.g., Paxos Commit [15]) incur the cost of communication with several nodes, even when no failures occur. Our approach, by aiming only for eventual consistency and by not addressing permanent node failures (leaving it as an orthogonal issue), does not require sending messages to multiple nodes, in the normal case, and has better availability and partition tolerance: in case of server crash or link failure, it is enough that a single alternative server is available and reachable, as opposed a majority of nodes.

The remainder of this paper is organized as follows: Sect. 2 briefly revisits classic strongly consistent distributed counters, and explains why they are not suitable for large-scale AP scenarios. In Sect. 3 we present the definition of ECDCs, stat-

ing both safety and liveness conditions. Section 4 describes a naive eventually consistent CRDT-based counter, that is AP in unreliable networks, and explains its scalability problems; it also discusses in more detail the problems that arise if replicas of this CRDT are restricted to server-side. In Sect. 5 we present *Handoff Counters*. Section 6 contains formal correctness proofs for the mechanism. In Sect. 7 we address some practical implementation issues. Then, in Sect. 8, we evaluate the scalability properties of *Handoff Counters* by running a discrete event simulation. In Sect. 9 we discuss how the handoff mechanism proposed can be applied to more general scenarios, beyond simple counters, to commutative monoids having an associative and commutative operation. In Sect. 10 we briefly discuss our approach to reliability and availability, comparing it to more standard approaches, and conclude in Sect. 11.

2 Classic distributed counters

In most papers about distributed counters, e.g., [2,28,32], a counter is an abstract data type that provides a fetch-and-increment operation (increment for short), which returns the counter value and increments it. The basic correctness criteria is usually: a counter in a quiescent state (when no operation is in progress) after n increments have been issued will have returned all values from 0 to $n-1$ with no value missing or returned twice. i.e., when reaching a quiescent state, all operations so far must have behaved as if they have occurred in some sequential order, what is known as *quiescent consistency* [2]. Some counter mechanisms [32] enforce the stronger *linearizability* [18] condition, which ensures that, whenever a first increment returns before a second one is issued, the first returns a lower value than the second.

Even not considering the stronger variants that enforce linearizability, classic distributed counters providing quiescent consistency are too strongly consistent if one is aiming for availability and partition tolerance. For classic distributed counters we have the following result for deterministic algorithms, with a trivial proof, which is nothing more than an instantiation of the CAP theorem:

Proposition 1 *A quiescently consistent counter with fetch-and-increment cannot be both available and partition tolerant.*

Proof Suppose a network with two nodes u and v, and a run A where they are partitioned. Assume an increment is issued at node u at time t_1 and no other operations are in progress. As the counter is available and partition tolerant, the increment will eventually return at some later time t_2. Because the system is in a quiescent state after t_2, this increment must have returned 0. Suppose an increment is then issued at node v at some later time $t_3 > t_2$. For the same reasons, this increment will eventually return, the system becomes quiescent again, and the returned value must, therefore, be 1. But as no messages got through between u and v, this run is indistinguishable by v from a run B in which u does not exist and only a single increment is issued by v. In run B, v will, therefore, behave the same as in run A and return the same value 1, which contradicts the requirement for run B that a single increment in the whole run should have returned 0 after reaching quiescence.

This proof formalizes the intuition that it is not possible to generate globally unique numbers forming a sequence in a distributed fashion without communication between the nodes involved.

3 Eventually consistent distributed counters

In this section we define a weaker variant of distributed counters, to overcome the too strongly consistent nature of classic distributed counters, and achieve availability and partition tolerance. We call it *eventually consistent distributed counters* (ECDCs).

Classic counters offer a combination of two different features: (1) keeping track of how many increments have been issued; (2) returning globally unique values. While powerful, this second feature is the problematic one when aiming for AP.

For many practical uses of counters (in fact what is being offered in NoSQL data stores like Cassandra and Riak) one can get away with not having the second feature, and having a counter as a data type that can be used to count events, by way of an *increment* operation, which does not return anything, and an independent *fetch* operation, which returns the value of the counter.

We are not aiming for obtaining globally unique sequence numbers, and this split in two operations, where fetch can return the same value several times, opens the possibility of returning stale values. Having two independent operations, one to mutate and the other to report, as opposed to a single atomic fetch-and-increment, corresponds also to the more mundane conception of a counter, and to what is required in a vast number of large scale practical uses (e.g., web page impressions), where many participants increment a counter while others (typically less, usually different) ask for reports.

It will be possible to obtain available and partition tolerant eventually consistent counters, by allowing fetch to return something other than the most up-to-date value. Nevertheless, we need concrete correctness criteria for ECDCs. We have devised three conditions. Informally:

- A fetch cannot return a value greater than the number of increments issued so far.

– At a given node, a fetch should return at least the sum of the value returned by the previous fetch (or 0 if no such fetch was issued) plus the number of increments issued by this node between these two fetches.
– At least all increments globally issued up to any given time will be reported eventually, at a later time, at every node (as soon as enough messages go through).

The first two criteria can be thought of as safety conditions. The first, not over-counting, is the more obvious one (and also occurs in classic distributed counters, as implied by their definition). The second is a local condition on session guarantees [30], analogous to having *read-your-writes* and *monotonic-read*, common criteria in eventual consistency [31]. The third is a liveness condition, which states that eventually, if network communication allows, all updates up to a given time are propagated and end up being reported everywhere. It implies that if increments stop being issued, eventually fetch will report the correct counter value, i.e., the number of increments. (In practice the third condition normally leads to implementations where, when the network is partitioned into multiple connected components, increments issued in each component are reported eventually in that same component.) We will now clarify the system model, and subsequently formalize the above correctness criteria for ECDCs.

3.1 System model

Consider a distributed system with nodes containing local memory, with no shared memory between them. Any node can send messages to any other node. The network is asynchronous, there being no global clock, no bound on the time it takes for a message to arrive, nor bounds on relative processing speeds. The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted). Some messages will, however, eventually get through: if a node sends infinitely many messages to another node, infinitely many of these will be delivered. In particular, this means that there can be arbitrarily long partitions, but these will eventually heal.

Nodes have access to stable storage; nodes can crash but eventually will recover with the content of the stable storage as at the time of the crash. Each node has access to its globally unique identifier.

As we never require that data type operations block waiting for other operations or for message reception, they are modeled as single atomic actions. (In I/O Automata [24] parlance, we will use a single action as opposed to a pair $\mathsf{opStart}$ (input action), and $\mathsf{opEnd}(r)$ (output action) "returning" r.)

This allows us to use op_i^t to mean that operation op was performed by node i at global time t, and in the case of fetch also for the result of that operation. The actions we use are fetch_i and incr_i for the data type operations, and $\mathsf{send}_{i,j}(m)$ and $\mathsf{receive}_{i,j}(m)$ for message exchange. Global time, under a total order, is only referred to in traces (sequences of actions) or correctness criteria but is not available to the nodes themselves.

3.2 Formal correctness criteria for ECDCs

An eventually consistent distributed counter is a distributed abstract data type where each node can perform operations fetch (returning an integer) and incr (short for increment), such that the following conditions hold (where $||$ denotes set cardinality and _ the unbound variable, matching any node identifier; also, for presentation purposes, we assume an implicit $\mathsf{fetch}_{\_}^0 = 0$ by all nodes). For any node i, and global times t, t_1, t_2, with $t_1 < t_2$:

Fetch bounded by increments

$$\mathsf{fetch}_i^t \leq \left|\{\mathsf{incr}_{\_}^{t'} | t' < t\}\right|,$$

Local monotonicity

$$\mathsf{fetch}_i^{t_2} - \mathsf{fetch}_i^{t_1} \geq \left|\{\mathsf{incr}_i^{t'} | t_1 < t' < t_2\}\right|,$$

Eventual accounting

$$\exists t' \geq t . \forall j . \mathsf{fetch}_j^{t'} \geq \left|\{\mathsf{incr}_{\_}^{t''} | t'' < t\}\right|.$$

These criteria, specific to counters, can be transposed to more general consistency criteria, namely they imply the analogous for counters of:

Eventual consistency From [31] *"[It] guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value."*. Eventual accounting is stronger than eventual consistency: it does not require increments to stop, but clearly leads to eventual consistency if increments do stop. All CRDTs include this consistency criteria [27].

Read-your-writes From [30] *"[It] ensures that the effects of any Writes made within a session are visible to Reads within that session."*. The analogous of this property, substituting increments for writes, is implied by local monotonicity: in a session where a process issues increments to a given node, at least the effect of those increments is seen by further fetches by that process.

Monotonic-reads Defined by [30] and using the formulation by [31] *"If a process has seen a particular value for the object, any subsequent accesses will never return any previous values"*. This property is also obtained by local monotonicity.

4 Naive CRDT-based counters

In state-based CRDTs [27] a replica can be locally queried or updated; it directly provides availability and partition tolerance, as all client-visible data type operations are performed locally, with no need for communication. Information is propagated asynchronously, by sending the local state to other nodes (e.g., using some form of Gossip [10]); the receiving node performs a *merge* between the local state and the received state. CRDTs are designed so that: the set of states that nodes can take conform to a join semilattice (a partially ordered set for which there is a defined least upper bound for any two elements [8]); the merge operation amounts to performing a mathematical *join* of the local and received states, computing their least upper bound; every data type operation is an *inflation* that moves the state to a larger value (i.e., $\forall x . f(x) \geq x$). This allows merges to be performed using arbitrary communication patterns: since join is associative, commutative and idempotent, duplicates are not a problem. In case of message loss, a message can be resent and possibly re-merged; old duplicates and messages received out-of-order are also not a problem. CRDTs overcome the problem of unreliable networks, with no need for some reliable communication mechanism involving global coordination.

The CRDT concept can be used to trivially obtain an ECDC. The local state will amount to a version-vector [25]: a map of node ids to non-negative integers. When a node wants to perform an increment, it increments the entry corresponding to its unique identifier (or adds an entry mapped to one if the id is not mapped). The fetch is obtained by adding all integers in the map. The merge operation also corresponds to reconciliation of version-vectors: maps are merged by performing a pointwise maximum (each key becomes mapped to the maximum of the corresponding values, assuming absent keys are implicitly mapped to 0).

Version-vector based counters respect the criteria for ECDCs: fetch is bounded by the number of increments, as it results from adding values corresponding to increments; local increments are immediately accounted, ensuring local monotonicity; increments by different nodes are accounted in disjoint entries in the map, which implies that, as each replica is propagated and merged to every other one, all increments performed by all nodes are eventually accounted.

4.1 The scalability problem of client-side CRDTs

Counters implemented as version-vectors, although meeting all criteria for ECDCs, suffer from serious scalability problems. Consider a network in which many nodes (clients) perform operations on the counter, while others (servers) allow information propagation and keep durable storage, allowing client nodes to cease from participating and retire. Suppose that new clients (each with a unique id) keep arriving over time, to use the counter.

In a pure CRDT approach each participating entity has a replica. In this case, each participating node (both clients and servers) will introduce its id in the map. Over time, as new clients arrive, and as replica state is propagated and merged with other replicas, the map will grow to unreasonable sizes, making both the storage space and communication costs (of transmitting the map) unbearable. The worst aspect is that the size does not depend only on the currently participating clients: it keeps growing, accumulating all ids from past clients that have already retired. This means that naive CRDTs such as version-vectors are not scalable when used client-side, being restricted to server-side in practice.

The mechanism we introduce in this paper can be seen as a non-naive CRDT, which: enforces *id containment*, preventing ids from spreading all over the network; allows graceful client retirement, removing ids that were temporarily stored in servers with which the client communicated, as we show in Sect. 8.4.

4.2 The availability problem of server-side CRDTs

Due to the above scalability problem, current version-vector based counters (e.g., in Cassandra or Riak) do not use the pure CRDT approach, but use CRDTs server-side only. This means that only a relatively small number of nodes (the servers) hold replicas, while clients use remote invocations to ask a server to perform the operation. Server-side CRDTs allow unreliable communication between servers, including partitions (e.g., between data-centers). However, the problem of unreliable communication between client and server remains.

As the increment operation is not idempotent, it cannot be simply reissued if there is doubt whether it was successful. In practice, this leads to the use of remote invocations over connection-oriented protocols (e.g., TCP [6]) that provide reliable communication. This only partially solves the problem: a sequence of acknowledged increments is known to have been applied exactly-once, but if the last increments are not acknowledged and the connection times out, these last increments are not known to have been successfully applied; these increments cannot be reissued using a new connection, to the same or a different server, as it could lead to overcounting (as they could have been applied after all, but only the reply was lost).

Attempts to circumvent this reliability problem by using a general data-type-agnostic communication layer bring back scalability and/or availability problems. If an infinite duration connection incarnation is maintained for each client, where no operation can fail due to a timeout, this will imply stable server state to manage each client connection, leading to state explosion in servers, as clients cannot be forgotten.

This is so because there is no protocol that gives reliable message transfer within an infinite connection incarnation for general unbounded capacity (e.g., wide-area networks) non-FIFO lossy networks that does not need stable storage between crashes [3,11]. This problem can be overcome by never failing due to a timeout, but allowing connections to be closed if there are no pending requests and the connection close handshake is performed successfully (e.g., before a partition occurs). With a three-way handshake an oblivious protocol is possible [4], with no need to retain connection specific information between connection incarnations, but only a single unbounded counter for the whole server.

With this last approach, the size of stable server state is not a problem in practice, but the reliability problem is overcome at the cost of availability: given a partition, a pending request will never fail due to a time-out, but the client that has issued a request to a given server will be forced to wait unboundedly for the result, without being able to give up waiting and continue the operation using a different server.

These problems can be summarized as: the use of a general purpose communication mechanism to send non-idempotent requests can provide reliability at the cost of availability. Our data-type specific mechanism overcomes this problem by allowing client-side CRDTs which are scalable.

5 Handoff Counters

In this section we present a novel CRDT based counter mechanism, which we call *Handoff Counters*, that meets the ECDC criteria, works in unreliable networks and is scalable. The mechanism allows arbitrary numbers of nodes to participate and adopts the pure CRDT approach, having a replica at each node without distinguishing clients and servers (therefore, overcoming the problems of server-side CRDTs), and allowing an operation (fetch or increment) to be issued at any node.

It addresses the scalability issues (namely the id explosion in maps) by: assigning a tier number (a non negative integer) to each node, in a hierarchical structure, where only a small number of nodes are classified as tier 0; having "permanent" version vector entries only in (and for) tier 0 nodes, therefore, with a small number of entries; having a *handoff* mechanism which allows a tier $n+1$ "client" to handoff values to some tier n "server" (or to any lower tier node, in general); making the entries corresponding to "client" ids be garbage-collected when the handoff is complete. Figure 1 illustrates a simple configuration, with end-client nodes connecting to tier 1 nodes in their regional datacenters.

Example 1 Even though no formal client/server distinction is made, a typical deployment scenario would be having, e.g., two (for redundancy) tier 0 nodes per data-center, devoted to inter-datacenter communication, a substantial, possibly variable, number of tier 1 server nodes per datacenter, and a very large number of tier 2 nodes in each datacenter. The datastore infrastructure would be made up of tier 0 and 1 nodes, while tier 2 nodes would be the end-clients (application server threads handling end-client connections or code running at the end-clients) connecting to tier 1 nodes in a datacenter. More tiers can be added in extreme cases, but this setup will be enough for most purposes: e.g., considering 5 datacenters, 50 tier 1 nodes per datacenter, each serving 1000 concurrent tier 2 clients, will allow 250,000 concurrent clients, with only 10 entries in the "permanent" version vectors.

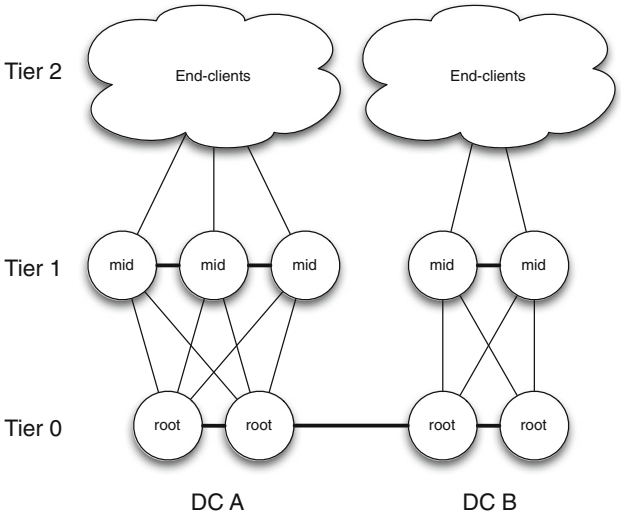

Fig. 1 A simple configuration with three tiers and two datacenters

The mechanism involves three aspects: classic version vector dissemination and merging between tier 0 nodes; the handoff, which migrates accounted values towards lower tiers, making all increments eventually be accounted in tier 0 nodes; "vertical" aggregation starting from tier 0, towards higher-tier nodes, to provide a best-effort monotonic estimate of the counter value.

Most of the complexity of the mechanism is related to achieving the handoff without violating correctness under any circumstances over unreliable networks, while allowing garbage-collection in typical runs. Other design aspects were considered, in addition to strictly achieving ECDC, namely:

- An end-client (maximum tier node) is typically transient; it should be able to know if locally issued increments have already been handed-off, so that it can stop the interaction and terminate;
- There should be no notion of session or affinity; a tier $n+1$ node u that started exchanging messages with a tier n node v, should be able to switch to another tier n node w at any time, e.g., if no message arrives and u suspects that v has crashed or there is a network partition between

u and v, but u is an end-client that wants to be sure its locally accounted increments have been handed-off to some server before terminating.

5.1 Network topology

Handoff Counters can be used with many different network topologies. The simplest one is to assume a fully connected graph, where any node can send messages to any other node. In general, handoff counters can work with less than full connectivity. The assumptions that we make about network topology for the remainder of the paper are:

- Each link is bidirectional.
- The network restricted to tier 0 nodes is a connected subnetwork.
- For each node u, there is a path from u to a tier 0 node along a strictly descending chain of tiers.
- If a node u is linked to two lower tier nodes v and w, then there is also a link between v and w.

These assumptions allow version vector propagation and merging in tier 0 nodes, while also allowing a client u to start by exchanging messages with a server v and later switching to a server w if v becomes unresponsive. These assumptions are met by Example 1 and by the Fig. 1 topology, where inter-datacenter communication is performed by tier 0 nodes, and where communication between tier 1 nodes or between tier $n+1$ and tier n nodes needs only be attempted within each datacenter. It should be emphasized that these assumptions reflect only what communications are attempted (not what links are available at any given time), and can be thought of as the rules for forming a communication overlay; any of these links may be down for some time, even possibly incurring temporary network partitions.

5.2 Distributed algorithm

A benefit of adopting the CRDT approach is not needing a complex distributed algorithm to achieve correctness—the complexity is transferred to the CRDT. Basically any form of gossip can be used, where each node keeps sending its replica state to randomly picked neighbors, and each node upon receiving a replica state merges it with the local one.

Without loss of generality we consider a single distributed counter replicated at every node. To describe both the mechanism and its correctness proofs we consider Algorithm 1 to be used, with operations defined in Figs. 4 and 5. In this distributed algorithm, each node i keeps a CRDT replica state C_i. Each replica is initialized using the globally unique node id and the node tier. Actions fetch_i and incr_i are delegated to the corresponding CRDT operations. Each node periodically picks a random neighbor j and sends it the local state. Upon receiving some state C_j through a link (j, i), it is merged with the local state, through the CRDT merge operation. The distributed algorithm is quite trivial, all effort being delegated to the Handoff Counter data type, namely through its init, fetch, incr and merge operations.

The state C_i is assumed to be stored in stable storage, and assigning to it is assumed to be an atomic operation. This means that temporary variables used in computing a data-type operation do not need to be in stable storage, and an operation can crash at any point before completing, in which case C_i will remain unchanged.

ALGORITHM 1: Distributed algorithm for generic node i.

constants:
i, globally unique node id
$tier_i$, tier of node i
$neigh_i$, set of neighbors of node i

state:
C_i, Handoff Counter replica state; initially, $C_i = \mathsf{init}(i, tier_i)$

on fetch_i
return $\mathsf{fetch}(C_i)$

on incr_i
$C_i := \mathsf{incr}(C_i)$

on $\mathsf{receive}_{j,i}(C_j)$
$C_i := \mathsf{merge}(C_i, C_j)$

periodically
let $j = \mathsf{random}(neigh_i)$
$\mathsf{send}_{i,j}(C_i)$

5.3 Handoff Counter data type

Unfortunately, the Handoff Counter data type is not so trivial. On one hand, a server which is receiving counter values from a client should be able to remove client-related entries in its state after the client stops participating; on the other hand, neither duplicate or old messages should lead to over-counting, nor lost messages lead to under-counting. Towards this, a Handoff Counter has state that allows a 4-way handshake in which some accounted value (a number of increments) is moved reliably from one node to the other (handed off).

The state of a Handoff Counter is a record with the fields shown in Fig. 2. Each replica keeps a map (vals), where the self-entry (the entry for key i in replica i) is only incremented or added-to locally, allowing local accounting of increments. In the case of tier 0 nodes there are also entries regarding other tier 0 nodes. In other words, the vals field is similar to a version vector, mapping node ids to integers, which has only the self-entry in the case of non tier 0 nodes, and also other tier 0 nodes entries in the case of tier 0 nodes.

id node id;
tier node tier;
val counter value that can be safely reported by fetch given local knowledge;
below lower bound of values accounted in lower tiers;
vals map from node ids to integers, storing increments;
sck source clock – logical clock incremented when creating tokens;
dck destination clock – logical clock incremented when creating slots;
slots map from source ids to pairs (sck, dck) of logical clocks;
tokens map from pairs (i, j) to pairs $((sck, dck), n)$ containing a pair of logical clocks and an integer;

Fig. 2 Handoff Counter data type state (record fields)

The state includes two components, one containing what we call *slots*—each slot serves as a capability of receiving a value—and the other contains *tokens*—a token holds a value and matches a single slot. The general idea is that, over time:

1. A slot is created in the state of a node j (destination of the handoff);
2. A token matching that slot is created in another node i (source of the handoff) to which some value (number of increments) accounted locally at vals is moved;
3. The slot at j is "filled": the slot is removed and the value in the corresponding token is acquired by j (added to the locally accounted value);
4. Node i removes the token.

The mechanism must ensure correctness no matter what communication patterns may occur. Several properties are ensured, namely:

- A given slot cannot be created more than once; even if it was created, later removed and later a duplicate message arrives;
- A token is created specifically for a given slot, and does not match any other slot;
- A given token cannot be created more than once; even if it was created, later removed and later a duplicate message having the corresponding slot arrives.

To achieve this, the CRDT state includes a pair of logical clocks, *source clock* and *destination clock*, that are used when a node plays the role of source of handoff and destination of handoff, respectively. If a node is always a leaf node, e.g. end-clients in the chosen topology, and never a destination of an handoff, it does not require a *destination clock*. Conversely, nodes that are destination only, e.g. tier 0 nodes, can drop the *source clock*. Each slot or token is identified by the quadruple: source id, destination id, source clock, destination clock. When creating a slot the destination clock is incremented; when creating a token for a given slot, the source clock of the source node is checked to be equal to the one in the slot and incremented. This ensures that neither a given slot nor a given token can be created more than once. Even though some analogy can be made to TCP sequence numbers, here not only each node keeps a pair of sequence numbers, but also each slot/token is identified by a pair of sequence numbers, one from each node (in addition to source and destination ids).

Even though slots and tokens are identified by quadruples, their entries in the state are maps (as opposed to, e.g., sets of slots) designed to obtain an efficient lookup in the case of slots. This is relevant as there may exist a considerable number of slots, depending on the number of concurrent clients, while there are typically very few tokens (just one in the more normal case). Such is possible due to the following:

- Each node j needs to keep at most one slot for any given node i. Therefore, a slot (i, j, sck, dck) is kept as an entry mapping i to pairs (sck, dck) in the slot map at j.
- For each pair of nodes i and j, there is only the need to keep at most one token that follows the form $((i, j, sck, dck), n)$. However, such token may be kept at nodes other than i. Tokens are stored in maps from pairs (i, j) to pairs $((sck, dck), n)$.

Figure 3 presents a run where a node i hands off some value (9 in this example) to a node j, when no messages are lost. The non-zero values in the source clock of node i and destination clock in node j indicate that the run is a continuation of a longer run that already did some handoffs from i to j. This example shows the evolution, after each merge, of the relevant fields in the state, illustrating the steps in the 4-way handshake:

1. Node i sends its C_i to node j; node j does $C'_j := \mathsf{merge}(C_j, C_i)$; the resulting C'_j has a slot created for node i, uniquely identified by the source and destinations clocks in i and j; the destination clock in j is incremented on slot creation;
2. Node j sends C'_j to i; node i then performs $C'_i := \mathsf{merge}(C_i, C'_j)$; the resulting C'_i has a token created specifically for that slot, into which the locally accounted number of increments has been moved; since a token was created the i source clock is incremented;
3. Node i sends C'_i to node j; node j does a $C''_j := \mathsf{merge}(C'_j, C'_i)$; this merge, seeing the token matching the slot, "fills the slot", i.e., acquires the accounted value in the token and removes the slot from the resulting C''_j;
4. Node j sends C''_j to i; node i then performs $C''_i := \mathsf{merge}(C'_i, C''_j)$; seeing that the slot is gone from C''_j, it removes the token from the resulting C''_i.

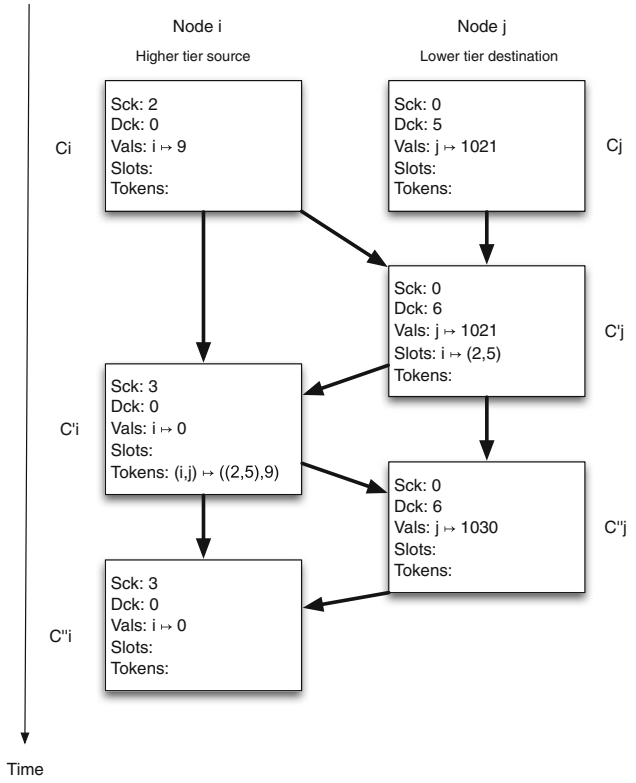

Fig. 3 A handoff from node i to j (only relevant fields are shown)

The end result of such an exchange is that an accounted value has been moved from node i to node j, and neither i has an entry related to j in its state C_i'', nor j has an entry related to i in C_j''. Temporary entries (slots and tokens) were removed, i.e., garbage collected.

It should be noted that, although a given handoff takes 4 steps, when a pair of nodes keep exchanging messages continuously, handoffs are pipelined: steps 1 and 3 are overlapped, as well as steps 2 and 4. When a message from a node i arrives at a lower tier node j, it typically carries a token. That token is acquired, filling the current slot at j, and a new slot is created; when the "reply" from j arrives at i, it makes i garbage collect the current token, and a new token is created. This means that in the normal no-loss scenario, each round-trip moves an accounted value (some number of increments) from i to j.

The reason for having two logical clocks per replica (source and destination clocks) has to do with liveness. For safety, having just one clock would suffice. But with a single clock there would be the danger that during the round-trip from "server" i to a lower tier node j, messages from clients arrive at i, increasing the clock at i when creating a slot, and making the handoff attempt from i to j fail (as a token would not be created at i because the clock at i would no longer match the source clock in the slot) and need to be repeated; under heavy load progress could be compromised. Having

$$
\begin{aligned}
\mathsf{init}(i, tier) &\doteq \{\mathsf{id} = i, \mathsf{tier} = tier, \mathsf{val} = 0, \mathsf{below} = 0, \mathsf{sck} = 0,\\
&\qquad \mathsf{dck} = 0, \mathsf{slots} = \{\}, \mathsf{tokens} = \{\}, \mathsf{vals} = \{i \mapsto 0\}\}\\
\mathsf{fetch}(C_i) &\doteq \mathsf{val}_i\\
\mathsf{incr}(C_i) &\doteq C_i\{\mathsf{val} = \mathsf{val}_i + 1, \mathsf{vals} = \mathsf{vals}_i\{i \mapsto \mathsf{vals}_i(i) + 1\}\}\\
\mathsf{merge}(C_i, C_j) &\doteq \mathsf{cachetokens}(\mathsf{createtoken}(\mathsf{discardtokens}(\mathsf{aggregate}(\\
&\qquad \mathsf{mergevectors}(\mathsf{createslot}(\mathsf{discardslot}(\mathsf{fillslots}(C_i, C_j),\\
&\qquad C_j), C_j), C_j), C_j), C_j), C_j), C_j)
\end{aligned}
$$

Fig. 4 Handoff Counter data type operations

two logical clocks per node allows a middle tier node to play both roles (source and destination), allowing the counter used for handoffs the node has started (as source) to remain unchanged so that the handoff may complete, even if the node is a busy server and is itself being the destination of handoffs.

To ensure the local monotonicity ECDC criteria, while allowing accounted increments to move between nodes, each Handoff Counter has two integer fields, val and below, always updated in a non-decreasing way. These fields have a role of supporting vertical aggregation, allowing higher tier nodes, the clients, to have an as accurate as possible estimate of the global count so far. Field val keeps the maximum counter value that can be safely reported according to local knowledge. Field below keeps a lower bound of values accounted in lower tiers (in the vals field). Also, by summarizing knowledge about tier 0 version vectors, the below field avoids the need for their dissemination to other tiers.

The Handoff Counter data type operations are shown in Fig. 4. The init operation creates a new CRDT replica for the counter; it takes as parameters the node id and tier number; it should be invoked only once for each globally unique id. Operation fetch simply returns the val field, which caches the highest counter value known so far. Operation incr increments both the self entry in the "version vector" (i.e., $\mathsf{vals}_i(i)$) and the cached value in val. For the purposes of conciseness and clarity, in a definition of an operation $\mathsf{op}(C_i) \doteq \ldots$, the fields of C_i can be accessed in the form field_i, e.g., tokens_i, and C_i denotes a record with field id containing i; i.e., $C_i \doteq \{\mathsf{id} = i, \mathsf{tier} = \mathsf{tier}_i, \ldots\}$.

The merge operation is by far the most complex one. It can be written as the composition of eight transformations that are successively applied, each taking the result of the previous transformation and the received state as parameters. Each of these transformations takes care of a different aspect of merging the two states into a new one; they are presented in Fig. 5, where the following notation is used.

Notation We use mostly standard notation for sets and maps/relations. A map is a set of (k, v) pairs (a relation), where each k is associated with a single v; to emphasize the functional relationship we also use $k \mapsto v$ for entries in a map. We use $M\{\ldots\}$ for map update; $M\{x \mapsto 3\}$ maps x to 3 and behaves like M otherwise. For records we use similar notations but with $=$ instead of $\mapsto$, to empha-

$$
\begin{aligned}
\mathsf{fillslots}(C_i, C_j) \doteq\ & C_i\{\mathsf{vals} = \mathsf{vals}_i\{i \mapsto \mathsf{vals}_i(i) + \textstyle\sum[n \mid (\_, n) \in S]\},\\
& \quad \mathsf{slots} = \mathsf{dom}(S) \lhd \mathsf{slots}_i\}\\
& \mathbf{where}\ S \doteq \{(src, n) \mid ((src, dst), (ck, n)) \in \mathsf{tokens}_j \mid\\
& \qquad\qquad dst = i \wedge (src, ck) \in \mathsf{slots}_i\}\\
\mathsf{discardslot}(C_i, C_j) \doteq\ & \mathbf{if}\ j \in \mathsf{dom}(\mathsf{slots}_i) \wedge \mathsf{sck}_j > \mathsf{fst}(\mathsf{slots}_i(j))\\
& \mathbf{then}\ C_i\{\mathsf{slots} = \{j\} \lhd \mathsf{slots}_i\}\\
& \mathbf{else}\ C_i\\
\mathsf{createslot}(C_i, C_j) \doteq\ & \mathbf{if}\ \mathsf{tier}_i < \mathsf{tier}_j \wedge \mathsf{vals}_j(j) > 0 \wedge j \notin \mathsf{dom}(\mathsf{slots}_i)\\
& \mathbf{then}\ C_i\{\mathsf{slots} = \mathsf{slots}_i\{j \mapsto (\mathsf{sck}_j, \mathsf{dck}_i)\}, \mathsf{dck} = \mathsf{dck}_i + 1\}\\
& \mathbf{else}\ C_i\\
\mathsf{mergevectors}(C_i, C_j) \doteq\ & \mathbf{if}\ \mathsf{tier}_i = \mathsf{tier}_j = 0\ \mathbf{then}\ C_i\{\mathsf{vals} = \cup^{\max}(\mathsf{vals}_i, \mathsf{vals}_j)\}\\
& \qquad\qquad \mathbf{else}\ C_i\\
\mathsf{aggregate}(C_i, C_j) \doteq\ & C_i\{\mathsf{below} = b, \mathsf{val} = v\}\\
& \mathbf{where}\ b \doteq \mathbf{if}\ \mathsf{tier}_i = \mathsf{tier}_j\ \mathbf{then}\ \max(\mathsf{below}_i, \mathsf{below}_j)\\
& \qquad\qquad \mathbf{else\ if}\ \mathsf{tier}_i > \mathsf{tier}_j\ \mathbf{then}\ \max(\mathsf{below}_i, \mathsf{val}_j)\\
& \qquad\qquad \mathbf{else}\ \mathsf{below}_i\\
& \qquad v \doteq \mathbf{if}\ \mathsf{tier}_i = 0\ \mathbf{then}\ \textstyle\sum[n \mid (\_, n) \in \mathsf{vals}_i]\\
& \qquad\qquad \mathbf{else\ if}\ \mathsf{tier}_i = \mathsf{tier}_j\ \mathbf{then}\ \max(\mathsf{val}_i, \mathsf{val}_j, b + \mathsf{vals}_i(i) + \mathsf{vals}_j(j))\\
& \qquad\qquad \mathbf{else}\ \max(\mathsf{val}_i, b + \mathsf{vals}_i(i))\\
\mathsf{discardtokens}(C_i, C_j) \doteq\ & C_i\{\mathsf{tokens} = \{(k, v) \in \mathsf{tokens}_i \mid \neg P(k, v)\}\\
& \mathbf{where}\ P((src, dst), ((\_, dck), \_)) \doteq (dst = j) \wedge\\
& \qquad \mathbf{if}\ src \in \mathsf{dom}(\mathsf{slots}_j)\ \mathbf{then}\ \mathsf{snd}(\mathsf{slots}_j(src)) > dck\\
& \qquad\qquad \mathbf{else}\ \mathsf{dck}_j > dck\\
\mathsf{createtoken}(C_i, C_j) \doteq\ & \mathbf{if}\ i \in \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{fst}(\mathsf{slots}_j(i)) = \mathsf{sck}_i\\
& \mathbf{then}\ C_i\{\mathsf{tokens} = \mathsf{tokens}_i\{(i, j) \mapsto (\mathsf{slots}_j(i), \mathsf{vals}_i(i))\},\\
& \qquad \mathsf{vals} = \mathsf{vals}_i\{i \mapsto 0\},\\
& \qquad \mathsf{sck} = \mathsf{sck}_i + 1\}\\
& \mathbf{else}\ C_i\\
\mathsf{cachetokens}(C_i, C_j) \doteq\ & \mathbf{if}\ \mathsf{tier}_i < \mathsf{tier}_j\\
& \mathbf{then}\ C_i\{\mathsf{tokens} = \cup^f(\mathsf{tokens}_i, t)\}\\
& \qquad \mathbf{where}\ t \doteq \{((src, dst), v) \in \mathsf{tokens}_j \mid src = j \wedge dst \neq i\},\\
& \qquad\qquad f((ck, n), (ck', n')) \doteq \mathbf{if}\ \mathsf{fst}(ck) \geq \mathsf{fst}(ck')\ \mathbf{then}\ (ck, n)\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (ck', n')\\
& \mathbf{else}\ C_i
\end{aligned}
$$

Fig. 5 Handoff Counter auxiliary transformations in merge

size a fixed set of keys. We use $\lhd$ for domain subtraction; $S \lhd M$ is the map obtained by removing from M all pairs (k, v) with $k \in S$. We use set comprehension of the forms $\{x \in S | P(x)\}$ or $\{f(x) | x \in S | P(x)\}$, or list comprehensions, using square brackets instead. The domain of a relation R is denoted by $\mathsf{dom}(R)$, while $\mathsf{fst}(T)$ and $\mathsf{snd}(T)$ denote the first and second component, respectively, of a tuple T. We use $\cup^f(m, m')$ to represent joining maps while applying f to the values corresponding to common keys, i.e., $\cup^f(m, m') = \{(k, v) \in m | k \notin \mathsf{dom}(m')\} \cup \{(k, v) \in m' | k \notin \mathsf{dom}(m)\} \cup \{(k, f(m(k), m'(k))) | k \in \mathsf{dom}(m) \cap \mathsf{dom}(m')\}$. To define a function or predicate by cases, we use **if** X **then** Y **else** Z to mean "Y if X is true, Z otherwise".

We now describe informally each of these transformations used in merge. We leave the formal proof of correctness for the next section.

$\mathsf{fillslots}(C_i, C_j)$ "Fills" all possible slots in C_i for which there are matching tokens in C_j, removing them from slots_i and adding the counter values in the tokens to the self version vector entry $\mathsf{vals}_i(i)$. We call this transfer of a counter value

in a token to the vals entry of the destination node *acquiring the token* and the corresponding slot removal *filling the slot*.
discardslot(C_i, C_j) Discards a slot, if any, in C_i for source j, that cannot ever be possibly filled by a matching token, because C_j ensures that no such token will ever be generated. For a slot that still remains in C_i (and, therefore, has not been filled by a matching token in C_j by the just-applied fillslots) this is the case if the source clock at C_j is greater than the corresponding value in the slot. Notice that there is at most one slot per source, in case a slot with an old source clock is discarded, at most one slot will be re-created (with a more up-to-date source clock) on the subsequent createslot() call. If a slot is kept, it must have been up-to-date and no new one is created in that subsequent call.
createslot(C_i, C_j) Creates a slot in C_i for a higher tier source node j, if C_j has some non-zero value to hand off and there is no slot for j at C_i. If a slot is created, the local destination clock is stored in it and increased, preventing duplicate creation of the same slot in case of duplicate messages. A slot merely opens the possibility of node j creating a corresponding token; not all slots will have corresponding tokens, some will be discarded or replaced by newer slots. In fact, the local knowledge in C_i after discarding information related to j makes it impossible to avoid subsequently creating a slot for j upon receiving a slow or duplicate message; this is not a problem as such slots will never match any token, being eventually discarded upon further communication.
mergevectors(C_i, C_j) Merges the corresponding version vector entries, doing a pointwise maximum for common ids. This is only done when merging two tier 0 replicas.
aggregate(C_i, C_j) Performs a vertical aggregation step (from lower to higher tiers up to the current one), that updates the below and val fields according to the knowledge provided by C_j. This can never decrease their current values. The effect is to propagate values accounted in tier 0 version vectors, while adding knowledge provided by intermediate nodes, up to C_i and C_j.
discardtokens(C_i, C_j) Discards from C_i the tokens that have already been acquired by C_j; this is the case for tokens with id $(src, j, \_, dck)$ if either there is a slot $(src, j, \_, dck')$ at C_j with $dck' > dck$ or if there is no slot $(src, j, \_, \_)$ at C_j and $\mathsf{dck}_j > dck$. Here src is normally, but not necessarily, equal to i, as C_i can cache tokens from a source other than i.
createtoken(C_i, C_j) Creates a token, to which the currently accounted value in $\mathsf{vals}_i(i)$ is moved, if there is a slot for i in C_j having a source clock equal to the current one at C_i. If a token is created, the local source clock is increased, preventing duplicate creation of the same token in case of duplicate messages.
cachetokens(C_i, C_j) Keeps a copy of tokens generated at a higher tier node j meant to some other destination k. For each pair source-destination, older tokens (that must have already been acquired) are replaced by newer ones. Caching tokens provides availability under faults, as it allows a client j in the middle of a handoff to k, to delegate to i the responsibility of finishing the handoff, in case j wants to terminate but either k has crashed and is recovering or the link between j and k is currently down. Only tokens that have been generated at node j are cached (other tokens currently cached at j are not considered) so that alternate handoff routes are provided, while preventing the flooding and the large sets of tokens that would result from a transitive dissemination of tokens to other nodes.

5.4 Implementation and experimental validation

We designed Handoff Counters in tandem with a prototype implementation and a testing infrastructure that allows for randomized runs over given topologies. We exercised the solution robustness, in particular by replaying old messages out of order, which allowed the detection of errors in several corner cases, enabling the correction of subtle bugs that existed in tentative versions of the mechanism.

Although testing does not ensure correctness, the implementation has successfully passed randomized traces with one hundred million steps, both in giving the correct result and also in garbage collecting temporary entries, making it a complement to a manual formal proof.

An experimentally robust solution was thus a prelude to the formal correctness proof in the following section, and added an independent assessment to the overall approach. This implementation and testing infrastructure, written in Clojure, is publicly available in GitHub (https://github.com/pssalmeida/clj-crdt), and can be used for interactively experimenting with counters in the Clojure REPL.

We also performed scalability evaluation of Handoff Counters in very large scenarios, which we present in Sect. 8. For that, given the excessive memory demands of Clojure, we wrote a Rust implementation, available in https://github.com/pssalmeida/handoff_counter-rs.

6 Correctness

In this section we establish the needed properties to verify that Handoff Counters meet the criteria for Eventually Consistent Distributed Counting.

Lemma 1 *Any slot (s, d, sck, dck) can be created at most once.*

Proof Each node uses its own id as d in the slot; therefore, no two nodes can create the same slot. In each node, a slot is only created when applying createslot, which also increments dck upon storing it in the slot; therefore, a subsequent createslot in the same node cannot create the same slot.

Lemma 2 *Any token with id* (s, d, sck, dck) *can be created at most once.*

Proof Identical to Lemma 1.

Lemma 3 *Any token with id* (s, d, sck, dck) *can be acquired at most once.*

Proof Such token can only be acquired in node d having a corresponding slot (s, d, sck, dck) while performing the fillslots function, which removes this slot from the resulting state, preventing a subsequent acquisition of the same token, as due to Lemma 1 this slot cannot be recreated. □

Proposition 2 *Given a token* T *with id* (s, d, sck, dck)*: (i)* T *will not be removed from any node before it has been acquired; (ii) a corresponding slot* S *will exist in node* d *between the time when* T *is created and the time when* T *is acquired.*

Proof By induction on the length of the trace of the actions performed by the system, using (i) and (ii) together in the induction hypothesis. The only relevant action is $\mathsf{receive}_{j,i}(C_j)$ of a message previously sent by a node j to a node i, with the corresponding merge being applied to the state of i. Given the asynchronous system model allowing message duplicates, we must assume that C_j can be the state of j at any point in the past, due to some $\mathsf{send}_{j,i}$ action that resulted in a message arbitrarily delayed and/or duplicated.

Regarding (i), a token T with id (s, d, sck, dck) can only be removed either: (1) In discardtokens when merging with a state C from d, with either a slot $(s, d, \_, dck')$ having $dck' > dck$, or with no slot for s and the destination clock in C greater than dck; either way, given that slots for destination d are created with increasing destination clock values, it implies that C corresponds to a time after slot S was created. By the induction hypothesis, S would have existed until T was acquired, and as S is absent from C, this implies that T was already acquired. (2) Or T could be removed by the tokens map entry for (s, d) being overwritten in createtoken; this last case is not possible because: tokens are created using the current source clock, which is then incremented; for a token with id $(s, d, \mathsf{sck}_s, \_)$ to be created, a received counter state C from d must contain a slot $(s, d, sck', \_)$ with $sck' = \mathsf{sck}_s$. This means that d would have previously received T from s and from the induction hypothesis, d would have had a corresponding slot and would have already acquired T filling the slot. When C arrived at s, T would be discarded by discardtokens before invoking createtoken, i.e., as in the first case above.

Regarding (ii), T is created, in createtoken, only if a corresponding slot has been created at some previous time in node d; this slot can only be removed either: in fillslots, when T is acquired; or in discardslot, when merging with a counter C from node s whose source clock is greater than sck, implying a state in s after T has been created. But in this case this slot at node d cannot reach the discardslot function as, by the induction hypothesis, T would be present in C and would have been acquired by fillslots, filling the slot, just before invoking discardslot.

Lemma 4 *Any token with id* (s, d, sck, dck) *will be eventually acquired.*

Proof Such token, created at node s, must have resulted from direct message exchanges between s and a lower tier node d. From Proposition 2, this token will remain at s and a corresponding slot will exist at d until the token is acquired. As s will keep sending its C_s to its neighbors, therefore to d, and from the system model assumptions (Sect. 3.1) messages eventually get through, if the token has not yet been acquired (e.g., by communication between d and some other node caching the token), C_s containing the token will eventually arrive at d, which will acquire it.

Definition 1 (*Configuration*) A configuration C is the set of replicas existing at a given time in all nodes, i.e., the set composed of each state C_i, for each node i.

When mentioning time explicitly, we denote the configuration at time t by C^t, and the state of replica i at time t by C_i^t.

Definition 2 (*Enabled token*) A given token with $((s, d, sck, dck), n)$ is called *enabled* if there exists a corresponding slot (s, d, sck, dck) at node d. The set of enabled tokens in a configuration C is denoted E_C.

Whether a token is enabled is a global property, not decidable given local state by nodes holding the token. From Proposition 2, a token is enabled when created; it remains enabled until it is acquired, when the corresponding slot is filled. As E_C is defined as the *set* (not multiset) of enabled tokens, the presence of duplicates of some token, created at one node and cached at some other node(s), is irrelevant.

Lemma 5 *Fields* below *and* val *are non-decreasing.*

Proof By induction on the length of the trace of the actions performed by the system. These fields only change by an incr_i at node i, which increments val_i, or when doing a merge when receiving a message, in aggregate, which either updates below and val using a maximum involving the respective current value, or stores in val_i the sum of vals_i entries, if i is a tier 0 node, which is also non-decreasing, as vals_i for tier 0 nodes contain a set of entries always from tier 0 nodes, only updated by a pointwise maximum (as tier 0 nodes never create tokens).

Definition 3 (*Cumulative Tier Value*) In a configuration C, the cumulative tier value for tier k, written $\mathsf{CTV}_C(k)$, is the

sum, for all nodes with tier up to k, of the self component of the vals field plus the tokens created by these nodes that are still enabled, i.e.:

$$\mathsf{CTV}_C(k) \doteq \sum[\mathsf{vals}_i(i) | C_i \in C | \mathsf{tier}_i \leq k] + \sum[n | ((i, \_, \_, \_), n) \in E_C | \mathsf{tier}_i \leq k].$$

Lemma 6 *For each k, $\mathsf{CTV}_C(k)$ is non-decreasing, i.e., for any transition between configurations C and C', $\mathsf{CTV}_C(k) \leq \mathsf{CTV}_{C'}(k)$.*

Proof For any node i, the only time $\mathsf{vals}_i(i)$ can decrease is when a token is created, enabled, and the value is moved to the token; in this case $\mathsf{CTV}_{C'}(k)$ remains unchanged for all k. When a token holding value n ceases to be enabled (being acquired), n is added to the $\mathsf{vals}_j(j)$ field for some lower tier node j; this makes $\mathsf{CTV}_{C'}(k)$ either unchanged or greater.

Lemma 7 *$\mathsf{CTV}_C(k)$ is monotonic over k, i.e., $k_1 \leq k_2 \Rightarrow \mathsf{CTV}_C(k_1) \leq \mathsf{CTV}_C(k_2)$.*

Proof Trivial from the CTV definition.

Proposition 3 *For any counter replica C_i in a configuration C: (i) $\mathsf{below}_i \leq \mathsf{CTV}_C(\mathsf{tier}_i - 1)$; (ii) $\mathsf{val}_i \leq \mathsf{CTV}_C(\mathsf{tier}_i)$.*

Proof By induction on the length of the trace of the actions performed by the system, using (i) and (ii) together in the induction hypothesis. Given that $\mathsf{CTV}_C(k)$ is non-decreasing (by Lemma 6), so are the right-hand sides of the inequalities, and the only relevant actions are those that update either below_i or val_i: (1) An increment incr_i at node i, resulting in an increment of both val_i and $\mathsf{vals}_i(i)$, in which case the inequality remains true. (2) A $\mathsf{receive}_{j,i}(M)$ of a message previously sent by a node j to node i, with the corresponding merge being applied to the state of i, and the fields being updated by aggregate. Regarding below_i, there are three cases: it remains unchanged, it can be possibly set to below_j if $\mathsf{tier}_j = \mathsf{tier}_i$, or it can be possibly set to val_j if $\mathsf{tier}_j < \mathsf{tier}_i$; in each case the induction hypothesis is preserved because CTV_C is non-decreasing (M can be any message from node j, arbitrarily from the past) and in the last case also due to the monotonicity of $\mathsf{CTV}_C(k)$ over k (by Lemma 7). Regarding val_i, either it is set to the sum of the vals_i values, if $\mathsf{tier}_i = 0$, which does not exceed $\mathsf{CTV}_C(0)$ due to the pointwise maximum updating of vals fields for tier 0 nodes; or it either remains unchanged, is set to val_j only if $\mathsf{tier}_i = \mathsf{tier}_j$, or is set to the sum of the values (computed for the next configuration) of below_i with $\mathsf{vals}_i(i)$ and also $\mathsf{vals}_j(j)$ when $\mathsf{tier}_i = \mathsf{tier}_j$; in each case the induction hypothesis is preserved.

Proposition 4 *The number of increments globally issued up to any time t, say I^t, is equal to the sum of the values held in the set of enabled tokens and of the self entries in the* vals *field of all nodes; i.e., for a network having maximum tier T, given a configuration C^t at time t, we have $I^t = \mathsf{CTV}_{C^t}(T)$.*

Proof By induction on the length of the trace of the actions performed by the system. The relevant actions are an increment at some node i, which results in an increment of the i component of the vals field of node i; or a $\mathsf{receive}_{j,i}(C_j)$ of a message previously sent by a node j to a node i, with the corresponding merge being applied to the state of i, leading possibly to: the filling of one or more slots, each slot S corresponding to a token (S, n), which adds n to $\mathsf{vals}_i(i)$ and removes slot S from i, which makes the token no longer enabled, leaving the sum unchanged; discarding a slot, which cannot, however, correspond to an enabled token, as from Proposition 2 a slot will exist until the corresponding token is acquired; merging vals pointwise for two tier 0 nodes, which does not change the self component $vals_i(i)$ of any node i; discarding tokens, which cannot be enabled because, by Proposition 2, tokens are only removed from any node after being acquired; the creation of an enabled token $(\_, n)$, at node i, holding the value $n = \mathsf{vals}_i(i)$ and resetting $\mathsf{vals}_i(i)$ to 0, leaving the sum unchanged; caching an existing token, which does not change the set of tokens in the system.

Proposition 5 *Any execution of Handoff Counters ensures ECDC fetch bounded by increments.*

Proof In any configuration C, a fetch_i at replica C_i simply returns val_i. From Proposition 3, this value does not exceed $\mathsf{CTV}_C(\mathsf{tier}_i)$, which, from the monotonicity of $\mathsf{CTV}_C(k)$ (Lemma 7) and Proposition 4, does not exceed the number of globally issued increments.

Proposition 6 *Any execution of Handoff Counters ensures ECDC local monotonicity.*

Proof Operation fetch_i simply returns val_i, which is non-decreasing (Lemma 5) and which is always incremented upon a local $incr_i$; therefore, for any node i the difference between fetch_i at two points in time will be at least the number of increments issued at node i in that time interval.

Proposition 7 *Any execution of Handoff Counters ensures ECDC eventual accounting.*

Proof Let T be the maximum node tier in the network, and N the set of nodes. From Proposition 4, the number of increments I^t globally issued up to any time t, for a configuration C^t, is equal to $\mathsf{CTV}_{C^t}(T)$. From Lemma 4, by some later time $t' > t$, all tokens from tier T enabled at time t will have been acquired by lower tier nodes (if $T > 0$), and also because CTV is non-decreasing, it follows that $I^t \leq \mathsf{CTV}_{C^{t'}}(T')$, for some $T' < T$. Repeating this reasoning along a finite chain $T > T' > \cdots > 0$, by some later time t'' we have

$I^t \leq \mathsf{CTV}_{C^{t''}}(0) = \sum[\mathsf{vals}_i^{t''}(i) | C_i^{t''} \in C^{t''} | \mathsf{tier}_i = 0]$, this last equality holds because there are no tokens created at tier 0; given the network topology assumptions of tier 0 connectedness, eventually at some later time t''', all vals entries in all tier 0 nodes will be pointwise greater than the corresponding self entry at time t'', i.e., $\mathsf{vals}_i^{t'''}(j) \geq \mathsf{vals}_j^{t''}(j)$ for all tier 0 nodes i and j. Given the topology assumptions of the existence, for each node i, of a path along a strictly descending chain of tiers $\mathsf{tier}_i > \cdots > 0$, eventually, by the aggregate that is performed when merging a received counter, repeated along the reverse of this path, at some later time the val_i field for each node i in the network will have a value not less than the sum above, and therefore, not less than the number of increments I^t globally issued up to time t, which will be returned in the fetch_i operation.

Theorem 1 *Handoff Counters implement eventually consistent distributed counters.*

Proof Combine Propositions 5, 6, and 7. □

7 Practical considerations and enhancements

Handoff Counters were presented as a pure CRDT, that works under a simple gossip algorithm "send counter to all neighbors and merge received counters". Here we discuss practical issues and outline some enhancements such as more selective communication, how to amortize the cost of durable writes while ensuring correctness, how to avoid sending the full CRDT state and the issue of client retirement. A formal treatment of these issues is deferred to further work.

7.1 Topology and message exchanges

We have described a mechanism which is arbitrarily scalable, through the use of any suitable number of tiers. In Example 1 we have described a three tier scenario: Tier 0 for permanent nodes, Tier 1 for serving nodes, appropriate to the number of end-clients, and Tier 2 for end-clients.

In practice, the most common deployment will probably consist of only Tier 0 (for the data-store) and Tier 1 (for the end-clients). This is because a large scale scenario typically involves not only many clients, but also many counters, with requests spread over those many counters. Having a couple of Tier 0 nodes per data-center per counter will cover the most common usage.

For presentation purposes, the distributed algorithm consisted simply of a general gossip, where each node keeps sending its counter replica to each neighbor. In practice, in the role of client, a node will simply choose one lower tier neighbor as server, to use in the message exchange, to maximize the effectiveness of the handoff. Not only does this avoid extra token caching by other nodes, and subsequent work in removing them after they have been acquired, but it also avoids the creation of slots that will have no chance of being filled and will have to be discarded. It is only when a client suspects that the chosen server is down, or there is a network partition, that the client should choose another node as server.

7.2 Fault tolerance

The mechanism was designed to tolerate network failures and transient node failures, but assumes that the state resulting from an operation is successfully stored in durable storage. We leave it as orthogonal to our mechanism the way each node achieves tolerance against permanent failure (e.g., through the use of storage redundancy).

To improve performance, in nodes that have the server role and receive handoffs from clients but do not receive local incr requests, the actual write to durable storage should not be made after each merge operation. But if the write to durable storage is delayed and messages continue to be exchanged, a node crash will violate the correctness assumptions, as the local in-memory state, which other nodes could already have received and merged, will be lost.

To overcome this problem, a maximum frequency of durable writes can be defined. Between durable writes, all state received from other nodes is merged to the transient in-memory state, but no messages are sent back; instead, sender node ids are collected in a transient set. After a durable write, messages containing the written state are sent to those nodes in the collected set of ids.

This means that all messages sent correspond to a durably stored state; if the node crashes the transient state is lost, but this is equivalent to losing the messages received since the last durable write. As the mechanism supports arbitrary message loss, correctness will not be compromised. This solution amortizes the cost of a durable write over many received requests, essential for a heavily loaded server. Under light load, durable writes can be made and a reply message sent immediately.

The maximum frequency of writes can be tuned according to both storage device characteristics and network latency. As an example, if clients of a given node are spread geographically and there is considerable latency (e.g., 50 ms), waiting some time (e.g., 5 ms) to collect and merge messages from several clients before writing to durable storage and replying should not have a noticeable impact.

At high-tier nodes, that only have a client role, and where incr requests are handled, the persistent logging requirement might also be a concern. Depending on the rate of incr operations issued, it might not be feasible (without assuming incoming new technology, such as *non-volatile memory*) to persist after each incr operation issued. The consequence of

not persisting immediately, is to have a vulnerability window of accepted and non persisted increments. However, if we consider the client tier to be located next to the user (e.g. in a smart-phone) and consequently only handling requests from that user, the rate of incr is probably low enough to support immediate persistence.

7.3 Restricting transmitted state through views

The algorithm as described adopts the standard state-based CRDT philosophy, in which the full replica state is sent in a message to be merged at the destination node. We can optimize by sending only the state that is relevant for the destination node. This strategy assumes that messages are sent to specific nodes (as opposed to, e.g., being broadcasted) and that the sender knows the node id and tier of the message destination. It also assumes that server nodes that a given client uses for handoff are all of the same tier. This assumption is reasonable, and met by the examples discussed, where clients of tier $n+1$ hand off to nodes of tier n.

The insight is that when a node i is sending C_i to a higher tier node j, in what regards the slots field, only the entry for node j is relevant when the merge is applied at j; all the other entries are ignored when $\mathsf{merge}(C_j, C_i)$ is performed at j, and can be omitted from the state to be sent. When i is sending to a lower tier node j, no slots need to be sent, because no slot from C_i is relevant for the merge at j. It is only when communicating with a node j of the same tier that the full slots_i map must be sent, as j may be caching tokens from some higher tier client, destined for i.

Using the insight above, instead of doing a $\mathsf{send}_{i,j}(C_i)$, node i can make use of a function view to restrict the state to the information relevant to node j, and do a $\mathsf{send}_{i,j}(\mathsf{view}(C_i, j))$. This function can be defined as:

$$\begin{aligned}\mathsf{view}(C_i, j) \doteq\ & \textbf{if } \mathsf{tier}_i < \mathsf{tier}_j \textbf{ then} \\ & \quad C_i\{\mathsf{slots} = \{(k,s) \in \mathsf{slots}_i | k = j\}\} \\ & \textbf{else if } \mathsf{tier}_i > \mathsf{tier}_j \textbf{ then } C_i\{\mathsf{slots} = \{\}\} \\ & \textbf{else } C_i\,.\end{aligned}$$

Even though this only involves the slots field, this component will constitute the largest part of the counter state in a busy server with many concurrent clients, as it can have one slot per client. This optimization will allow sending only a small message to each client, and also avoid sending slots to lower tier nodes (e.g., when a tier 1 node communicates with tier 0).

7.4 Client retirement

Given that each node accounts locally issued increments until they are handed off, when an end-client has stopped issuing increments and wants to retire, it should continue exchanging messages until it is certain that those increments will be accounted elsewhere (in lower tier nodes).

The normal way of doing so is to keep exchanging messages with the chosen server, until the vals self component is zero and the tokens map is empty. This can, however, mean waiting until a partition heals, if there is a token for a partitioned server. The token caching mechanism allows the client to start a message exchange with an alternate server, which will cache the token, to be delivered later to the destination.

While an end-client i wishes to remain active, even if some node k has already cached a token from i to server j, client i cannot discard the token unless it communicates with j after j has acquired it; otherwise, it could cause an incorrect slot discarding at j. But in the retirement scenario, if $\mathsf{vals}_i(i)$ self component is zero, and i has learned that all its tokens are already cached at other nodes (by having seen them in messages received from those nodes), i can stop sending messages and retire definitely. As no more messages are sent, no incorrect slot removal will occur, and as all tokens from i are cached elsewhere they will be eventually acquired, implying a correct eventual accounting of all increments issued at i.

Another issue regarding client retirement is slot garbage collection. The mechanism was designed to always ensure correctness, and to allow temporary entries (slots and tokens) to be removed in typical runs. As such, slots must be kept until there is no possibility of them being filled. The mechanism was designed so that a server can remain partitioned an arbitrary amount of time after which a message arrives containing a token. This raises the possibility that: a client C sends a message to a server S_1, a slot is created at S_1, a partition occurs just before a corresponding token is created at C, the client starts exchanging messages with another server S_2 and successfully hands off the local value to S_2 and retires; in this scenario, the slot at S_1 will never be garbage collected, as C is no longer alive to communicate with S_1. (Under our system model C is not expected to retire for ever, and all partitions eventually heal, but dealing with client retirement is a relevant practical extension.)

In this example, even though correctness was not compromised, each such occurrence will lead to irreversible state increase which, even if incomparable in magnitude to the scenario of naive CRDTs with client ids always polluting the state, is nevertheless undesirable. This motivates a complementary mechanism to improve slot garbage collection: if a client starts using more than one server, it keeps the set of server ids used; when it wishes to retire the intention is communicated to the server, together with the set of server ids, until the retirement is complete; a server which receives such intention keeps a copy of the last token by that client (in a separate data-structure, independently of whether the server caches or acquires the token), and starts an algorithm which disseminates the token to the set of servers used by

the client and removes it after all have acknowledged the receipt. The insight is that when one of these servers sees the token, it can remove any slot for that client with an older source clock. For this, it is essential that this information is piggy-backed in the normal messages between servers, and processed after the normal merge, so that a server that has a slot corresponding to an enabled token for that client, which may be cached in another server will see the token and fill the corresponding slot, before attempting slot garbage collection by this complementary mechanism.

8 Scalability evaluation

We now evaluate how scalable is the mechanism in practice. To efficiently evaluate scenarios with up to one hundred thousand clients we reimplemented Handoff Counters and built a discrete event simulator for asynchronous networks using the Rust programming language. The simulation infrastructure, including the scripts used to obtain the results for each scenario below (as well as the runs performed) can be obtained from https://github.com/pssalmeida/handoff_counter_simulator-rs.

Even though the actual latency distribution is not important, as long as "messages are in transit for some time", as in [20], inspired by [19], we model message latency by the sum of a fixed component and a delay given by a Weibull distribution; in our case we used 25 ms plus a Weibull of shape 2 and scale 25 ms, except for communication between two tier 0 nodes, where we use twice the latency, i.e., $50 + \text{Weibull}(2, 50)$, as they tend to be more geographically dispersed.

For comparison with a naive counter CRDT, there was no need to actually simulate a counter vector, and we just store and propagate sets of identifiers. For a compact set of integers representation we used *Roaring Bitmaps* [7] (through the `roaring` Rust *crate*). Unsurprisingly, each identifier reaches every node very fast (in under one second) and at any time the set of ids at each node roughly corresponds to the set of all nodes.

We evaluated several scenarios. In all of them we used a three tier network, with a fixed small number (10) of tier 0 nodes, representing the permanent infrastructure; we used a larger number of tier 1 nodes (either 100 or 1000 nodes), what we now call *servers*; and we use a varying number of tier 2 nodes, what we now call *clients*, up to one hundred thousand. The reason for a three tier network is to allow more scalability and adaptability, allowing the choice of tier 1 nodes according to needs; tier 0 nodes cannot be retired with no impact, and should always be a small number.

We essentially ignored tier 0 nodes, and concentrated on evaluating the impact on the servers (tier 1 nodes) of having many possible clients (tier 2 nodes) under different scenarios. The essential measure of scalability for Handoff Counters that we used was the average number of slots per tier 1 node, as a good measure of space complexity. Slots are the state component that can vary a lot, while the other components do not in typical interactions (i.e. ignoring naive gossip patterns that would not be used in practice).

For the scalability evaluation, and towards a positive outcome, it was important to deploy some of the "practical considerations" from the previous section; otherwise, e.g., using a naive gossip from a client to many servers, larger counter states could be created, e.g. with many tokens, or many slots that would persist for a long time without being discarded. In all scenarios:

- When a client arrives, it chooses a server randomly and keeps communicating with this server, unless otherwise stated;
- We use the view function from the previous section to send state, to keep messages in transit small.

We evaluated four scenarios:

1. Runs with clients arriving continuously at a given rate, while keeping actively doing increments and exchanging messages;
2. Runs with clients arriving continuously at a given rate, but where each client only performs increments in sessions occupying a given percentage of total time, while keeping affinity with the chosen server;
3. Runs where clients disconnect abruptly and choose a different random server when reconnecting, exchanging messages only with this new server;
4. Runs with client retirement, where clients both arrive and retire at a given rate, while keeping a given number of active clients.

The client disconnect scenario provided useful insight, exposing a scalability problem in which many slots persisted. It was possible to overcome it by "smarter" message exchange logic without modifying the Handoff Counter mechanism itself, but only how it was used.

8.1 Varying number of clients

The first scalability test involved varying the number of client nodes, while fixing the number of servers. Due to the little variability of outcomes, we present single runs where clients continuously arrive and keep active, for two scenarios, with 100 and 1000 servers. Figure 6 presents two such runs, each starting with 0 clients and a client arriving each 10 ms, up to 100,000 clients. The plots depict the number of clients, the average for all servers of the size of the set of ids at each server (representing the size of the vector in naive CRDT

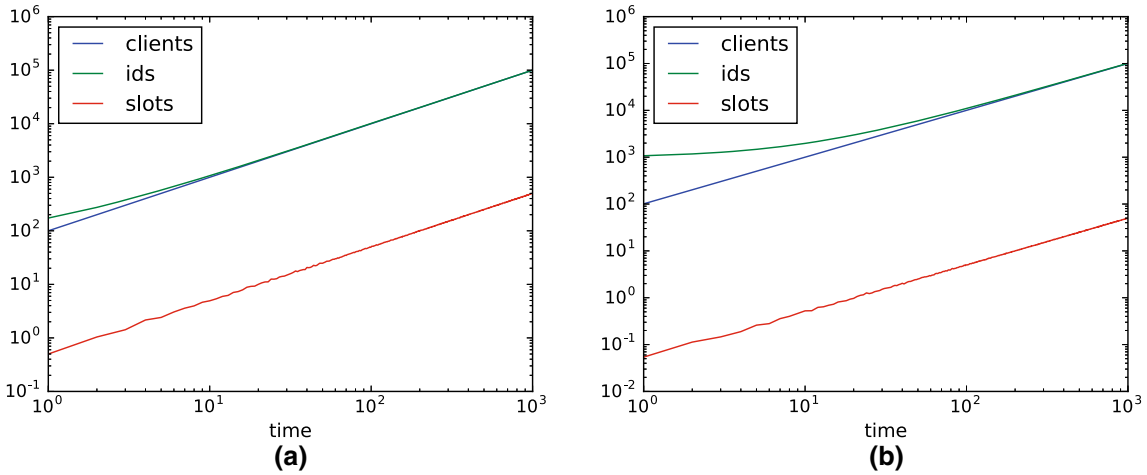

Fig. 6 Number of clients and average number of ids and slots per server. Comparing tier 1 configurations with 100 and 1000 servers, both for up to 100,000 clients

counters) and the average number of slots for all servers, along simulated time.

These runs show that the ids set size is slightly above the current number of clients (because although there is propagation delay, it also includes tier 0 and tier 1 ids, not only clients). It (unsurprisingly) confirms that classic counter CRDTs do not allow scaling to cope with more clients: if a given server node has some space capacity, it does not help to increase the number of servers if there are more clients, as the space-per-server cannot be kept bounded, and keeps increasing linearly with clients. On the other hand, Handoff Counters allow the server infrastructure to be scaled to cope with more clients: if a given number of slots, say, 100, is the maximum desirable for a given server, 100 servers can be used if there are 10,000 clients, and if the number of clients increases to 100,000, the number of servers can be also scaled to 1000, allowing to keep the space-per-server bounded.

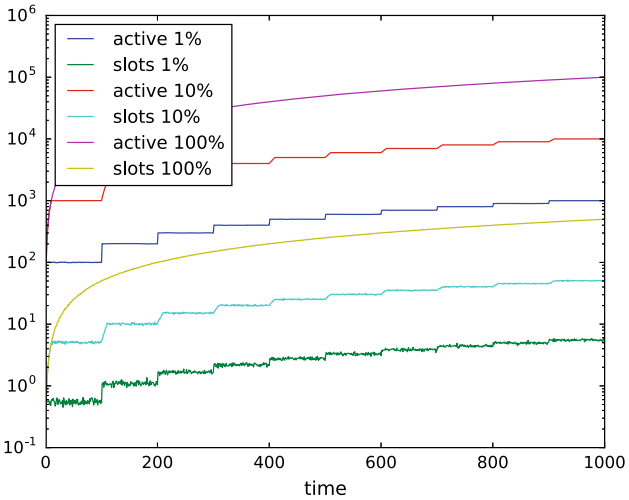

Fig. 7 Number of active clients and slots per server, with clients sessions taking 1, 10, and 100% of time, with server affinity

8.2 Client activity sessions

In the second experiment clients also arrive continuously, one each 10 ms, but each client is only active (performs increments) in sessions ocupying a given percentage of time (1, 10, 100%), while keeping affinity with the choosen server. Each client undergoes a number of sessions, each lasting the given percentage of a period of 100 s, being inactive the remaining time. (So, each session lasts either 1, 10 or the full 100 s.)

Figure 7 shows the number of active clients and the average number of slots per server along time, for each activity percentage, in a scenario with 100 servers. It can be seen that the number of slots is roughly proportional to the number of active clients, not the total number of clients. This shows another scalability benefit of Handoff Counters: it allows a smaller number of servers to be used to handle a given number of clients, under the common case where clients are active only a percentage of time.

8.3 Client reconnection

We experiment a more stressful scenario, where clients disconnect and reconnect, but without keeping affinity to the previous server. Now, each time a client reconnects it chooses a random server to exchange messages with. Also, contrary to the previous scenario, now clients disconnect abruptly, as opposed to merely stopping issuing increments but allowing some further communication; this will cause slots being left at a server when a disconnect happens.

The first experiment shows the disaster that would occur if a naive client were used: a client which simply exchanges messages with the new chosen server, ignoring the old ones.

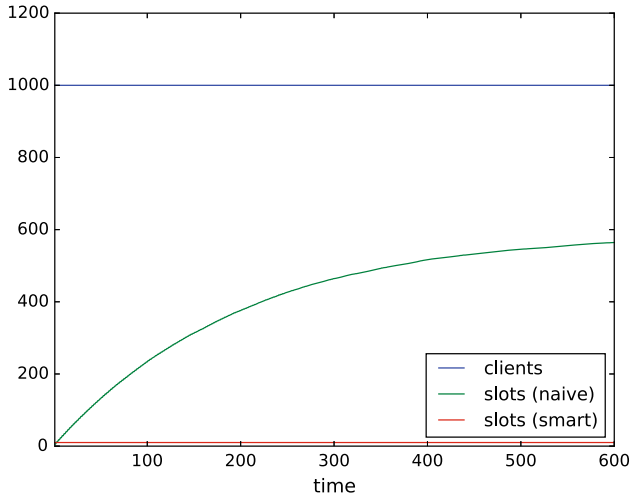

Fig. 8 Slots per server for 100 servers and 1000 clients abruptly disconnecting and reconnecting to random server, ignoring old servers

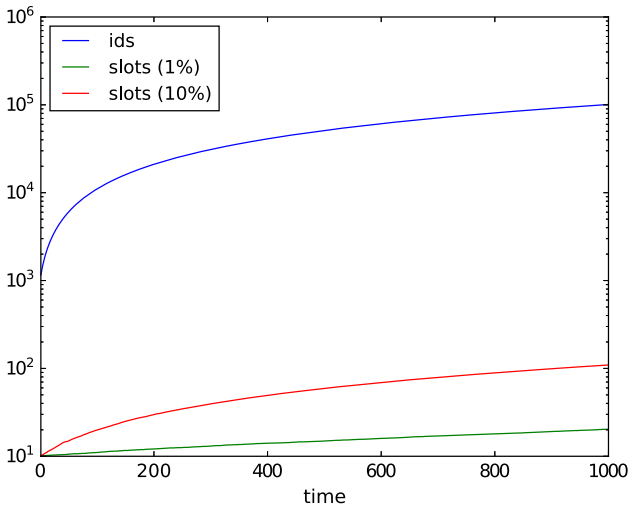

Fig. 9 Slots per server for 100 servers and 1000 clients, with one client retiring a new one arriving each 10 ms. Clients retire gracefully, unless a partition happens, with probability 1 and 10%

Figure 8 shows one run, with 100 servers and a fixed number of clients (1000). Each client is active one second, disconnects abruptly, remains 1 s offline, and reconnects choosing a new random server.

It can be seen that, along time the number of slots per server keeps increasing, as clients disconnect and reconnect. After 10 min, there are around 564 slots per server; this contrasts with the 5 slots per server in this scenario if no disconnects happened.

This unacceptable situation can be overcome by improving the logic of message exchanges, even while still allowing abrupt disconnects. The next experiment used such an improved logic where:

- clients periodically send the counter to the current server, plus to any other server for which the client still has corresponding tokens;
- servers periodically send the counter to any client for which the server still has corresponding slots open;
- both servers and clients "reply" to such messages after merging the received counter.

The need for this active behavior from the part of servers (as opposed to just reacting to messages received) is essential; otherwise, a slot for which no corresponding token had been created in a client would not be removed, as a client would not know that it exists. Figure 8 also shows the result of running this "smart" control algorithm, in the same scenario as before, leading to an incomparably better 10 slots per server along the run.

8.4 Client retirement

The final experiment performed involved assessing the impact of permanent client retirement on server state. The experiment involved 100 servers and 1000 starting clients, with a new client arriving each 10 ms and a random client being retired each 10 ms, to keep the number of current clients constant. Each retired client is always active and keeps affinity to the server. Message exchange is by the same "smart" algorithm from the previous section, but now clients try to retire gracefully, keeping exchanging messages until they have no tokens nor self-value to handoff.

We measure the impact of a client being partitioned when it decides to retire: with a given probability of partitioning, a client retires abruptly instead of performing the graceful retirement (causing slots to remain forever in servers). The experiment assesses whether the exchange algorithm can handle retirement discarding slots successfully. Figure 9 shows two such runs, for partition probabilities of 1 and 10%, showing the number of slots per server and also the number of ids (representing the vector size if standard CRDT counters were used). It can be seen that, even if some unrecoverable slot pollution happens, it is orders of magnitude better than in standard CRDTs.

9 Beyond counters

We have up to now addressed distributed counters, given their wide applicability and importance. Using counters was also useful for presentation purposes, as something concrete and widely known. The resulting mechanism and lessons learned are, however, applicable far beyond simple counters.

What we have devised is a mechanism which allows some value to be handed off reliably over unreliable networks, through multiple paths to allow availability in the face of temporary node failures or network partitions. Values are

moved from one place to another by "zeroing" the origin and later "adding" to the destination. Reporting is made by aggregating in two dimensions: "adding" values and taking the "maximum" of values. The value accounted at each node is updated by a commutative and associative operation which "inflates" the value. This prompts a generalization from simple counters over non-negative integers to more general domains.

The handoff counter CRDT can be generalized to any commutative monoid M (an algebraic structure with an associative and commutative binary operation ($\oplus$) and an identity element ($\mathbf{0}$)) which is also a join-semilattice (a set with a partial order ($\sqsubseteq$) for which there is a least upper bound ($x \sqcup y$) for any two elements x and y) with a least element ($\bot$), as long as it also satisfies:

$$\bot = \mathbf{0}$$
$$x \sqcup y \sqsubseteq x \oplus y$$

The CRDT state and definition of merge remain unchanged, except:

- Fields val, below, range of vals entries and token payload now store elements of M instead of simple integers;
- Those fields are initialized to $\mathbf{0}$ in the initialization of the CRDT; $\mathbf{0}$ is also used for resetting the self $\mathsf{vals}_i(i)$ entry in createtoken;
- The sum operation ($+$) over elements of the above fields is replaced by the $\oplus$ operation;
- The max operation used in mergevectors and aggregate is replaced by the $\sqcup$ operation;

In terms of client-visible mutation operations, instead of incr, any set of operations that are associative and commutative and that can be described as inflations over elements of M (i.e., such that $x \sqsubseteq f(x)$) can be made available.

In terms of reporting operations, instead of fetch, the data type can make available functions that can be defined over elements of M (that result from the aggregation made resorting to $\oplus$ and $\sqcup$).

9.1 Example: map of counters

Sometimes more than a single counter is needed. Instead of having a group of Handoff Counter CRDTs, a new CRDT can be devised, that holds a group of counters together, made available as a map from counter id to value. This will allow amortizing the cost of the CRDT state over the group of counters, instead of having per-counter overhead.

The CRDT for the map-of-counters can then be defined by making elements of M be maps from ids to integers and defining:

$$\mathbf{0} \doteq \{\}$$
$$x \oplus y \doteq \cup^{+}(x, y)$$
$$x \sqcup y \doteq \cup^{\max}(x, y)$$

and by making available:

$$\mathsf{fetch}(C_i, c) \doteq \mathsf{val}_i(c)$$
$$\mathsf{incr}(C_i, c) \doteq C_i\{\mathsf{val} = \cup^{+}(\mathsf{val}_i, \{c \mapsto 1\}),\ \mathsf{vals} = \mathsf{vals}_i\{i \mapsto \cup^{+}(\mathsf{vals}_i(i), \{c \mapsto 1\})\}\}$$

9.2 Example: PN-counter

A PN-Counter [26] can be both incremented and decremented. It can be implemented as a special case of the previous example, with two entries in the map: the p entry, counting increments, and the n entry, counting decrements. The fetch operation returns the difference between these values:

$$\mathsf{fetch}(C_i) \doteq \mathsf{fetch}(C_i, \mathsf{p}) - \mathsf{fetch}(C_i, \mathsf{n})$$
$$\mathsf{incr}(C_i) \doteq \mathsf{incr}(C_i, \mathsf{p})$$
$$\mathsf{decr}(C_i) \doteq \mathsf{incr}(C_i, \mathsf{n})$$

10 Discussion

The standard approach to achieving reliability and availability in distributed systems is to use a replicated service and distributed transactions, with a fault tolerant distributed commit protocol, that works if some majority of nodes are working (and not partitioned), e.g., Paxos Commit [15]. This standard approach attacks several problems in the same framework: network failures, temporary node failures and permanent node failures. By doing so, it incurs a performance cost, due to the need to communicate with several nodes, even when no failures occur. Our approach does not impose such cost: when no failures occur, a node playing the role of client only communicates with just one server.

Regarding availability, our approach (even after ensuring that increments were handed off to some server, so that a client can retire) is also better, as in case of server crash or link failure, it is enough that a single alternative server is available and reachable, as opposed to a majority of servers.

A significant characteristic of our approach is that it focuses on addressing network failures and temporary node failures, while not addressing permanent node failures, leaving them as an orthogonal issue to be attacked locally, e.g., through storage redundancy at each node. By not conflating temporary and permanent node failures, our approach does not impose on the distributed execution the cost of tolerating the latter.

Our approach can be seen to fit in the general philosophy described by [16] when aiming for "almost-infinite scaling": in avoiding large-scale distributed transactions and only assuming atomic updates to local durable state; in not requiring exactly-once messaging and having to cope with duplicates; in using uniquely identified entities; in remembering messages as state; in having entities manage "per-partner state". Our approach can be seen as applying that philosophy in designing a scalable distributed data type.

But the CRDT approach that we adopt goes further: since messages are unified with state, which evolves monotonically with time, the required message guarantees are even weaker than the at-least-once as assumed in the paper above. Messages with what has become an old version of the state need not be re-transmitted, as the current state includes all relevant information, subsuming the old state, so it suffices to keep transmitting the current state to enable progress (assuming that some messages will eventually get through).

11 Conclusion

We have addressed the problem of achieving very large scale counters over unreliable networks. In such scenarios providing strong consistency criteria precludes availability in general and, even if there are no network partitions, will impact performance. We have, therefore, defined what we called *ECDCs—Eventually Consistent Distributed Counters*, that provide the essence of counting (not losing increments or over-counting), while giving up the total ordering approach of the stronger classic distributed counters.

While ECDCs can be naively implemented using the CRDT approach, such implementation is not scalable, as it suffers from what is being perceived to be the major problem with CRDT approaches: the state size explosion due to pollution with node id related entries. This pollution involves not only current concurrent nodes, but also all already retired nodes.

We have presented a solution to ECDC, called *Handoff Counters*, that adopts the CRDT philosophy, making the "protocol" state be a part of the CRDT state. This allows a clear distinction between what is the durable state to be preserved after a crash, and what are temporary variables used in the computation. It also allows a correction assessment to focus on CRDT state and the merge operation, while allowing a simple distributed gossip algorithm to be used over an unreliable network (with arbitrary message loss, reordering or duplication).

Contrary to a naive CRDT based ECDC, our solution achieves scalability in two ways. First, node id related entries have a local nature, and are not propagated to the whole distributed system: we can have many thousands of participating nodes and only a few level 0 entries. Second, even not guaranteeing it in the general case, it allows garbage collection of entries for nodes that participate in the computation and then retire, in normal runs, while assuring correctness in arbitrary communication patterns. (We have also sketched an enhancement towards improving garbage collection upon retirement, which we leave for future work.) These two aspects make our approach usable for large scale scenarios, contrary to naive CRDT based counters using client-based ids, and avoiding the availability or reliability problems when using server-based CRDTs and remote invocation.

Moreover, our approach to overcoming the id explosion problem in CRDTs is not restricted to counters. As we have discussed, it is more generally applicable to other data types involving associative and commutative operations.

Acknowledgements We would like to thank Marc Shapiro, the anonymous reviewers and the associate editor for the comments that helped improve the paper.

References

1. Ascó, A.: SyncFree: WP1 deliverable. In: Application and Environment Requirements. https://syncfree.lip6.fr/attachments/article/46/d1_1.pdf, (2015)
2. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM **41**(5), 1020–1048 (1994)
3. Attiya, H., Dolev, S., Welch, J.L.: Connection management without retaining information. In: Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, vol. 2, pp. 622–631. (1995)
4. Attiya, H., Rappoport, R.: The level of handshake required for establishing a connection. In: Tel, G., Vitnyi, P. (eds.) Distributed Algorithms. Lecture Notes in Computer Science, vol. 857, pp. 179–193. Springer, Berlin Heidelberg (1994)
5. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '00, p. 7, New York (2000)
6. Cerf, V., Kahn, R.: A protocol for packet network intercommunication. IEEE Trans. Commun. **22**(5), 637–648 (1974)
7. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps. Softw. Pract. Exp. **46**(5), 709–719 (2016)
8. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. Cambridge University Press, Cambridge (2002)
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pp. 205–220, New York (2007)
10. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '87, pp. 1–12, New York (1987)
11. Fekete, A., Lynch, N., Mansour, Y., Spinelli, J.: The impossibility of implementing reliable communication in the face of crashes. J. ACM **40**(5), 1087–1107 (1993)
12. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002)

13. Goodman, J.R., Vernon, M.K., Woest, P.J.: Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-III, pp. 64–75, New York (1989)
14. Goudarzi, A.: Cassandra-4775: Counters 2.0. https://issues.apache.org/jira/browse/CASSANDRA-4775, (2012)
15. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006)
16. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: CIDR, pp. 132–141. www.cidrdb.org, (2007)
17. Herlihy, M., Lim, B.-H., Shavit, N.: Scalable concurrent counting. ACM Trans. Comput. Syst. **13**(4), 343–364 (1995)
18. Herlihy, Maurice P., Wing, Jeannette M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
19. Hernandez, J., Phillips, I.: Weibull mixture model to characterise end-to-end Internet delay at coarse time-scales. IEE Proc. Commun. **153**(2), 295–304 (2006)
20. Jesus, P., Baquero, C., Almeida, P.S.: Flow updating: fault-tolerant aggrega-tion for dynamic networks. J. Parallel Distrib. Comput. **78**, 53–64 (2015)
21. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming, CUFP '10, p. 14:1, New York (2010)
22. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
23. Lebresne, S.: Cassandra-2495: add a proper retry mechanism for counters in case of failed request. https://issues.apache.org/jira/browse/CASSANDRA-2495, (2011)
24. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '87, pp. 137–151, New York (1987)
25. Parker Jr., D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. IEEE Trans. Softw. Eng. **9**(3), 240–247 (1983)
26. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of convergent and commutative replicated data types. In: Rapport de recherche 7506, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, (2011)
27. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS'11, pp. 386–400. Springer, Berlin (2011)
28. Shavit, N., Zemach, A.: Diffracting trees. ACM Trans. Comput. Syst. **14**(4), 385–428 (1996)
29. Stone, H.S.: Database applications of the fetch-and-add instruction. IEEE Trans. Comput. **33**(7), 604–612 (1984)
30. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.W.: Session guarantees for weakly consistent replicated data. In: IEEE Computer Society and Proceedings of the Third International Conference on Parallel and Distributed Information Systems, PDIS '94, pp. 140–149, Washington (1994)
31. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
32. Wattenhofer, R., Widmayer, P.: The counting pyramid: an adaptive distributed counting scheme. J. Parallel Distrib. Comput. **64**(4), 449–460 (2004)
33. Yew, P.-C., Tzeng, N.-F., Lawrie, D.H.: Distributing hot-spot addressing in large-scale multiprocessors. IEEE Trans. Comput. **36**(4), 388–395 (1987)