View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Universidade do Minho: RepositoriUM

Exactly-Once Quantity Transfer

Ali Shoker, Paulo Sérgio Almeida and Carlos Baquero
HASLab / INESC TEC & University of Minho
Braga, Portugal

***Abstract*—Strongly consistent systems supporting distributed transactions can be prone to high latency and do not tolerate partitions. The present trend of using weaker forms of consistency, to achieve high availability, poses notable challenges in writing applications due to the lack of linearizability, e.g., to ensure global invariants, or perform mutator operations on a distributed datatype. This paper addresses a specific problem: the exactly-once transfer of a "quantity" from one node to another on an unreliable network (coping with message duplication, loss, or reordering) and without any form of global synchronization. This allows preserving a global property (the sum of quantities remains unchanged) without requiring global linearizability and only through using pairwise interactions between nodes, therefore allowing partitions in the system. We present the novel quantity-transfer algorithm while focusing on a specific use-case: a redistribution protocol to keep the quantities in a set of nodes balanced; in particular, averaging a shared real number across nodes. Since this is a work in progress, we briefly discuss the correctness of the protocol, and we leave potential extensions and empirical evaluations for future work.**

***Keywords*-Distributed monoid-like data-types; exactly-once quantity-transfer, idempotence.**

I. Introduction

The trend of distributed storage systems nowadays is to use relaxed forms of consistency to improve availability. This is often established through delaying inter-replica synchronization and offering the requesting client a fast (though stale) response based on the local state, that is coordinated with other replicas off the critical path in an asynchronous fashion. In order to relax consistency in a way that is tolerated by application semantics, that semantics needs to be considered. In this paper, we focus on *monoid*-like datatypes that hold partitionable quantities, that can be split and added back, such as counters or multi-sets.

In the simplest formulation, we consider the distributed datatype state to depict a *quantity*, say a collection of tickets, that is partitioned among a set of nodes. Contrary to replicated systems where the same *total* value is present at all replicas, here the local quantity is a *part* of the whole, and can be immediately operated upon, e.g., increased or decreased by local requests, with no need for node synchronization, resulting in high availability and low latency. Local operations only depend on the quantity locally available and, by being conservative, a global invariant can be preserved as a result from a local invariant: if a decrease is limited to the local quantity, it will remain non-negative, and therefore, so will the sum of all quantities in the system.

Over time, the quantities can become unbalanced across nodes: excess of tickets on some nodes and scarcity on others. This motivates the asynchronous transfer of quantities between nodes in order to balance them. The transfer can be performed pairwise, opportunistically, without requiring global connectivity, and therefore with part of the system being partitioned. The challenge of this approach is how to perform the transfer reliably, with an exactly-once guarantee, to preserve the total quantity in the system.

Many *redistribution* protocols (e.g., [1], [2], [3], [4], [5], [6]) have been proposed to redistribute quantities, however none was immune to message duplication, i.e., the messages involved were not idempotent. In this work, we propose an new redistribution (a.k.a., quantity transfer) protocol with idempotent messages.

Redistribution protocols in the 80's suffered either from latency issues due to resource locking and extensive use of 2PC (two-phase commit) or from delivery ordering constraints [2], [1], [7]. The demarcation protocol [3], [4] was then proposed as an alternative solution that is immune to message delays and reception order: Whenever a node wishes to perform an unsafe operation (e.g., may violate an invariant), it requests that the other node perform a corresponding safe operation and waits for notification. (The addressed problem in this protocol was mainly redistributing limits by granting or receiving a slack which is analogous to the quantity exchange problem we address here.) This allowed the propagation of any number of consecutive changes to be made without having to wait for acknowledgments. For these reasons, in addition to its simplicity, the demarcation protocol is still being used nowadays [8], [9], [10], [11]. However, the authors themselves admit that the protocol mis-behaves if no assumptions about message delivery are made. Even though safety is not violated, over time, under message duplication or loss, resources can be "lost" or limits can become overly restrictive, as explained in [6] and [8]. Krishnakumar and Jain tried to avoid these problems in mobile inventory services [6]; however, they used multiple 2PC phases and a third party server, which not only it is very costly but also a single point of failure.

Addressing the problem of reliable communication between two parties, in practice, requires retaining unique message identifiers for the set of received, and delivered, messages at the destination endpoint. Messages can be retransmitted when not acknowledged for some time, and the identifier set in the destination can always filter out received duplicates and ensure exactly-once delivery. The filter set, however, will grow linearly with the number of messages received. In settings that aim for reliable FIFO communication, the long term space requirements in the destination endpoint can be improved to be linear with the number of sources, by storing for each source the number that identifies the last message delivered. Messages received out of order must still be buffered. (Notice that quantity transfers do not necessarily require FIFO, since adding received quantities is commutative.)

Transport layer protocols, such as TCP/IP, try to ensure that only within a connection, data sent from one end-point is delivered *exactly-once* to the other end-point, and in FIFO order [12]. However, if a connection breaks while non-acknowledged sent messages are present, those messages are only guaranteed to be delivered *at-most-once*. To enforce exactly-once, the connection management protocol would have to retain connection specific information between different connection *incarnations* [13], something that TCP/IP avoids [14]. Even weaker properties are provided by UDP, where messages can be lost, duplicated or re-ordered.

Reducing storage requirements at the destination are only possible at the expense of time. Attiya and Rappoport have shown, in [13], that endpoints can retain counters that are not connection specific if *at least* a three-way handshake is used to establish a connection. This incurs a latency cost in the first exactly-once transmission. Since its quite reasonable to expect quantity transfers among geo-distributed data centers that aim to keep local escrow for high-availability, and these will have be connected by high latency links, the three-way handshake is particularly taxing for short lived connections.

In this paper, we leverage the fact that we focus on specific short lived task, *quantity transfer*, to make as much progress as possible in the two initial communications of the three-way handshake and use the third communication step to do the exactly-once transfer. The extra information that is piggy-backed in the initial steps, while important for progress, is not required to be done exactly-once; and thus any undetected duplications do not harm the correctness of the exchange and the conservation of quantities. The protocol keeps extra information during the exchange, but after the transfer occurs the state in a node only stores a globally unique node identifier and one counter (nodes that both send and receive store two counters).

The original idea in this paper is inspired from *Handoff Counters* [15] where the authors design scalable eventually consistent counter CRDTs [16] that can work correctly despite network partitions, and avoid the identity explosion problems of previous CRDTs like G-Counters [16]. However, this paper generalizes this idea to quantity transfers in any "splittable" datatype and also expands the application spectrum of the idea to new possible use-cases.

Given the limited paper space, we present our redistribution protocol addressing a specific use-case: reliably moving quantities from a source host to a destination host. A quantity is a simple abstraction that represents a value that can be split into two values that added back together produces the original value. A simple example is that of money transfer between two wallets. Money in origin wallet o in variable $o.\mathsf{val}$ is split into $o.\mathsf{val}'$ and m, with $o.\mathsf{val} = o.\mathsf{val}' + m$; then m is transferred, exactly-once, to a destination wallet d that changes the stored amount to $d.\mathsf{val}' = d.\mathsf{val} + m$. No money is lost or created, since $o.\mathsf{val} + d.\mathsf{val} = o.\mathsf{val}' + d.\mathsf{val}'$. The same principle can be applied to many applications: stock escrow, token transfers, service handoffs, etc [17], [6], [18].

$$
\begin{aligned}
\mathsf{0} &\doteq 0\\
\oplus &\doteq +\\
\mathsf{needs}(x,y) &\doteq \frac{y-x+|y-x|}{4}\\
\mathsf{split}(x,h) &\doteq (\frac{x-h+|x-h|}{2}, \frac{x+h-|x-h|}{2})
\end{aligned}
$$

Fig. 1. $\mathbb{R}$ Data type example: Positive reals that ask for half difference (when smaller) and give as much as possible.

In the future, we plan to present this concept more formally, including transfer policies discussions, protocol variants, and empirical experimentation.

II. Protocol

A. 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.

The system is composed of n nodes. 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. We assume no Byzantine or Rational behaviors.

B. Payload Data Types

Valid data values types T must be commutative monoids with a generic sum operator $\oplus$, and identity element $\mathsf{0}$. (Splittable data values in the related work history were called partitionable, fragmentable, or even escrowable; in this paper we choose to use commutative monoids as it captures the essential mathematical properties that are actually required.) Fragmenting a quantity is done via a user defined split function; it can be any function such that $(x', q) = \mathsf{split}(x, h) \Leftrightarrow x' \oplus q = x$. Some split functions can use the hint h to further ensure that $\mathsf{0} \leq q \leq h$, but this is not needed for correction. Load-balancing is abstracted via a user defined needs function that compares a local amount to a remote amount, deciding how much to ask. It can be such that $h = \mathsf{needs}(x, y)$ creates a hint h and that typically we will have $\mathsf{0} \leq h \leq y$, with h representing a value that is beneficial to split from y and move to x.

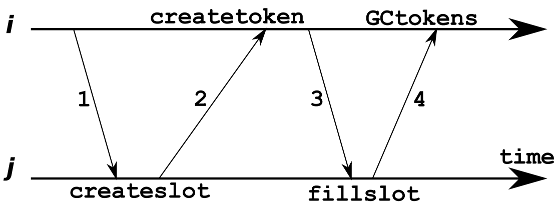

Fig. 2. Basic fault-free communication scenario.

val data value reported by fetch;
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 $((sck, dck), D)$ containing a pair of logical clocks and a data value;
tokens map from destination ids to pairs $((sck, dck), D)$ containing a pair of logical clocks and a data value;

Fig. 3. Replica state (record fields)

C. Use-case and Redistribution Policy

Since discussing redistribution policies is not the focus of this short article, we assume the following use-case: a quantity (e.g., a *real* number) that needs to be evenly balanced over all replicas, so that they try to keep similar fractions of the global amount. Replicas periodically share their values; a replica that owns more credits will *split* its value and transfer them to another replica that *needs* it (i.e., has scarce resources). Fig. 1 shows how split and needs are performed over real numbers. In addition, we assume that redistribution occurs in a periodic fashion. We aim at supporting more policies in the future.

While Fig. 1 shows how these datatype-specific functions can be implemented for positive reals, we have also built and tested similar definitions for integers and for maps from identifiers to integers, that would more directly represent stock/inventory abstractions.

D. The Algorithm

Our protocol, running on each node, is depicted in Algorithm 1, and makes use of auxiliary functions defined in Fig. 4.. Each node has access to a globally unique identifier i, a set of neighbors n_i, and an initial quantity v^0 of a valid data type. The algorithm shows operations that can be invoked locally, to act on the local data type; how to handle messages received; and triggers periodic transmissions to other nodes. The node state is a record with fields shown in Fig. 3.

a) Overview: Fig. 2 depicts the basic fault-free communication scenario of our algorithm. Node j receives a (periodically sent) message from i. Node j notices that i has more resources (a larger quantity) and asks for some quantity by creating a *slot* (a receptor) for i. When i eventually receives the message, it checks if it still has extra resources and *splits* its local val, creating a *token* containing the split quantity, and sends a message with the token to j. As soon as j receives the token, it adds the quantity to its val, removing the slot (function fillslots) and replies back to i. Finally, i can safely garbage collect the token (function GCtokens). We describe the algorithm in more detail in the following.

Algorithm 1: Distributed algorithm for a generic node i.

```
constants:
    $i$, globally unique node id
    $n_i$, set of neighbors of node $i$
    $v^0$, initial data value of type T
state:
    $C_i = \{\mathsf{val} = v^0, \mathsf{sck} = 0, \mathsf{dck} = 0,$
            $\mathsf{slots} = \{\}, \mathsf{tokens} = \{\}\}$
local $\mathsf{fetch}_i$
    return $C_i.\mathsf{val}$
local $\mathsf{plus}_i(q)$
    $C_i := C_i\{\mathsf{val} = \mathsf{val}_i \oplus q\}$
local $\mathsf{minus}_i(q)$
    let $(v, q') = \mathsf{split}(C_i.\mathsf{val}, q)$
    $C_i := C_i\{\mathsf{val} = v\}$
    return $q'$
on $\mathsf{receive}_{j,i}(C_j)$
    $C_i := \mathsf{fillslots}(C_i, C_j)$
    $C_i := \mathsf{createslot}(C_i, C_j)$
    $C_i := \mathsf{GCtokens}(C_i, C_j)$
    $C_i := \mathsf{createtoken}(C_i, C_j)$
periodically
    for $j \in n_i$ do
        let $m = C_i\{$
            $\mathsf{slots} = \{(k, s) \in \mathsf{slots}_i \mid k = j\},$
            $\mathsf{tokens} = \{(k, s) \in \mathsf{tokens}_i \mid k = j\}\}$
        $\mathsf{send}_{i,j}(m)$
```

b) 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 emphasize 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 form $\{x \in S \mid P(x)\}$. 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. 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".

c) Local functions: Function fetch returns the val field; operation plus adds an amount to val; operation minus attempts to subtract an amount from val, limited to the available quantity, as val cannot go below zero, returning the amount

$$
\begin{aligned}
\mathsf{fillslots}(C_i, C_j) \quad &\doteq \quad \mathbf{if}\ (i, (ck, q)) \in \mathsf{tokens}_j \wedge (j, (ck, \_)) \in \mathsf{slots}_i \\
&\qquad \mathbf{then}\ C_i\{\mathsf{val} = \mathsf{val}_i \oplus q, \mathsf{slots} = \{j\} \lhd \mathsf{slots}_i\} \\
&\qquad \mathbf{else\ if}\ (j, ((sck, \_), \_)) \in \mathsf{slots}_i \wedge \mathsf{sck}_j > sck \\
&\qquad\quad \mathbf{then}\ C_i\{\mathsf{slots} = \{j\} \lhd \mathsf{slots}_i\} \\
&\qquad \mathbf{else}\ C_i \\
\mathsf{createslot}(C_i, C_j) \quad &\doteq \quad \mathbf{let}\ h = \mathsf{needs}(\mathsf{val}_i, \mathsf{val}_j) \\
&\qquad \mathbf{if}\ j \notin \mathsf{dom}(\mathsf{slots}_i) \wedge h \neq 0 \\
&\qquad\quad \mathbf{then}\ C_i\{\mathsf{slots} = \mathsf{slots}_i\{j \mapsto ((\mathsf{sck}_j, \mathsf{dck}_i), h)\}, \mathsf{dck} = \mathsf{dck}_i + 1\} \\
&\qquad \mathbf{else}\ C_i \\
\mathsf{GCtokens}(C_i, C_j) \quad &\doteq \quad \mathbf{if}\ j \in \mathsf{dom}(\mathsf{tokens}_i) \wedge (i \in \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{snd}(\mathsf{fst}(\mathsf{tokens}_i(j))) < \mathsf{snd}(\mathsf{fst}(\mathsf{slots}_j(i))) \\
&\qquad\qquad \vee\ i \notin \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{snd}(\mathsf{fst}(\mathsf{tokens}_i(j))) < \mathsf{dck}_j) \\
&\qquad\quad \mathbf{then}\ C_i\{\mathsf{tokens} = \{j\} \lhd \mathsf{tokens}_i\} \\
&\qquad \mathbf{else}\ C_i \\
\mathsf{createtoken}(C_i, C_j) \quad &\doteq \quad \mathbf{if}\ i \in \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{fst}(\mathsf{fst}(\mathsf{slots}_j(i))) = \mathsf{sck}_i \\
&\qquad\quad \mathbf{then\ let}\ (v, q) = \mathsf{split}(\mathsf{val}_i, \mathsf{snd}(\mathsf{slots}_j(i))) \\
&\qquad\qquad\qquad C_i\{\mathsf{tokens} = \mathsf{tokens}_i\{j \mapsto (\mathsf{fst}(\mathsf{slots}_j(i)), q)\}, \\
&\qquad\qquad\qquad\quad \mathsf{val} = v, \\
&\qquad\qquad\qquad\quad \mathsf{sck} = \mathsf{sck}_i + 1\} \\
&\qquad \mathbf{else}\ C_i
\end{aligned}
$$

Fig. 4. Auxiliary functions in receive.

actually subtracted. It makes use of function split that splits val into two amounts.

d) Sending: Periodically, each node i sends a message to each neighbor j, containing the *view* of its state, containing only the information that is relevant to the specific receiver j. Notice that while the connection between two nodes is unreliable, as sending is done periodically, eventually a message will be received. We do not specify a specific network topology, but the algorithm will balance values in each connected component. For simplicity the reader can picture a simple topology with a single connected component, such as a ring or a complete graph.

e) Receiving: Once i receives a new message from another node j, it incorporates it into its state by performing four steps, using the functions from Fig. 4. These functions receive as argument two state records and return the new state, possibly with some of its fields updated.

Node i starts by checking if it has open slots for j and tries to fill them if so (fillslots); it first verifies if j has a token for i (that must have been previously created) and if that very token has a locally opened slot on i (a matching ck). In this case, i adds the received amount q to its val and removes the corresponding slot. On the contrary, if a slot for j exists on i but ck is not matching, i tries to garbage collect the slot if the source clock of j is ahead the clock registered in the designated local slot. This basically means that j has already created a token to another node (and incremented it local clock sck_j) to acquire lacking amounts and discarded creating a token corresponding to a previously sent slot by i.

Then it decides whether it should create a slot for j (createslot); if i has no open slot for j, it opens a corresponding slot only if $h \neq 0$ using the needs function (meaning that j has excess amount to offer to i). Thus, i stores the newly created slot that corresponds to (sck_j, dck_i) and advances its sending clock dck_i. Notice that since this is only done if i has no open slots for j, this guarantees that no slots are created for duplicate messages if sck_j has not been incremented (otherwise garbage collection would have occurred and a new slot creation is allowed).

The next step is to check if node i has a token for j due to a previous contact. In this case, the token may have been successfully merged by j, and thus this token has to be garbage collected GCtokens if: j has no open slots for i and its slots clock dck_j is ahead the said clock of the stored token. The last step is to create a token if j has an open slot for i such that the clock of the slot and node i are matching. In this case, i splits its val using split to hand it off to j. Recall that, split shall not return the exact amount needed by j if val is not large enough according to the policy in Fig. 1).

III. Correctness

We provide an informal proof for the correctness (safety and liveness) of our protocol. We postpone formal proofs to an extended version due to page limits.

A. Safety

We explain safety by focusing on duplicated messages, reordering, and "lost resources" problems since this is the aim of the protocol. We omit the cases of lost message as we assume eventual delivery and we explain re-ordering when needed.

Consider phase 1 in Fig. 2; this phase is safe under message duplication since a duplicate slot will never be created as per the conditions in $\mathsf{createslot}$. In phase 2, upon receiving an open slot, i creates a corresponding token and advances its clock sck_i. Receiving a duplicate slot will have no effect since the slot's clock will not be matching anymore with sck_i. In phase 3, once j receives a token from i it fills the corresponding slot (and deletes it); receiving another duplicate of the same token will have no effect since there is no receptor slot at j. The final phase 4 is also duplication-safe since a token will be garbage collected only once.

The algorithm is also safe against message re-ordering. As depicted on Fig. 2, there are only two re-ordering possibilities: (1) Phase 1 and 3 are re-ordered. This is impossible to occur since there is no way that j creates a slot (and consequently i creates the corresponding token) unless if j received a prior message, i.e., in phase 1. (2) Phase 2 and 4 are re-ordered. This case is safe as i would simply discard the message since it has no matching token for j.

As for "lost resources", the only way for offering resources is to $\mathsf{createtoken}$ (in which split is called); but as explained above, this can only occur if the other node, j, has already a corresponding open slot. In addition, j could not delete a slot unless i's sck is ahead the slot's clock (**else if** in $\mathsf{fillslots}$), which means that i has already sent a token to another node and it could not offer j a quantity; thus j must eventually add (in $\mathsf{fillslots}$) the split quantity (in $\mathsf{createtoken}$) sent in the token from i (phase 3).

B. Liveness

As for liveness, we first recall that we assume eventual delivery of messages across all system nodes. Therefore, network partitions, though possible, are considered transient and messages will eventually go through. Now, we informally demonstrate the liveness of the algorithm using Fig. 2.

First, notice in Fig. 2 that node j can $\mathsf{createslot}$ and $\mathsf{fillslots}$ without blocking. In fact, j can always run $\mathsf{createslot}$ to create a slot for i if its quantity is less than that of i. (An existing slot would have been removed in $\mathsf{fillslots}$.) In addition, j does not have to wait until a token is received from i; however, it could create other slots to other nodes too. Node j can thus remove a created slot only when it receives a matching token as shown in $\mathsf{fillslots}$ function in Fig. 4; otherwise, the slot is kept (until it is eventually garbage collected), which has no impact on progress.

As for node i, it only creates a token in $\mathsf{createtoken}$ if an open slot is received and it still has larger quantity (as it could have transferred some to another node by sending a prior token). This is okay since j will eventually garbage collect the corresponding slot. After creating the token, i will increment its clock sck_i; this prevents it from creating any other token to j unless it has received an ACK (i.e., a new slot with matching sck_i) from it, since the condition $\mathsf{fst}(\mathsf{fst}(\mathsf{slots}_j(i))) = \mathsf{sck}_i$ will not be satisfied in $\mathsf{createtoken}$. However, in all cases, i will be able to create tokens to other nodes in the system if a matching slot (holding the new incremented sck_i) is received and i has extra quantity to transfer. Node i can then send new tokens to j once an ACK from j is received and $\mathsf{GCtokens}$ is applied (which will eventually occur).

Finally, the protocol will not block due to the transfer policy since nodes with larger quantities will always offer quantities to other nodes. This is guaranteed as we assume all nodes are periodically exchanging states even if no local events occurred. This can obviously be done in more efficient ways according to the policy chosen (which we do not discuss here).

IV. Discussion and Future Work

In this paper, we focused on presenting the idea of the algorithm on a simple real number averaging example. However, the reader can easily notice that this algorithm can be used in other cases of similar $split/merge$ nature, as in [17], [5], [19], [8], etc. We described our algorithm keeping in mind an averaging policy whereas multiple policies could by addressed. In this specific policy we did not address if averaging occurs or not, but we rather focused on the correctness of the algorithm. We conducted preliminary empirical evaluations to this averaging problem on up to 1000 nodes and the results seem promising: all nodes started with high variance of quantities and came to an average value, while all meta-data (tokens and slots) were garbage collected. We aim to provide more evaluation and comparison results in the future.

In addition, we assumed that messages are simply disseminated in a periodic fashion (e.g., through gossiping); clearly, other options can be of interest too like having the node with scarce resources ask other nodes (avoiding periodic dissemination). We have also assumed that no transitive sending occurs between nodes, meaning that a third-party node could not deliver a message on behalf of another node. We think that this case is worth more focus in the future.

V. Related Work

The problem of quantity transfer or redistribution (sometimes called repartitioning or reconfiguration) first appeared in the context of database transactions by Carvalho et al. [2] to maintain the invariants (or *limits*) on different servers as in the Escrow Transactional method [20], [17]. The aim was to redistribute an "escrowable" (or fragmentable) value (a limit) over multiple partitions in a distributed storage by "splitting" an amount on one replica and adding it to another. This idea of "splitting" was first proposed by Davidson et al., in [21], inspired by [22] in the context of reliable networks. Another protocol was later proposed in [1] where a node can "borrow" elements from neighbors (and waits) until acknowledged. They used "partitionable operators" (similar to the $\oplus$ operator we use in our paper); however, this protocol had impractical weaknesses like blocking, re-ordering, and duplication.

The famous "demarcation" protocol was then introduced by Barbara et al. in [3], [4]. This protocol aimed at maintaining invariants in distributed databases using escrow-like method [20], [17]. The demarcation protocol was immune to message delays and the order of reception and would allow the propagation of any number of consecutive changes to be made without having to wait for acknowledgments. This was a substantial improvement over its predecessors as it could tolerate network partitions. The protocol however is not immune to network problems like message dropping and duplication which can lead to incorrect behaviors like more conservative limits (in case of limit management) or "lost resources" (in case of quantity transfer as in our work). Several protocols were then proposed by Krishnakumar et al., in [7], [23], [6], in the context of mobile services and inventory to overcome these problems; however, they used many 2PC phases which was not practical for systems that focus on low latency and high availability.

To the best of our knowledge, no further improvements were made to the demarcation protocol, and it is still being used by current systems despite its aforementioned caveats [8], [9], [10], [11].

VI. Conclusion

We introduced a new redistribution protocol to perform an exactly-once transfer of a "quantity" from one node to another in a distributed system. The protocol is immune to delivery problems like message dropping, duplication, and re-ordering. Although this protocol addressed a single "averaging" problem of a distributed *real* number, it is easy to adapt to other contexts, use-cases, and applications. The paper focused on presenting the algorithm and showing its correctness properties leaving other details to a future work, like distribution and splitting policies and other variants of the protocol. Since this is a work in progress, we aim at presenting more formal presentation accompanied with experimentations in a longer version. We already had some promising results showing that the protocol brings all replicas (up to one thousand) to an average value without leaving any garbage traces or meta-data.

VII. Acknowledgment

This work is financed by the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013; and by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 609551, SyncFree project.

References

[1] N. Soparkar and A. Silberschatz, "Data-value partitioning and virtual messages," Austin, TX, USA, Tech. Rep., 1989.

[2] O. S. Carvalho and G. Roucairol, "On the distribution of an assertion," in *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '82. New York, NY, USA: ACM, 1982, pp. 121–131. [Online]. Available: http://doi.acm.org/10.1145/800220.806689

[3] D. Barbará and H. Garcia-Molina, "The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems," in *Proceedings of the 3rd International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '92. London, UK, UK: Springer-Verlag, 1992, pp. 373–388. [Online]. Available: http://dl.acm.org/citation.cfm?id=645336.649877

[4] D. Barbará-Millá and H. Garcia-Molina, "The demarcation protocol: A technique for maintaining constraints in distributed database systems," *The VLDB Journal*, vol. 3, no. 3, pp. 325–353, Jul. 1994. [Online]. Available: http://dx.doi.org/10.1007/BF01232643

[5] R. Jain and N. Krishnakumar, "Network support for personal information services to pcs users," in *Networks for Personal Communications, 1994. Conference Proceedings., 1994*, Mar 1994, pp. 1–7.

[6] N. Krishnakumar and R. Jain, "Escrow techniques for mobile sales and inventory applications," *Wirel. Netw.*, vol. 3, no. 3, pp. 235–246, Aug. 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1019161318592

[7] N. Krishnakumar and A. J. Bernstein, "High throughput escrow algorithms for replicated databases," in *Proceedings of the 18th International Conference on Very Large Data Bases*, ser. VLDB '92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 175–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=645918.672481

[8] V. Balegas, D. Serra, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Shapiro, and M. Najafzadeh, "Extending eventually consistent cloud databases for enforcing numeric invariants," *CoRR*, vol. abs/1503.09052, 2015. [Online]. Available: http://arxiv.org/abs/1503.09052

[9] A. Elmagarmid, J. Jing, and O. Bukhres, "An efficient and reliable reservation algorithm for mobile transactions," in *Proceedings of the Fourth International Conference on Information and Knowledge Management*, ser. CIKM '95. New York, NY, USA: ACM, 1995, pp. 90–95. [Online]. Available: http://doi.acm.org/10.1145/221270.221338

[10] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 253–264, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687657

[11] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 113–126. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465363

[12] P. Helland, "Idempotence is not a medical condition," *Queue*, vol. 10, no. 4, pp. 30:30–30:46, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2181796.2187821

[13] H. Attiya and R. Rappoport, "The level of handshake required for establishing a connection," in *Distributed Algorithms*, ser. Lecture Notes in Computer Science, G. Tel and P. Vitnyi, Eds. Springer Berlin Heidelberg, 1994, vol. 857, pp. 179–193. [Online]. Available: http://dx.doi.org/10.1007/BFb0020433

[14] R.Braden, "Tcp extensions for transactions," *RFC*, Jul. 1994. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1644.txt

[15] P. S. Almeida and C. Baquero, "Scalable eventually consistent counters over unreliable networks," *CoRR*, vol. abs/1307.3207, 2013. [Online]. Available: http://arxiv.org/abs/1307.3207

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050613.2050642

[17] A. Kumar and M. Stonebraker, "Semantics based transaction management techniques for replicated data," *SIGMOD Rec.*, vol. 17, no. 3, pp. 117–125, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/971701.50215

[18] Paulo S'ergio Almeida and Ali Shoker and Carlos Baquero., "Efficient State-based CRDTs by Delta-Mutation," in *Proceedings of the International Conference of Networked sYStems*, ser. NETYS'15. Springer, May 2015.

[19] M. Mouly and M.-B. Pautet, *The GSM System for Mobile Communications*. Telecom Publishing, 1992.

[20] P. E. O'Neil, "The escrow transactional method," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 405–430, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/7239.7265

[21] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a partitioned network: A survey," *ACM Comput. Surv.*, vol. 17, no. 3, pp. 341–370, Sep. 1985. [Online]. Available: http://doi.acm.org/10.1145/5505.5508

[22] M. Hammer and D. Shipman, "Reliability mechanisms for sdd-1: A system for distributed databases," *ACM Trans. Database Syst.*, vol. 5, no. 4, pp. 431–466, Dec. 1980. [Online]. Available: http://doi.acm.org/10.1145/320610.320621

[23] N. Krishnakumar and R. Jain, "High throughput escrow algorithms for replicated databases," in *Proceedings of the MOBIDATA Workshop*, ser. MOBIDATA '94. Rutgers Univ., 1994.