(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
24 April 2014 (24.04.2014)

WIPO | PCT

(10) International Publication Number
WO 2014/060393 A1

(51) International Patent Classification:
G06F 9/52 (2006.01)

(21) International Application Number:
PCT/EP2013/071489

(22) International Filing Date:
15 October 2013 (15.10.2013)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
12188516.4 15 October 2012 (15.10.2012) EP

(71) Applicant: **NEC EUROPE LTD.** [DE/DE]; Kurfuersten-Anlage 36, 69115 Heidelberg (DE).

(72) Inventor: **DOBRE, Dan**; Albert-Mays-Strasse 8, 69115 Heidelberg (DE).

(74) Agent: **ULLRICH & NAUMANN**; Schneidmühlstraße 21, 69115 Heidelberg (DE).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

[Continued on next page]

(54) Title: A METHOD FOR BUILDING A RANKED REGISTER AND A COMPARE-AND-SWAP OBJECT, A RANKED REGISTER AND COMPARE-AND-SWAP-OBJECTS

(57) Abstract: For allowing a very reliable access of shared data within a storage by multiple clients a method for building a concurrent data structure in the form of a ranked register based on a Compare-And-Swap, CAS, functionality and an according ranked register is claimed. Further, a method for building a CAS object on the basis of such a ranked register and an according CAS object are claimed. Further, use of said ranked register and CAS object for transactional access to shared data within a storage by multiple clients is claimed.

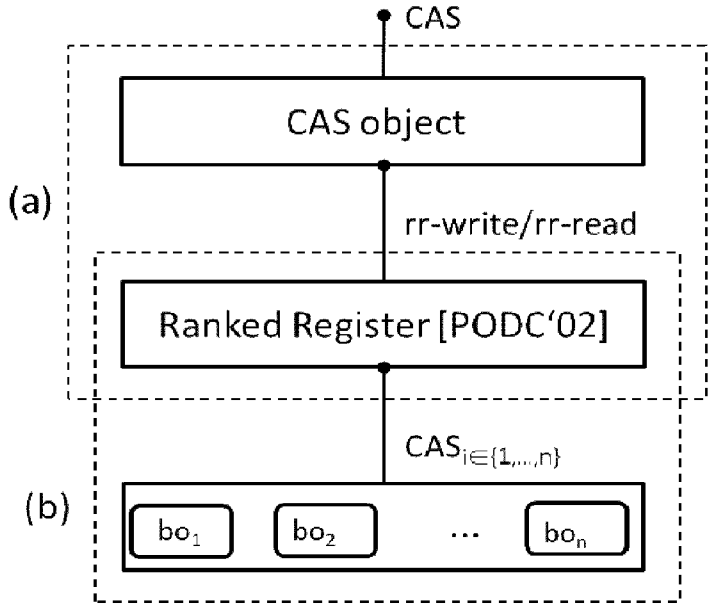

Fig. 2

WO 2014/060393 A1

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

A METHOD FOR BUILDING A RANKED REGISTER AND A COMPARE-AND-SWAP OBJECT, A RANKED REGISTER AND COMPARE-AND-SWAP-OBJECTS

The present invention relates to a method for building a concurrent data structure in the form of a ranked register, RR, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns.

Further, the present invention relates to a concurrent data structure in the form of a ranked register, RR, preferably built by the method according to any one of claims 1 to 6, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns.

Further, the present invention relates to a method for building a Compare-And-Swap, CAS, object, wherein a concurrent data structure in the form of a ranked register, RR, is used, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns,

wherein increasing versions of values within data objects are stored for ensuring that newer values are never overwritten with older ones.

Further, the present invention relates to a Compare-And-Swap, CAS, object, preferably built by the method according to any one of claims 8 to 14, based on a concurrent data structure in the form of a ranked register, RR, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns,

wherein increasing versions of values within data objects are stored for ensuring that newer values are never overwritten with older ones.

Further, the present invention relates to a method for providing a Compare-And-Swap, CAS, object for transactional access to shared data within a storage by multiple clients,

wherein the storage comprises multiple storage units, each of which being accessible via a separate Application Programming Interface, API, each of which comprising a Compare-And-Swap, CAS, primitive,

wherein an entirety of multiple such storage units is assigned one further API comprising a further Compare-And-Swap, CAS, primitive for access by the clients and

wherein as the CAS object a CAS object according to claim 15 based on a ranked register according to claim 7 is used,

so that the CAS object comprises multiple CAS objects in the form of said multiple storage units with their APIs.

Further, the present invention relates to a Compare-And-Swap, CAS, object for transactional access to shared data within a storage by multiple clients,

wherein the storage comprises multiple storage units, each of which being accessible via a separate Application Programming Interface, API, each of which comprising a Compare-And-Swap, CAS, primitive,

wherein an entirety of multiple such storage units is assigned one further API comprising a further Compare-And-Swap, CAS, primitive for access by the clients and
wherein the CAS object is a CAS object according to claim 15 based on a ranked register according to claim 7,
so that the CAS object comprises multiple CAS objects in the form of said multiple storage units with their APIs.

Finally, the present invention relates to a method for providing transactional access to shared data within a storage by multiple clients,
wherein the storage comprises multiple storage units, each of which being accessible via a separate Application Programming Interface, API, each of which comprising a Compare-And-Swap, CAS, primitive,
wherein an entirety of multiple such storage units is assigned one further API comprising a further Compare-And-Swap, CAS, primitive for access by the clients and
wherein as a functional connection between the entirety of multiple such storage units and the further API a ranked register is used.

Technical Background:

Thanks to the advent of cloud services, distributed databases have benefitted from deployments spanning a cluster of machines within a single cloud. In such a setting, replication is the advocated solution to achieve data scalability and fault-tolerant durability. However, entire clouds may fail, rendering critical data unavailable, which ultimately leads to revenue losses.

With this invention we are addressing the following problem: How can we leverage multiple storage clouds or servers to enable transactional access to shared data among a large, possibly unbounded, number of clients despite crashes, asynchrony and concurrency?

An increasing number of data serving platform providers such as Amazon and Yahoo! have recognized the need for transactional access to shared data in addition to atomic read/write. It is possible to construct strong coordination primitives enabling transactional access just from atomic read/write operations. However, it is known that such constructions do not scale since the space- and communication overhead is proportional in the number of clients. Furthermore, leaving it to the developer to directly deal with the intricacies of implementing strong data-sharing primitives, such as read-modify-write, from weaker ones may result in inefficient and/or error-prone implementations.

As a consequence a number of data serving platforms such as DynamoDB, PNUTS and cloud data-bases such as Couchbase, MongoDB, Redis etc. have started including in their APIs coordination abstractions stronger than read/write. The most powerful among such primitives is Compare-And-Swap, CAS, for it enables implementing any shared functionality in a non-blocking manner, i.e. without using locks.

Intuitively, CAS updates a storage location only if the current value of that location is as expected, where the expected value is supplied to CAS along with the new value. Typically, CAS is used for optimistic concurrency control as follows: (1) a storage location x is read into a local variable v, then (2) based on the value of v some local computation is done that changes v and then (3) x is updated with v via CAS. If x didn't change, then x takes the new value v. Else x remains unchanged. In that case steps (1)-(3) are repeated. Used this way, CAS enables transactional access to shared data.

Relevant Reference:

[PODC02] Gregory Chockler and Dahlia Malkhi. 2002, "**Active disk paxos** with infinitely many processes", In Proceedings of the twenty-first annual symposium on Principles of distributed computing (PODC '02).

It is an object of the present invention to improve and further develop a method for building a concurrent data structure in the form of a ranked register, an according ranked register, a method for building a CAS object, an according CAS object, a method for providing a CAS object for transactional access to shared data within a storage by multiple clients, an according CAS object for transactional access to shared data within a storage by multiple clients and a method for providing transactional access to shared data within a storage by multiple clients for allowing a very reliable access of shared data within a storage by multiple clients.

In accordance with the invention, the afore mentioned object is accomplished by a method comprising the features of claim 1, by a concurrent data structure comprising the features of claim 7, by a method comprising the features of claim 8, by a CAS object comprising the features of claim 15, by a method for providing a CAS object comprising the features of claim 16, by a CAS object comprising the features of claim 17 and by a method for providing transactional access comprising the features of claim 18.

According to claim 1 the method is characterized in that building of the RR is based on a Compare-And-Swap, CAS, functionality for storing increasing versions of values within data objects for ensuring that newer values are never overwritten with older ones.

According to claim 7 the concurrent data structure is characterized in that the RR is based on a Compare-And-Swap, CAS, functionality for storing increasing versions of values within data objects for ensuring that newer values are never overwritten with older ones.

According to the invention it has been recognized that it is possible to provide a very reliable access of shared data within a storage by multiple clients when using CAS functionality within the construction of a ranked register. Contrary to the situation within [PODC02], the present invention does not require server-side code. In contrast, CAS supports only “update if nothing changed” functionality that tests for equality only. Within the inventive method and ranked register a fault-tolerant ranked register is constructed from fault-prone CAS objects.

Within a preferred embodiment each base object or data object holds a read rank, rR, changed only by rr-read, as well as a value, val, and a write rank, wR, changed only by rr-write, wherein the designation CAS_r and CAS_w is used to distinguish CAS invocations by rr-read and rr-write, respectively.

Further preferred, an rr-read operation R with rank r attempts to change the read rank of a majority of base objects or data objects to r, wherein at the same time, R accumulates the values returned.

Within a further preferred embodiment R tries to update a data object only if the current rank rR is lower than r, ensuring that ranks are monotonically increasing.

Further preferred, once a majority of base objects or data objects hold a rank, either rW or rR, greater or equal to r, R returns the rank-value pair with the highest write rank wR among the values or accumulated values.

Further preferred, an rr-write operation W with rank-value pair (r, v) attempts to update a majority of data objects with (r, v), wherein W tries to update a data object x only if the current ranks, both rR and wR, of x are not higher than r, wherein if W commits, then it has successfully updated a majority of data objects with (r, v), else, if W is not able to update x and some of the ranks, either rR or wR, is higher than r, then W aborts.

With the method for building a CAS object according to claim 8 it is possible to construct an obstruction-free CAS object from a ranked register.

Within a preferred embodiment a client picks a rank r, and first fetches the current value V using rr-read(r); next, it checks if V.val equals an expected value exp, and if yes, it attempts to update V with (r, new) using rr-write, else, if V.val differs from exp, the client attempts to update V with (r, v), i.e. only the rank is changed.

Within a further preferred embodiment, if the operation aborts, the steps are repeated with a higher rank.

Within a further preferred embodiment, the operation returns when rr-write commits.

Within a further preferred embodiment, for increasing the chances of solo-runs of clients, a concurrency management is used.

Within a further preferred embodiment, the concurrency management comprises exponential back-off or leader election, where all clients throw their CAS operations into a global set, and a particular client, i.e. the leader, linearizes all operations in the set.

Finally, within a further preferred embodiment, an extended CAS object is implemented, that aborts whenever the underlying rr-write operation aborts, wherein contention resolution would then be handled by an invoking application.

Within claim 16 is provided a method for providing a CAS object for transactional access to shared data within a storage by multiple clients. According to the invention this CAS object is a CAS object according to claim 15 which is based on a ranked register according to claim 7. As a result, the crash-tolerant CAS object is provided from a set of crash-prone CAS objects in the form of the storage units while supporting an unbounded number of crash-prone clients with constant space.

According to claim 18 a method for providing transactional access to shared data within a storage by multiple clients is provided. Under consideration of a storage with multiple storage units a Compare-And-Swap API is build on top of an entirety of at least some of said storage units.

Within a preferred embodiment the ranked register is a ranked register according to claim 7.

Within a further preferred embodiment the storage units can be storage clouds or storage servers which can be provided within a data center.

On the basis of the present invention ranked registers can be leveraged in the construction of fault-tolerant CAS objects from fault-prone CAS objects. Further, a fault-tolerant ranked register can be constructed from fault-prone CAS objects. Moreover, a CAS object can be constructed from a ranked register.

Important aspects of the invention together with embodiments are summarized as follows:

1) Robust transactional access to shared data via fault-tolerant CAS objects.
2) Our solution does not require deploying and running any server code.
3) Our construction of CAS from a set of fault-prone CAS is:
    a. Space efficient
    b. Non-blocking (i.e. obstruction-free)
    c. Optimally resilient
    d. Exhibits low-latency in contention-free runs.

The present invention builds entirely on the specification of CAS and does not require running customized server-side code as in [PODC02]. Our method leverages existing APIs without requiring any change in the server.

There are several ways how to design and further develop the teaching of the present invention in an advantageous way. To this end, it is to be referred to the patent claims subordinate to patent claim 1, patent claim 8 and patent claim 18 on the one hand and to the following explanation of preferred examples of embodiments of the invention, illustrated by the drawing on the other hand. In connection with the explanation of the preferred embodiments of the invention by the aid of the drawing, generally preferred embodiments and further developments of the teaching will be explained. In the drawing

Fig. 1 is illustrating a system with examples of cloud/NoSQL data-bases offering CAS API,

Fig. 2 is illustrating architectural building blocks and representation of their constructions (a) and (b),

Fig. 3 is illustrating the construction of an obstruction-free CAS object from a ranked register according to (a) of Fig. 2 and

Fig. 4 is illustrating the construction of a fault-tolerant ranked register from fault-prone CAS objects according to (b) of Fig. 2.

According to embodiments of the present invention a method and a system for transactional data-sharing in a multi-cloud or multi-storage setting can be provided. Embodiments of the present invention leverage the Compare-And-Swap API provided by existing data-serving platforms such as Yahoo! PNUTS, Amazon DynamoDB, Couchbase etc. to build a Compare-And-Swap API on top of multiple such storage clouds. The present construction tolerates the crash of up to a minority of storage clouds, i.e. optimal resilience, and supports arbitrary many clients with constant space. Unlike [PODC02], the present solution does not require server-side code.

Embodiments of the present invention describe a method and system of implementing a crash-tolerant CAS object from a set of crash-prone CAS objects while supporting an unbounded number of crash-prone clients with constant space. Specifically, for each high-level CAS object – client view –, a single low-level CAS object is required per server, see Fig. 1. Concretely, each CAS object like for example DynamoDB, Couchbase or MongoDB comprises the objects x_1, x_2, … which are accessible for the clients from their CAS API – from above within Fig. 1. The software for using the access functionality could only be provided at the client. Such a software can contact all single clouds or storages.

The present invention is related to Active Disk Paxos [PODC02], which introduces the abstraction of the Ranked Register, RR, to capture the safety property of Consensus – safety here means decision on a single value. The authors of [PODC02] present a crash-tolerant construction of a RR from a set of crash-prone Read-Modify-Write, RMW, objects. Unlike CAS, that is a generic, reusable

functionality readily available in cloud data stores, e.g. DynamoDB, Couchbase, MongoDB etc., a RMW object requires customized server-side code, which is quite different from our setting, see Fig. 1, in which individual storages have a well-defined API.

A correct construction of a RR requires that servers store monotonically increasing versions, which ensures that newer values are never overwritten with older ones. This property is enforced in [PODC02] by specialized RMW functionality executed in the server that compares versions to determine the highest one. In contrast, CAS supports only "update if nothing changed" functionality that tests for equality only, turning the construction of a RR into a challenging task.

In embodiments of the present invention a two-step approach is taken, see Fig. 2 (a) + (b) for an overview:

(a) We present for the first time an obstruction-free construction of a CAS object from a RR.
(b) We present for the first time a wait-free construction of a fault-tolerant RR from a set of fault-prone CAS objects, illustrated in Fig. 3.

Together (a) + (b) yield the first obstruction-free construction of CAS from a set of crash-prone or fault-prone CASs.

It is important to note that the results (a) and (b) are interesting in their own right. For instance, in combination with [PODC02], the result (b) yields fault-tolerant constructions of arbitrary atomic objects.

From [PODC02] is obtainable a detailed description of a construction of a ranked register. This explanation and further details and facts which are important for the present invention are explicitly included in this application by reference. In the following a summary of a ranked register and CAS functionality is provided.

Ranked Register, RR:

A ranked register is a concurrent data structure with two methods **rr-read** and **rr-write**. The rr-read method takes a rank, i.e. monotonically increasing timestamp, as argument and returns a rank-value pair. The rr-write method takes a rank-value pair as argument and returns either commit – we say it commits – or abort – we say it aborts. A ranked register needs to satisfy the following specification as defined in [PODC02]:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation. Additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1. Then R returns (r',v') where r' >= r_1 (we say R sees W).
Intuitively, safety demands that a rr-read operation doesn't miss any rr-write with a lower rank than itself.

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns.
Intuitively, non-triviality stipulates that rr-write operations are allowed to abort only because of concurrency. Note that the spec does not rule out that crashed clients cause aborts, but any sensible implementation needs to prevent such a scenario.

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns.
This is the definition of wait-freedom, the strongest possible liveness property.

Compare-And-Swap,CAS:

Definition – Compare-and-swap Sequential Specification. A compare-and-swap object C supports a single operation called CAS and stores a scalar value over some domain V. The operation CAS(exp,new), for exp, new $\in$ V, induces the following state transition of the compare-and-swap object. If C's value is exp, C's

value is changed to new; otherwise, C's value remains unchanged. In any case, the operation returns C's old value.

Construction of CAS object from a single Ranked Register:

The idea is illustrated in Fig. 3, left. V is a tuple: V = (rank, val). A client picks a rank r, and first fetches the current value V using rr-read(r). Next, it checks if V.val equals the expected value exp, and if yes, it attempts to update V with (r, new) using rr-write. Else, if V.val differs from exp, the client attempts to update V with (r, v), i.e. only the rank is changed. If the operation aborts, the steps are repeated with a higher rank. The operation returns when rr-write commits. The construction is obstruction-free, i.e. termination is guaranteed when the client invoking CAS eventually runs solo. To increase the chances of solo-runs, we use exponential back-off.

A viable alternative to exponential back-off is leader election, where all clients throw their CAS operations into a global set, and a particular client, i.e. the leader, linearizes all operations in the set. Another alternative is to implement an extended CAS object that aborts whenever the underlying rr-write operation aborts. Contention resolution would then be handled by the invoking application.

We now argue that the construction in Fig. 3 is correct. Correctness here means that our construction is atomic and that it satisfies the sequential specification of CAS, Definition. Furthermore, we argue that it is obstruction-free.

If the code is run sequentially, it is easy to see that our construction is obstruction-free and that it abides the sequential specification of CAS, Definition. Now, assume that two clients c_i and c_j run concurrently, and that both return, committing with rank r_i and r_j respectively. Let CAS_i and CAS_j be the instances invoked by c_i and c_j respectively. We will show that if $r_i < r_j$ then the execution is equivalent to a sequential execution in which CAS_i is sequenced before CAS_j, where $i,j \in \{1,2\}$ and $i \neq j$. Without loss of generality, let i = 1 and j = 2.

Since c_1 commits with r_1 and r_1 < r_2, by the Safety property of the ranked register, when c_2 invokes rr-read(r_2), it sees the change committed by c_1. Therefore, c_2 does the comparison based on the state left by c_1. Thus, this is equivalent to a sequential permutation in which CAS_1 is sequenced before CAS_2.

Furthermore, if a client eventually runs solo, then it picks a rank which is higher than any other rank used so far, in which case the rr-write operation commits and the client returns from the CAS invocation.

Finally, the READ operation on the right side of Fig. 3 is added for completeness, since CAS is typically used as part of a first-read-then-write pattern. Since CAS is executed with equal parameters, here the initial value, nothing is changed.

Construction of a fault-tolerant Ranked Register from fault-prone CAS objects:

The idea is illustrated in Fig. 4. Each base object holds a read rank (rR), changed only by rr-read, as well as a value (val) and write rank (wR), changed only by rr-write. We write CAS_r/CAS_w to distinguish CAS invocations by rr-read / rr-write respectively.

An rr-read operation R with rank r attempts to change the read rank of a majority of base objects to r. At the same time, R accumulates the values returned. R tries to update an object only if the current rank rR is lower than r, ensuring that ranks are monotonically increasing. Once a majority of base objects hold a rank, either rW or rR, greater or equal to r, R returns the rank-value pair with the highest write rank wR among the accumulated values.

An rr-write operation W with rank-value pair (r,v) attempts to update a majority of objects with (r,v). W tries to update an object x only if the current ranks, both rR and wR, of x are not higher than r. If W commits, then it has successfully updated

a majority of objects with (r,v). Else, if W is not able to update x and some of the ranks, either rR or wR, is higher than r, then W aborts.

We now argue that the construction is correct.

1) Non-triviality holds because an rr-write operation aborts only if it sees a higher-ranked operation in one of the base objects it accesses.

2) Liveness is satisfied because each operation op = rr-read(r) / rr-write((r,v)) eventually returns. The explanation is that for any rank r, there are only a finite number of operations that are invoked with a lower rank, and thus op re-invokes CAS on a base object x only a finite number of times. Eventually, either x is successfully updated by op or x holds a rank higher than r. In any case, op does not block on x.

3) Safety: Let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1. We will argue that R returns (r',v') where r' >= r_1, thus satisfying Safety. Let O_w and O_r be the sets of based objects accessed by W and R respectively. Let k $\in$ O_w $\cap$ O_r. Let CAS_r and CAS_w be the last operation on x_k by R and W respectively. If CAS_r is sequenced before CAS_w, then CAS_w returns a rank higher than r_1 (at least r_2). Since CAS_w is invoked with rank at most r_1, the value returned by CAS_w does not match the expected value and W would abort, contradicting our assumption. Otherwise, if CAS_w is sequenced before CAS_r, CAS_r would assign to x[k] a value with rank r_1 or higher and consequently R would return a rank-value pair with rank r' >= r_1, as needed.

Intra-datacenter coordination:

In addition to the multi-cloud setting, our method is useful in a single data-center setting with multiple storage units as well. For instance, it enables building data-centric coordination on top of fault-prone storage servers within a data center. Since all our solution needs is a generic CAS interface, we do not require full

customization of the controller logic as in [PODC02] and the interface can be reused by several applications.

Many modifications and other embodiments of the invention set forth herein will come to mind the one skilled in the art to which the invention pertains having the benefit of the teachings presented in the foregoing description and the associated drawings. Therefore, it is to be understood that the invention is not to be limited to the specific embodiments disclosed and that modifications and other embodiments are intended to be included within the scope of the appended claims. Although specific terms are employed herein, they are used in a generic and descriptive sense only and not for purposes of limitation.

Claims

1. A method for building a concurrent data structure in the form of a ranked register, RR, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns,

characterized in that building of the RR is based on a Compare-And-Swap, CAS, functionality for storing increasing versions of values within data objects for ensuring that newer values are never overwritten with older ones.

2. A method according to claim 1, wherein each base object or data object holds a read rank, rR, changed only by rr-read, as well as a value, val, and a write rank, wR, changed only by rr-write, wherein the designation CAS_r and CAS_w is used to distinguish CAS invocations by rr-read and rr-write, respectively.

3. A method according to claim 2, wherein an rr-read operation R with rank r attempts to change the read rank of a majority of base objects or data objects to r, wherein at the same time, R accumulates the values returned.

4. A method according to claim 3, wherein R tries to update a data object only if the current rank rR is lower than r, ensuring that ranks are monotonically increasing.

5. A method according to claim 4, wherein, once a majority of base objects or data objects hold a rank, either rW or rR, greater or equal to r, R returns the rank-value pair with the highest write rank wR among the values or accumulated values.

6. A method according to claim 5, wherein an rr-write operation W with rank-value pair (r, v) attempts to update a majority of data objects with (r, v), wherein W tries to update a data object x only if the current ranks, both rR and wR, of x are not higher than r, wherein if W commits, then it has successfully updated a majority of data objects with (r, v), else, if W is not able to update x and some of the ranks, either rR or wR, is higher than r, then W aborts.

7. A concurrent data structure in the form of a ranked register, RR, preferably built by the method according to any one of claims 1 to 6, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns,

c h a r a c t e r i z e d in that the RR is based on a Compare-And-Swap, CAS, functionality for storing increasing versions of values within data objects for ensuring that newer values are never overwritten with older ones.

8. A method for building a Compare-And-Swap, CAS, object, wherein a concurrent data structure in the form of a ranked register, RR, is used, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns,

wherein increasing versions of values within data objects are stored for ensuring that newer values are never overwritten with older ones.

9. A method according to claim 8, wherein a client picks a rank r, and first fetches the current value V using rr-read(r); next, it checks if V.val equals an expected value exp, and if yes, it attempts to update V with (r, new) using rr-write, else, if V.val differs from exp, the client attempts to update V with (r, v), i.e. only the rank is changed.

10. A method according to claim 9, wherein, if the operation aborts, the steps are repeated with a higher rank.

11. A method according to claim 9 or 10, wherein the operation returns when rr-write commits.

12. A method according to one of claims 8 to 11, wherein for increasing the chances of solo-runs of clients, a concurrency management is used.

13. A method according to claim 12, wherein the concurrency management comprises exponential back-off or leader election, where all clients throw their CAS operations into a global set, and a particular client, i.e. the leader, linearizes all operations in the set.

14. A method according to one of claims 8 to 13, wherein an extended CAS object is implemented, that aborts whenever the underlying rr-write operation aborts, wherein contention resolution would then be handled by an invoking application.

15. A Compare-And-Swap, CAS, object, preferably built by the method according to any one of claims 8 to 14, based on a concurrent data structure in the form of a ranked register, RR, wherein within the RR two operations rr-read and rr-write are defined, wherein the rr-read operation is provided for taking a rank as argument and for returning a rank-value pair, wherein the rr-write operation is provided for taking a rank-value pair as argument and for returning either commit or abort and wherein the RR satisfies the following specification:

Safety: Every rr-read operation returns a rank-value pair that was written in some rr-write invocation; additionally, let W = rr-write(r_1, v) be a rr-write operation that commits and let R = rr-read(r_2), such that r_2 > r_1; then R returns (r',v') where r' >= r_1 (we say R sees W), wherein v and v' are values and wherein r', r_2 and r_1 are ranks,

Non-triviality: If a rr-write operation W invoked with rank r_1 aborts, then there exists a rr-read or rr-write operation with rank r_2 > r_1 which is invoked before W returns, and

Liveness: If an operation rr-read or rr-write is invoked by a non-faulty process, then it eventually returns,

wherein increasing versions of values within data objects are stored for ensuring that newer values are never overwritten with older ones.

16. A method for providing a Compare-And-Swap, CAS, object for transactional access to shared data within a storage by multiple clients,
wherein the storage comprises multiple storage units, each of which being accessible via a separate Application Programming Interface, API, each of which comprising a Compare-And-Swap, CAS, primitive,
wherein an entirety of multiple such storage units is assigned one further API comprising a further Compare-And-Swap, CAS, primitive for access by the clients and
wherein as the CAS object a CAS object according to claim 15 based on a ranked register according to claim 7 is used,
so that the CAS object comprises multiple CAS objects in the form of said multiple storage units with their APIs.

17. A Compare-And-Swap, CAS, object for transactional access to shared data within a storage by multiple clients,
wherein the storage comprises multiple storage units, each of which being accessible via a separate Application Programming Interface, API, each of which comprising a Compare-And-Swap, CAS, primitive,
wherein an entirety of multiple such storage units is assigned one further API comprising a further Compare-And-Swap, CAS, primitive for access by the clients and
wherein the CAS object is a CAS object according to claim 15 based on a ranked register according to claim 7,
so that the CAS object comprises multiple CAS objects in the form of said multiple storage units with their APIs.

18. A method for providing transactional access to shared data within a storage by multiple clients,
wherein the storage comprises multiple storage units, each of which being accessible via a separate Application Programming Interface, API, each of which comprising a Compare-And-Swap, CAS, primitive,
wherein an entirety of multiple such storage units is assigned one further API comprising a further Compare-And-Swap, CAS, primitive for access by the clients and

wherein as a functional connection between the entirety of multiple such storage units and the further API a ranked register is used.

19. A method according to claim 18, wherein the ranked register is a ranked register according to claim 7.

20. A method according to one of claims 16 to 19, wherein the storage units are storage clouds or storage servers.

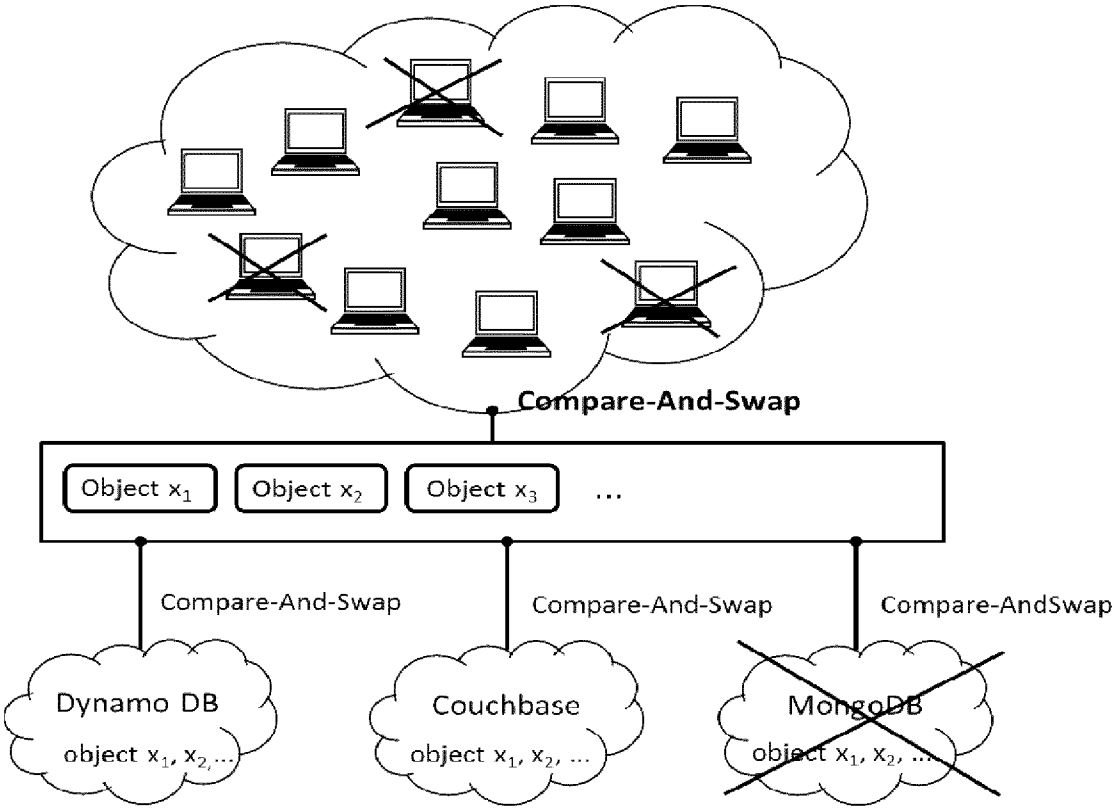
Compare-And-Swap
Object x_1
Object x_2
Object x_3
...
Compare-And-Swap
Compare-And-Swap
Compare-AndSwap
Dynamo DB
object x_1, x_2, ...
Couchbase
object x_1, x_2, ...
MongoDB
object x_1, x_2, ...

Fig. 1

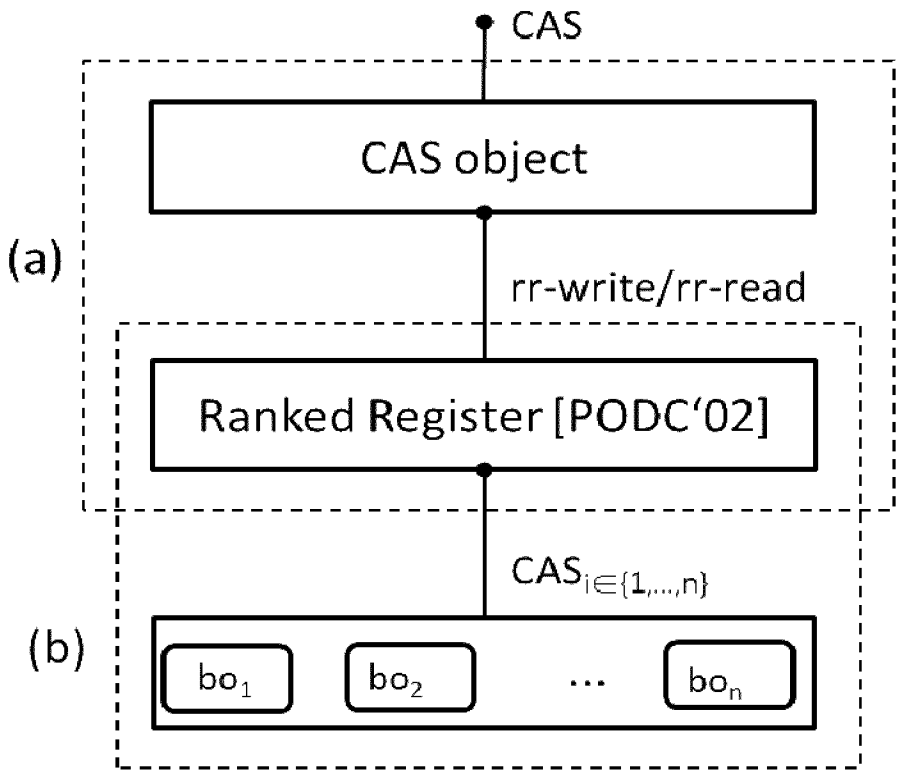
CAS
CAS object
(a)
rr-write/rr-read
Ranked Register [PODC'02]
CAS$_{i\in\{1,\ldots,n\}}$
(b)
bo$_1$
bo$_2$
...
bo$_n$

Fig. 2

```
1: Definitions
2:    RVals: (Ranks × Vals) ∪ {(r_0, ⊥)} with selectors rank and val
3: Base-objects
4:    x ∈ RVals is a ranked register, initially (r_0, ⊥)
5: State
6:    r: Ranks initially r_0
7:    old: Vals initially ⊥
```

```
8:  operation CAS(exp, new)
9:     while true do
10:       r ← chooseRank(r)
11:       V ← x.rr-read(r)
12:       old ← V.val
13:       if V.val = exp then
14:          V.val ← new
15:       V.rank ← r
16:       if x.rr-write(V) = commit then
17:          return old
18:       else
19:          wait(a_while)  //exp.back-off
```

```
20: operation READ()
21:    v ← CAS(⊥, ⊥)
22:    return v
```

Fig. 3

1: **Definitions**
2: X: $(Ranks \times Ranks \times Vals) \cup \{v_0\}$ with selectors rR and wR and val
3: $v_0 : (r_0, r_0, \perp)$

4: **Base-objects**
5: $\forall i \in S$, $x_i \in X$ is a base-object on storage-node i, initially v_0

6: **State**
7: $x[k]$: X, for every $k \in S$, initially v_0
8: *new*: X, initially v_0
9: *retval*: { *abort*, *commit* }, initially *commit*
10: $R : 2^{\mathbb{N}_0}$, initially $\emptyset$

11: **operation** *rr-write*$((r, v))$
12: $new.wR \leftarrow r$
13: $new.val \leftarrow v$
14: in parallel for all $k \in S$ **do**
15: **invoke** $x_k.CAS_w(v_0, new)$
16: **wait** until $|R| \geq \lceil (|S|+1)/2 \rceil$
17: return *retval*
18: **operation** *rr-read*(r)
19: $new.rR \leftarrow r$
20: in parallel for all $k \in S$ **do**
21: **invoke** $x_k.CAS_r(v_0, new)$
22: **wait** until $|R| \geq \lceil (|S|+1)/2 \rceil$
23: $k \leftarrow \text{ARGMAX}_{k \in R}\{x[k].wR\}$
24: **return** $(x[k].wR, x[k].val)$
25: **upon** completion $x_k.CAS_w(u, *)$ returning v
26: **if** $u \neq v \wedge r \geq v.rR \wedge r > v.wR$ **then**
27: $new.rR \leftarrow v.rR$
28: **invoke** $x_k.CAS_w(v, new)$
29: **else**
30: $R \leftarrow R \cup \{k\}$
31: **if** $u \neq v$ **then**
32: *retval* $\leftarrow$ *abort*
33: **upon** completion $x_k.CAS_r(u, *)$ returning v
34: $x[k] \leftarrow v$
35: **if** $u \neq v \wedge r > v.rR \wedge r > v.wR$ **then**
36: $x[k].rR \leftarrow r$
37: **invoke** $x_k.CAS_r(v, x[k])$
38: **else** $R \leftarrow R \cup \{k\}$

Fig. 4

INTERNATIONAL SEARCH REPORT

International application No
PCT/EP2013/071489

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/52
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	GREGORY CHOCKLER ET AL: "Active Disk Paxos with infinitely many processes", DISTRIBUTED COMPUTING, SPRINGER, BERLIN, DE, vol. 18, no. 1, July 2005 (2005-07), pages 73-84, XP019347301, ISSN: 1432-0452, DOI: 10.1007/S00446-005-0123-X paragraph [0004]	1-20

-/--

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
3 February 2014	10/02/2014

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Wiltink, Jan Gerhard

2

Form PCT/ISA/210 (second sheet) (April 2005)

INTERNATIONAL SEARCH REPORT

International application No
PCT/EP2013/071489

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	GREGORY CHOCKLER ET AL: "Active disk paxos with infinitely many processes", PROCEEDINGS OF THE TWENTY-FIRST ANNUAL SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 21 July 2002 (2002-07-21), pages 78-87, XP55099740, DOI: 10.1145/571825.571837 ISBN: 978-1-58-113485-8 paragraph [04.1] -----	1-20

2