2009 29th IEEE International Conference on Distributed Computing Systems

On Optimal Concurrency Control for Optimistic Replication

Weihan Wang and Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada

Abstract

Concurrency control is a core component in optimistic replication systems. To detect concurrent updates, the system associates each replicated object with metadata, such as, version vectors or causal graphs exchanged on synchronization opportunities. However, the size of such metadata increases at least linearly with the number of active sites. With recent trends in cloud computing, multi-regional collaboration, and mobile networks, the number of sites within a single replication system becomes very large. This imposes substantial overhead in communication and computation on every site.

In this paper, we first present three version vector implementations that significantly reduce the cost of vector exchange by incrementally transferring vector elements. Basic rotating vectors *(*BRV*) support systems providing no conflict reconciliation, whereas* conflict rotating vectors *(*CRV*) extend* BRV *to overcome this limitation.* Skip rotating vectors *(*SRV*) based on* CRV *further reduce data transmission. We show that both* BRV *and* SRV *are optimal implementations of version vectors, which, in turn, have minimal storage complexity among all known concurrency control schemes for state-transfer systems. We then present a causal graph exchange algorithm for operation-transfer systems with optimal communication overhead. All these algorithms adopt* network pipelining *to reduce running time.*

1. Introduction and background

Optimistic replication is a key technology to achieve high availability, scalability, and performance for data replication systems. Unlike pessimistic replication requiring replica updates to be strictly synchronized, optimistic replication assumes that conflicts happen rarely and allows updates on different sites to occur simultaneously without a priori synchronization [1].

Optimistic replication has a wide spectrum of applications. It enables cloud computing and storage systems like Amazon S3, Dynamo [2] and OceanStore [3] to achieve unprecedented large scales. Optimistically replicated databases, such as, Bayou [4] and Couch DB (http://couchdb.apache.org) have special advantages in supporting mobile computing and efficient replication. Dozens of file systems, including Coda [5], Ficus [6], and Microsoft WinFS [7] employ optimistic replication for high availability and fault tolerance. Distributed revision control systems, such as, Mercurial (http://selenic.com/mercurial) and Pastwatch [8] allow developers to work on the same project in relative isolation. Computer supported cooperative work (CSCW) optimistically replicates users' workspace across geographical regions [9]. Recently, the technique has also been investigated to facilitate information sharing in developing countries [10].

A core component of any optimistic replication system is the concurrency control mechanism to detect *and* resolve conflicts due to concurrent updates. Conflict detection techniques can be categorized into three classes: *syntactic* detection records causal relations of updates, flagging all causally independent updates on the same data item as conflicts; *semantic* detection depends on application semantics and raises a conflict if updates are semantically inconsistent; *semantic-over-syntactic* detection filters out syntactic conflicts that are semantically consistent.

Conflict resolution can be either manual or automatic. *Manual* resolution excludes two conflicting replicas from the system once detected, waiting for human operators to solve the conflict manually. This approach is common in revision control systems. *Automatic* resolution merges concurrent updates and generates a new version without excluding replicas from the system. The algorithms presented in this paper are applicable to all types of replication systems except those detecting conflicts only semantically. Readers are referred to Saito and Shapiro [1] for a survey and taxonomy of optimistic replication.

Systems based on syntactic or semantic-over-syntactic conflict detection attach certain metadata, such as, version vectors [11], hash histories [12], or revision graphs [8] to each data item replica. The metadata is updated and examined at runtime to determine conflicts. Because sites must exchange metadata when synchronizing replicas, an important issue is the large amount of bytes transmitted for metadata exchange and the cost of processing them. Traditionally, the entire metadata is sent. As we shall see shortly, the known minimal size of the per-replica metadata is proportional to the number of sites in the system. Even for a system of moderate size, transmitting the entire metadata imposes substantial overhead on every site, if the system hosts many objects or sites synchronize frequently.

This scalability issue becomes more acute with recent,

1063-6927/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDCS.2009.71

sharp increases in the number of sites a replicated system accommodates. In cloud computing, thousands or more loosely coupled machines are installed in data centers to host a vast volume of replicated data [2, 3]. A multi-regional collaboration system may span many institutions and users [8]. A participatory, replicated data store for DTN or general wireless networks may house numerous small objects spreading over a large number of mobile devices [10]. In this last scenario, power constraints on mobile devices also call for reducing network communication as much as possible.

In this paper, our key idea for minimizing communication and processing overheads in metadata exchange is to transmit only the metadata differences across replicas, instead of the entire metadata. Towards this, we propose three vector implementations of increasing complexity to approach optimality for *state-transfer* systems, and an optimal synchronization algorithm that sends only the difference between causal graphs in *operation-transfer* systems.

We first focus on *state-transfer* systems, where the entire object is overwritten during synchronization. Version vectors are the best known approach for such systems in terms of storage space requirements. However, the conventional vector synchronization algorithm is not optimal. We introduce basic rotating vectors (BRV) by augmenting version vectors with a total order of elements; the ith element becomes the first in the order when an update occurs on site i. During synchronization, elements are sent in order until the receiver finds an element value less or equal than its own.

While BRV is an optimal vector implementation, it only supports manual conflict resolution as automatic resolution distorts element orders. Conflict rotating vectors (CRV) use an additional bit on each element to mark distorted parts of the order. Although ensuring correctness, CRV is suboptimal due to transmission of elements that the receiver already knows about. In systems where conflict rates are rare, CRV incurs insignificant redundant transfer. However, overheads increase with the conflict rate.

To address this problem, we propose skip rotating vectors (SRV) by adding a second bit to each element. These bits split a vector into "segments" in order to avoid sending segments known by the receiver. In addition to transmitting different elements between two vectors, SRV also sends $O(1)$ information about each skipped segment. We proceed to show that this is unavoidable for any vector synchronization algorithm, and SRV is in fact optimal.

In contrast to state transfer, *operation transfer* logs and exchanges update operations for each object. Operations can be in different forms, such as, database statements or delta between revisions. A causal graph or equivalent is maintained for each replica to capture inter-operation relationships. To address communication overheads in causal graph synchronization, we present an optimal algorithm that sends only the difference between graphs. In all the proposed algorithms, we use a technique called *network pipelining* to reduce algorithm running time. This is helpful in high-latency networks and bulk transfer of small data items.

The rest of the paper is organized as follows. § 2 introduces the system model and version vectors. § 3 presents BRV and CRV along with their complexity analysis. § 4 discusses SRV and its complexities. § 5 gives the lower bound of the vector synchronization problem. § 6 presents the causal graph synchronization algorithm for operation transfer. § 7 summarizes related work and § 8 concludes the paper.

2. Preliminaries

2.1. System model

Without loss of generality, we model an optimistic replication system as follows. Each participating site stores at most one *replica* for each replicated *object* locally. An object can be as large as a full-fledged relational database [13], or as small as a single file or log entry [7]. An object is created on one site and can be updated on all the sites having a replica of that object. A site may replicate an object to *any* other site through opportunistic communication. When propagating a replica to the other site which hosts a replica of the same object – a process that we call *replica synchronization*, a *syntactic conflict* or simply *conflict* is detected iff an update on one of the replicas is *concurrent* (i.e., causally independent) with an update on the other. Once detected, sites resolve conflicts either automatically or manually. Synchronizing conflicting replicas using automatic conflict resolution is called *reconciliation*. Manual resolution provides no reconciliation since conflicting replicas are always excluded from the system.

Replicas are called *consistent* iff their states are identical or merely semantically equivalent. Two replicas become consistent after synchronization. The system achieves either eventual consistency, i.e., all replicas of an object become consistent in finite time after the last update on the object, or some consistency model stronger than eventual consistency [1, §3.6].

2.2. Version vector algorithm and its optimality

To detect syntactic conflicts, every replica is associated with metadata maintained by the site hosting the replica. Version vectors [11], a special use of vector timestamps [14], are one such type of metadata. In this subsection, we review version vectors and show why they are the technique of choice over all known schemes for state-transfer systems. In operation-transfer systems however, using vectors is insufficient. We shall come back to operation transfer and its technique of choice, causal graphs, in § 6.

Definition A *version vector* v is a set of n pairs, where n is the number of sites. The ith pair is in the form of $(i, v[i])$, where i is a site name and $v[i]$ is the number of updates made on the site. We call $(i, v[i])$ the ith *element*, and $v[i]$ the ith *value* of v. Sites are exemplified using letters and vectors are written as $\langle A{:}2, B{:}1, C{:}3\rangle$, where $v[A] = 2$, $v[B] = 1$, and $v[C] = 3$.

A replica associated with vector a causally precedes another replica with vector b, denoted as $a \rightarrowtail b$, iff $a[i] \leq b[i]$ for all i and there exists a j such that $a[j] < b[j]$. If none of $a = b$, $a \rightarrowtail b$, and $b \rightarrowtail a$ holds, the two replicas are concurrent with each other, denoted as $a \parallel b$. Non-concurrent cases are symbolized as $a \nparallel b$. When synchronizing replicas, their vectors are also synchronized so that the ith element of the resulting vector is set to be $\max(a[i], b[i])$, for all i. After synchronizing with a concurrent vector (during conflict reconciliation), the site i increments the ith value to ensure compatibility with old versions [11, §C]. We consider this step as a separate update rather than a part of synchronization.

Is version vector optimal for concurrency control? Charron-Bost [15] has shown that an n dimensional data structure is required to accurately capture causality using real numbers in a distributed system of n processes. This means that the size of metadata for each replica is at least $O(n)$. To be precise, $n \log m$ bits are required to construct the data structure, where m is the number of updates on each site (assumed to be constant). This is exactly the size of version vectors. Baldoni and Melideo [16] further show that the minimum number of bits to encode causality could be less than $n \log m$. However, both papers suggest that algorithms better than version vectors are unlikely to exist. Other work has proposed trading accuracy for space [17, 18], or reducing the size by removing inactive sites from a vector [19, 20]. The latter approach is equivalent to the original version vector plus a distributed membership manager.

Observation 2.1: Among all known solutions, version vectors and variants have the minimal storage complexity for accurate conflict detection.

There are other schemes besides version vectors. Unfortunately, none can accurately detect conflicts with less than $O(n)$ bytes. For example, schemes based on predecessor sets attach to each replica a set of identifiers of previously executed operations [1, §4.2]. Although the size requirement of these approaches seems independent of n, each active site corresponds to at least one entry in the set, making it even bigger than a version vector. Log truncation does not help, because the vector is a compact representation of the set; any size smaller than the vector would cause false positives on detecting conflicts. Excessive truncation is equivalent to removing active sites from a vector. Nevertheless, set-based approaches are useful for operation transfer and other purposes (e.g., [12]).

Algorithm 1 COMPARE(a, b): compare BRV a and b

1: $(l_a, u_a) \leftarrow \lfloor a \rfloor$, $(l_b, u_b) \leftarrow \lfloor b \rfloor$
2: **if** $u_a = b[l_a] \wedge a[l_b] = u_b$ **then** $a = b$
3: **else if** $u_a \leq b[l_a]$ **then** $a \rightarrowtail b$
4: **else if** $u_b \leq a[l_b]$ **then** $b \rightarrowtail a$
5: **else** $a \parallel b$
6: **end if**

Besides storage requirement, complexities in metadata synchronization and comparison are also metrics in describing the optimality of a conflict detection algorithm. Although version vectors have known minimal storage overhead, the vector synchronization algorithm may not be optimal. We now present three novel vector implementations towards optimizing synchronization. As we shall see, vector comparison has been achieved with constant space and time.

3. Basic and conflict rotating vectors

Traditionally, synchronizing two version vectors involves $O(n)$ network transmission to send one entire vector to the other site. In this section, we present two vector implementations to reduce this overhead. By sending only the different elements, the first implementation is optimal for systems without conflict reconciliation. The second implementation incurs only small overheads when conflict rates are low.

3.1. Basic rotating vectors (BRV)

We observe that the difference between two vectors consists of the elements modified after the last synchronization between the two. Therefore, we track the order of element modifications and send elements in that order, until reaching an element unmodified since the last synchronization.

Definition A *basic rotating vector* v is a version vector paired with a total order $\prec_v$ of the elements in v (or $\prec$ when understandable from the context). A BRV is written as $\langle C{:}3, A{:}2, B{:}1\rangle_{\prec}$, where the verbatim order is identical to $\prec$. We use $\lfloor v \rfloor$ and $\lceil v \rceil$ to denote the least (first) and greatest (last) elements in the order. In the above example, $\lfloor v \rfloor = (C, 3)$ and $\lceil v \rceil = (B, 1)$.

Definition Operation ROTATE$_v(p, i)$ adjusts the position of the ith element in $\prec_v$ such that if $p \neq \phi$, $(p, \cdot) \prec_p (i, \cdot) \prec_p (q, \cdot)$, where $(q, \cdot)$ is the former successor of $(p, \cdot)$ in $\prec_v$, and that if $p = \phi$, $(i, \cdot) \prec_p (q, \cdot)$, where formerly $(q, \cdot) = \lfloor v \rfloor$.

The element order is initialized arbitrarily. Whenever a site i updates the replica that vector v is associated with, in addition to incrementing $v[i]$, ROTATE$_v(\phi, i)$ is also performed, causing the ith element to "rotate" to the first position in the order (hence the name of rotating vectors).

Algorithm 1, COMPARE, compares two rotating vectors. Unlike the well known algorithm that compares all the elements, COMPARE finishes in *constant* running time. This is possible because the rotating vector remembers (through $\prec$) the site who made the latest update. This information can be used to quickly determine precedence: if $u_a \leq b[l_a]$, then b must have knowledge about all the updates that a knows about, hence a either causally precedes b or equals b. Interested readers are referred to [21, Lemma 3.4] for a detailed rationale.

The most benefit of BRV manifests when synchronizing vectors: if vector a is found through COMPARE to causally precede vector b, it shall be overwritten by b (its corresponding replica shall be overwritten, too). To do so, traditional version vectors need $O(n)$ operations to send all the elements over the network. In contrast, BRV uses Algorithm 2, SYNCB, to transmit only the different elements.

Suppose a is hosted on site A and b is hosted on site B. In SYNCB, b's elements are sent to A in the ascending order of $\prec_b$ until B receives a $HALT$ message from A (Line 8). On the receiving side, for each ith element A receives, it sets $a[i]$ equal to $b[i]$ until the latter is less or equal than the former (Line 4, 9). A also adjusts $\prec_a$ (Line 7, 8), such that at the time of algorithm termination, the least k elements in $\prec_a$ will have the same order and values as the least k elements in $\prec_b$, assuming that the $(k+1)$th element is the one the algorithm halts on.

SYNCB requires $a \not\parallel b$, meaning that it cannot handle concurrent vectors. As a result, BRV is only applicable in systems with manual conflict resolution that provide no conflict reconciliation. The following theorem depicts the correctness of SYNCB.

Theorem 3.1: $c := \text{SYNCB}_a(b)$ where $a \not\parallel b$. $c = b$ if $a \rightarrowtail b$ and $c = a$ otherwise.

Due to space constraints, readers are referred to [22] for the proof of the theorem. A technique called *network pipelining* is used by all the algorithms in this paper: instead of stop-and-wait on every element, the sender speculatively sends all elements until an asynchronous negative response is heard (Line 8, b's site). Pipelining reduces algorithm running time by $(k-1) \times rtt$ with k the number of items sent and rtt the network round-trip time. This is particularly useful when network latency or the number of data items to be sent is large. Pipelining also suppresses $(k-1)$ reply messages as they now become implicit. However, let β be the product of network bandwidth and round-trip time, pipelining results in β bytes of excess transmission after the reply is emitted.

Algorithm 2 $\text{SYNCB}_b(a)$: synchronize BRV a with b

Require: $a \not\parallel b$

On b's hosting site:

1: $cur \leftarrow \lfloor b \rfloor$ {cur is a pair}
2: **repeat**
3: send cur
4: **if** $cur = \lceil b \rceil$ **then**
5: send $HALT$; **halt**
6: **end if**
7: $cur \leftarrow cur$'s successor in $\prec_b$
8: **until** received $HALT$

On a's hosting site:

1: $prev \leftarrow \phi$
2: **repeat**
3: receive (i, u_i)
4: **if** $u_i \leq a[i]$ **then**
5: send $HALT$; **halt**
6: **else**
7: $\text{ROTATE}_a(prev, i)$
8: $prev \leftarrow i$
9: $a[i] \leftarrow u_i$
10: **end if**
11: **until** the next received data is $HALT$

3.2. Conflict rotating vectors (CRV)

We design conflict rotating vectors, CRV, in order to synchronize concurrent vectors. In fact, we can remove the "$a \not\parallel b$" requirement from SYNCB without compromising correctness (see [22] for a proof). However, correctness does not hold for subsequent SYNCB calls. Consider, for example, two concurrent vectors $\theta_1 = \langle A{:}2, B{:}1\rangle_\prec$ and $\theta_2 = \langle B{:}2, A{:}1\rangle_\prec$. The system invokes $\text{SYNCB}_{\theta_1}(\theta_2)$ generating $\theta_3 = \langle A{:}2, B{:}2\rangle_\prec$. When calling $\text{SYNCB}_{\theta_3}(\theta_1)$ afterward, the algorithm halts at the Ath element leaving $\theta_1[B]$ unsynchronized.

The problem is that from θ_1 to θ_3, $(A, 2)$ is rotated to the front but with its value unchanged. This hides elements behind $(A, 2)$ from proper synchronization even if they are new to θ_1. To solve this problem, whenever synchronizing two concurrent vectors, the algorithm tags all the modified elements to prevent them from hiding unmodified ones. When it is invoked afterward, the algorithm overlooks all tagged elements when deciding termination.

Definition A *conflict rotating vector* is a BRV paired with one *conflict bit* for each element. We denote the bit for the ith element of vector v as $v.c[i]$. A CRV is written as $\langle \overline{A}{:}2, B{:}2\rangle_\prec$: a bar is drawn above each element with the bit set.

We use conflict bits as tags. $v.c[i]$ is initially zero and is reset whenever $v[i]$ is incremented due to a replica update on site i. Algorithm 3, SYNCC, synchronizes CRVs. It is identical to SYNCB except the use of the conflict bit. First, it sets the bits for all the elements modified during reconciliation (Line 14). Second, when deciding termination, it skips all the elements with the bit set (Line 5–7). Correctness of SYNCC is shown in [22].

Algorithm 3 SYNCC$_b(a)$: synchronize CRV a with b

On b's hosting site:

Same as SYNCB except that cur becomes a triple.

On a's hosting site:

```
1: prev ← φ
2: reconcile ← a ∥ b
3: repeat
4:    receive (i, u_i, c_i)
5:    if u_i ≤ a[i] then
6:       if c_i = 1 then
7:          reconcile ← true
8:       else
9:          send HALT; halt
10:      end if
11:   else
12:      ROTATE_a(prev, i)
13:      prev ← i
14:      if reconcile then c_i ← 1 end if
15:      a[i] ← u_i; a.c[i] ← c_i
16:   end if
17: until the next received data is HALT
```

3.3. Complexities of BRV and CRV

We make three assumptions throughout the paper for complexity analysis: i) Version vectors have $O(1)$ complexity, in time and space for element lookup, insertion, and deletion, ii) The length of site names and element values are fixed. Therefore, $\log n$ and $\log m$ are treated as constants, iii) Network bandwidth and latency are also constant. Below we list various complexities of BRV and CRV.

Storage. The total order can be implemented as a doubly linked list with $O(n)$ storage complexity.

Comparison. COMPARE is $O(1)$ in space, time, and communication. To be precise, $(2 \log mn)$ bits are transferred, which is the minimum amount of information required for the vector comparison problem.

Synchronization. Let Δ be the set of elements with values in b greater than the corresponding elements in a. SYNCB$_b(a)$ is $O(1)$ in space and $O(|\Delta|)$ in both time and communication, which is obviously optimal as Δ is the smallest set of elements that has to be transmitted. SYNCC$_b(a)$ is $O(1)$ in space but $O(|\Delta| + |\Gamma|)$ in time and communication, where Γ is the set of elements with values in b less or equal than the corresponding elements in a, but still transmitted due to their conflict bits being set.

Table 1 lists the above notations. Table 2 summarizes the complexities of vector synchronization algorithms. The table also shows the worst case communication. In the next section, we attempt to eliminate the $|\Gamma|$ term in SYNCC's complexities.

	Meaning	Def. in
n	the number of sites	
m	the number of updates on each sites	
Δ	$\{i : b[i] > a[i]\}$	§ 3.3
Γ	$\{i : b[i] \leq a[i] \wedge b[i]$ is received by $a\}$	§ 3.3
γ	the number of skipped segments	§ 4.1

Table 1. Notations in the context of SYNC$^\star{}_b(a)$

	Space	Time / Comm.	Comm. upper bound (bits)
Optimal	$O(1)$	$O(\|\Delta\| + \gamma)$	–
BRV	$O(1)$	$O(\|\Delta\|)$	$n \log 2mn + 2$
CRV	$O(1)$	$O(\|\Delta\| + \|\Gamma\|)$	$n \log 4mn + 2$
SRV	$O(1)$	$O(\|\Delta\| + \gamma)$	$n \log 8mn + n \log 2n + 1$

Table 2. Complexities of vector synchronization

4. Skip rotating vectors (SRV)

BRV is optimal for systems without conflict reconciliation. Although CRV is suboptimal, it works well when conflicts are infrequent, as $|\Gamma|$ is bounded by the number of elements modified during reconciliation. Although low conflict rates are a generic assumption for optimal replication, there may be situations where high conflict rates are expected. For example, semantic-over-syntactic conflict detection may use syntactic conflicts as a trigger to more costly semantic detectors. In this case, heavily updated objects can generate numerous syntactic-only conflicts (e.g., a replicated append-only log file).

To assist discussion and formal analysis of SRV, the third vector implementation, we introduce a graphical representation inspired by Parker et al.'s partition graph [11]. The *replication graph* of an object o is a directed acyclic graph (dag) where each node represents identical replicas of o, and is optionally labeled with the names of sites hosting these replicas. Each node corresponds to a vector, which is the vector of the replicas the node represents. A node may have at most two parents. Nodes with one parent result from a single update on the parent replica; nodes with two parents are the results of conflict reconciliation. The source node represents o's initial replica. Apparently, each replication graph has a single source. Because all replicas are eventually consistent, it also has a single sink. Figure 1 shows an example graph where nodes are identified using numbers and double-parent nodes are colored in gray.

Coalesced replication graphs or CRG are the same as replication graphs except that consecutive single-parent nodes each with at most one child are merged together. We define the vector corresponding to a coalesced node as the vector of the "youngest" node in the hierarchy being merged, and define the node corresponding to the merged vectors as the node they coalesce into. Figure 2 shows the coalesced version of Figure 1.

Let us look at Figure 2 closely. We denote the vector of

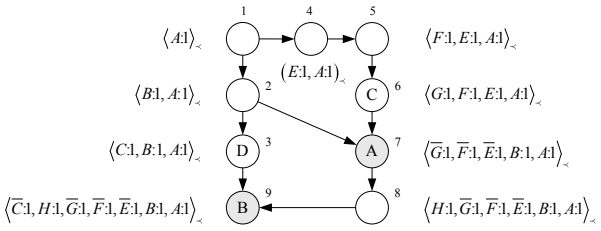

Figure 1. A replication graph. Zero valued elements have been removed from vectors.

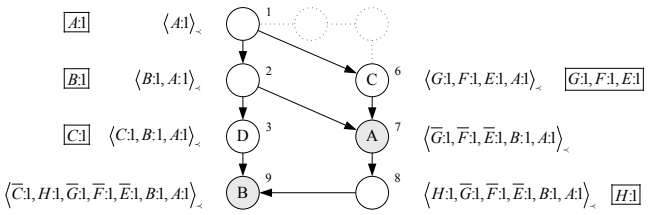

Figure 2. A coalesced replication graph (CRG)

node i as θ_i.[1] Now suppose site A synchronizes its vector θ_7 with site B's vector θ_9. SYNCC$_{\theta_9}(\theta_7)$ sends θ_9's first six elements from B to A but only the first two elements are new to A. Here, $|\Delta| = 2$ and $|\Gamma| = 3$. The reason of the $O(|\Gamma|)$ redundant transfer is that the exact difference between two vectors is unknown otherwise.

Fortunately, the order in rotating vectors sheds light on reducing the redundancy. It is observed that in a CRG, single-parent vectors[2] prefix parent vectors with a unique *segment*, i.e. a list of vector elements. For example, θ_3 prefixes θ_2 with $\langle C{:}1\rangle$; θ_6 prefixes θ_1 with $\langle G{:}1, F{:}1, E{:}1\rangle$. We refer to the segments that prefix single-parent vectors as *prefixing segments*. They are shown in Figure 2 in boxes. Double-parent vectors create no new segments but combine their parents' (in arbitrary order). As a results, a vector in CRG contains nothing but a series of segments.

Segments have three important properties. i) A segment has a unique set of elements: elements of the same site name must have different values in different segments. As soon as an element value is modified, the element is removed from the segment and rotated to the front of the vector, becoming part of a new prefixing segment. ii) Intra-segment order is persistent from vector to vector. For example, θ_6's prefixing segment appears in θ_7, θ_8, and θ_9 in the same order. iii) Segments never grow in size. They only shrink when elements are modified. A segment *vanishes* when its size reaches zero.

These three properties guarantee that a segment can be uniquely identified by *any* element in it. If an element e of a segment s in vector a is found in vector b with an equal or greater value, b must have the knowledge of s. In the equal case, s is part of b and e is still in s; in the greater case, e has been rotated out of s and s might have already vanished from b. This, in fact, leads to the design of SRV. For example, the five segments in θ_9 are $\langle C{:}1\rangle$, $\langle H{:}1\rangle$, $\langle G{:}1, F{:}1, E{:}1\rangle$, $\langle B{:}1\rangle$, and $\langle A{:}1\rangle$. When sending θ_9 to θ_7, we skip an entire segment if the value of the first element in the segment is less or equal than the corresponding element in θ_7. Eventually, only C, H, G and Bth elements are sent. Segment $\langle A{:}1\rangle$ is skipped all together because the Bth element has the conflict bit of zero.

Definition A *skip rotating vector* v is a CRV with a *segment bit* for each element. We denote the ith segment bit as $v.s[i]$.

Segment bits define segment boundaries. A segment bit of one indicates the last element of a segment. An observation is whenever a skip occurs during conflict reconciliation, a new segment forms in the vector being modified. Therefore, upon each skip, the algorithm sets the segment bit of the last modified element. Algorithm 4, SYNCS, shows operation details. The step of setting segment bits before skipping is shown at Line 7–10 on a's side. If the conflict bit of the received element is zero, the algorithm halts instead of skipping (Line 17).

A subtlety is that due to pipelining, the sender might have already sent the whole segment and moved on to the next one on hearing a skip request. Therefore, the receiver shall explicitly indicate which segment to skip. In the algorithm, both parties track the number of segments they have seen (variables $segs$). The sender skips a segment only if its own number matches the number sent with $SKIP$. A flag $skipping$ is maintained on the receiver to ignore received elements that should have been skipped.

A final issue is the maintenance of segment bits. Because an element with the segment bit set is the last one of a segment, when the element is rotated, the bit shall be carried on to its predecessor in the order of $\prec$. Therefore, we modify operation ROTATE(p, i) to set the segment bit of the ith element's predecessor if the ith segment bit is one.

4.1. Complexities of SRV

The storage complexity of SRV is $O(n)$. Incrementing an element in SRV due to replica updates consumes $O(1)$ space and time. Now we focus on the complexities of SYNCS.

Communication. An $O(1)$-byte $SKIP$ message is transmitted for each skipped segment. We denote the number of skipped segments as γ. The overall communication cost of SYNCS is then $O(|\Delta|+\gamma)$. To evaluate γ, we denote the set of nodes in the CRG that consists of the node of vector v and v's single-parent ancestors as Π_v. It is easy to see that the set of segments in v, including the vanished ones, bijectively maps to Π_v. Because the skipped segments in SYNCS$_b(a)$

1. θ_7 := SYNCC$_{\theta_6}(\theta_2)$ and θ_9 := SYNCC$_{\theta_3}(\theta_8)$. SYNCC$_{\theta_2}(\theta_6)$ and SYNCC$_{\theta_8}(\theta_3)$ would generate different element orders.

2. Since there is a one-to-one correspondence between nodes and vectors in the CRG, we use "nodes" and "vectors" interchangeably.

Algorithm 4 SYNCS$_b(a)$: synchronize SRV a with b

On b's hosting site:

```
1:  cur ← ⌊b⌋ {cur is a quadruple}
2:  segs ← 0
3:  skipping ← false
4:  repeat
5:    if ¬skipping then
6:      send cur
7:    end if
8:    if (received (SKIP, arg)) ∧ (arg = segs) then
9:      skipping ← true
10:   end if
11:   if cur's segment bit = 1 then
12:     segs ← segs + 1
13:     skipping ← false
14:   end if
15:   if cur = ⌈b⌉ then
16:     send HALT; halt
17:   end if
18:   cur ← cur's successor in ≺_b
19: until received HALT
```

On a's hosting site:

```
1:  prev ← φ
2:  segs ← 0 {maintenance of segs omitted for brevity}
3:  skipping ← false
4:  reconcile ← a ∥ b
5:  repeat
6:    receive (i, u_i, c_i, s_i)
7:    if u_i ≤ a[i] then
8:      if ¬skipping then
9:        if reconcile ∧ prev ≠ φ then
10:         a.s[prev] ← 1
11:       end if
12:       if c_i = 1 then
13:         send (SKIP, segs)
14:         skipping ← true
15:         reconcile ← true
16:       else
17:         send HALT; halt
18:       end if
19:     end if
20:   else
21:     skipping ← false
22:     ROTATE_a(prev, i)
23:     prev ← i
24:     if reconcile then c_i ← 1 end if
25:     a[i] ← u_i; a.c[i] ← c_i; a.s[i] ← s_i
26:   end if
27: until the next received data is HALT
```

are those shared by a and b, γ is bounded by $|\Pi_b \cap \Pi_a|$. γ may be smaller if some segments in b have vanished, or some are not reached before algorithm termination (e.g. $\langle A{:}1\rangle$ of θ_9 in Figure 2). Let Φ_b and Λ_b be the set of vanished and not reached segments in b, respectively, then $\gamma = |(\Pi_b \cap \Pi_a) \setminus \Phi_b \setminus \Lambda_b|$. The composition of Φ_b and Λ_b highly depend on the evolving pattern of the system.

Time. Because the sender iterates through all the skipped elements, the time complexity of SYNCS is the same as SYNCC which is $O(|\Delta| + |\Gamma|)$. In fact, this can be easily reduced to $O(|\Delta| + \gamma)$ by using site name fields on both boundaries of a segment to indicate the other boundary. We do not implement it for clarity and brevity.

Space. SYNCS is O(1) in space.

5. The lower bound of vector synchronization

Is SRV an optimal implementation? To answer this question, we shall find the minimal information exchange required by the vector synchronization problem (other complexities of SRV are trivially optimal).

Theorem 5.1: Any vector synchronization algorithm with $O(n)$ storage complexity requires $\Omega(|\Delta| + |\Pi_a \cap \Pi_b|)$ communication to synchronize vectors a and b in the worst case.

Proof: The synchronization problem is to modify one of vectors a and b, such that the ith value of the modified vector becomes equal to $max(a[i], b[i])$ for all i. Assuming a is the vector to be modified, at least $\Omega(|\Delta|)$ bytes need to be sent from b to a where $\Delta = \{i : b[i] > a[i]\}$. However, Δ shall be identified first. This is equivalent to identifying the complementary set $\overline{\Delta} = \{i : b[i] \leq a[i]\}$.

A *unique element* is a vector element whose site name and value combined are unique. For example, $(A, 1)$ and $(A, 2)$ are different unique elements. Let E be the set of unique elements, P be the set of prefixing segments, and N be the set of single-parent nodes in a CRG. According to the discussion on segments in § 4, for any given CRG, there is a surjection from E to P. Because P bijectively maps to N, there exists a surjection $f : E \mapsto N$. According to the properties of CRG, for any unique element e, $f(e) \in \Pi_v$ iff vector v contains e or v contains an element with the same site name and a value greater than e.

Therefore, for all $i \in \overline{\Delta}$, $f(b_i) \in \Pi$ where b_i is the ith element of b and $\Pi = \Pi_a \cap \Pi_b$. Obviously, $\Pi \subseteq N$. We denote the preimage of Π under f as $f^{-1}[\Pi]$, denote the bijection from b_i to i as g_b, and denote the set of elements in b as E_b. Then $\overline{\Delta} = g_b\left[f^{-1}[\Pi] \cap E_b\right]$. This means that $\overline{\Delta}$ is identified iff both f^{-1} and Π are known. Function f^{-1} can be maintained locally by exploiting the properties of segments, as described in § 4. Π has to be determined through information exchange between a and b. Nodes in a CRG are interrelated: if a node π is in Π, π's ancestor nodes must be in Π as well. Therefore, the minimal information exchange to determine Π could be less than

$\Omega(|\Pi|)$. However, it requires storage complexity greater than $O(n)$ to trace causal dependency using CRGs or equivalent data structures.

If no CRG or equivalent is stored, from the algorithm's perspective, a node's presence in Π is a random event independent of other nodes. Therefore, $\Omega(|\Pi|)$ is the minimal communication required to determine Π in the worst case. In turn, $\Omega(|\Pi|)$ is also required to determine $\overline{\Delta}$. That is, there exists no set M and surjection $h : E \mapsto M$, such that $\forall \Pi \subset N, \exists \Theta \subset M, h^{-1}[\Theta] = f^{-1}[\Pi] \wedge \Omega(|\Theta|) < \Omega(|\Pi|)$. It can be shown using the properties of surjection. We omit the proof due to space constraints. □

Corollary 5.2: Any version vector synchronization algorithm with $O(n)$ storage complexity requires $\Omega(|\Delta| + \gamma)$ communication.

The proof of this corollary is in [22]. Since SRV achieves $O(|\Delta| + \gamma)$ communication complexity, it is optimal among all vector implementations.

6. Optimizing for operation transfer

While state transfer overwrites an entire replica during synchronization, *operation transfer* maintains a history of operations and sends only missing operations to bring a replica up to date. *Hybrid transfer* intermingles state and operation transfer. For example, a system may preserve a short history of operations and when a replica is too old, the entire object is transmitted [1, §7.2]. As hybrid transfer is a degeneration of operation transfer, we do not distinguish the two models in the rest of the paper.

Operation transfer records causal relations between operations for fine grained conflict resolution and other purposes. For example, operational transformation (OT) relies on the relations to alter operations [9]; distributed revision control systems use the causal hierarchy for versioning control and efficient three-way merging. In these systems, a *causal graph* is usually attached to each replica. A causal graph is the same as a replication graph (Figure 1), except that a node in a causal graph represents an operation instead of replicas. The sink of a replica r's causal graph represents the latest operation executed on r. For example, the causal graph of site A in Figure 1 consists of nodes 1, 2, 4–7; site C's causal graph consists of nodes 1, 4–6. They are redrawn on Figure 3. It is notable that causal graphs of the same object share at least the same source node (node 1 in the example).

Two replicas are compared as follows. If the sink node of one replica is found in the causal graph of the other replica but not vice versa, the former replica causally precedes the latter. If neither sink is found in the other graph, the two replicas are concurrent. They are identical if sinks are found in both ways. We assume that a hash table is used to lookup nodes in $O(1)$ running time. Comparison is therefore an optimal operation.

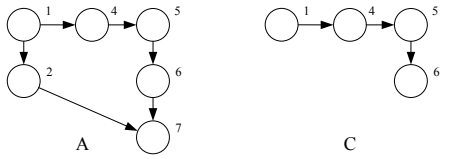

Figure 3. Causal graphs of site A and C from Figure 1

When synchronizing replicas, their causal graphs must also be synchronized so that the resulting graphs are identical to each other. Traditionally, the entire graph is sent which brings much overhead in communication and processing, particularly when the size of the graph is large due to frequent updates or long object lifespan.

6.1. Incremental synchronization of causal graphs

We propose an optimal algorithm to incrementally synchronize causal graphs. Given two graphs with the same source node, the synchronization problem is to modify one graph so that it becomes the union of the two original graphs. This shall be true regardless of the causal relation between them. For example, synchronizing conflicting sites A and D in Figure 1 shall result in a graph containing nodes 1–7. After this, conflict reconciliation is invoked and a new node is added as the new sink.

Similar to rotating vectors, the idea of the algorithm is to send only the difference. It is more intuitive than previous algorithms due to explicit representation of causal relations. Suppose the algorithm modifies causal graph a to synchronize with b. It runs a depth-first search (DFS) on b, starting from the sink and proceeding in the reverse direction of the arcs. For each node being traversed, it sends the information of the node to a. On detecting a node π that already exists in a, the receiving site signals the other site to abort processing on the current "branch" of b and to continue from the next branch. It is safe to do so, because all the ancestors of π are guaranteed to be in a if π is so. The algorithm proceeds until no more branches are left.

Take Figure 3 as an example. Suppose the algorithm modifies C's graph to synchronize with A's. There are two branches in A's graph. Assuming the branch of nodes $7, 6 \cdots 1$ is visited first, on receiving node 6 from A, C notifies A to abort. Therefore, instead of continuing on to node 5 and beyond, C jumps to the other branch starting from node 2. Hence, only the missing nodes plus an overlapping node are transmitted for each branch, dramatically reducing network overhead for large graphs with small differences.

Although the idea is simple, the implementation needs extra efforts to support pipelining (§ 3.1). For example, by the time A receives C's abort request, it might have finished the current branch and moved on to another one. A shall ignore the request in this case. This issue is further complicated by possibly sophisticated graph topologies. In

Algorithm 5 SYNCG$_b(a)$: synchronize graph a with b

V_g, A_g: the set of nodes and arcs in causal graph g
$LP(i), RP(i)$: return node i's left and right parent

On b's hosting site:

```
1:  visited ← φ {the set of visited nodes}
2:  s ← {b's sink node} {the stack for depth-first search}
3:  while s ≠ φ do
4:     i ← s.pop()
5:     if i ∉ visited then
6:        visited ← visited ∪ {i}
7:        send i, LP(i), RP(i)
8:        if RP(i) ≠ nil then s.push(RP(i)) end if
9:        if LP(i) ≠ nil then s.push(LP(i)) end if
10:    end if
11:    if (received skipto) ∧ (skipto ∉ visited) then
12:       while s.peek() ≠ skipto do s.pop() end while
13:    end if
14: end while
15: send HALT
```

On a's hosting site:

```
1:  s' ← φ {the mirroring stack}
2:  skipping ← false
3:  repeat
4:     receive i, lp, rp
5:     if i ∈ V_a then
6:        if ¬skipping then
7:           skipping ← true
8:           send s'.pop()
9:        end if
10:    else
11:       skipping ← false
12:       if i = s'.peek() then s'.pop() end if
13:       V_a ← V_a ∪ {i}
14:       if lp ≠ nil then
15:          A_a ← A_a ∪ {(lp, i)}
16:       end if
17:       if rp ≠ nil then
18:          A_a ← A_a ∪ {(rp, i)}
19:          s'.push(rp)
20:       end if
21:    end if
22: until the next received data is HALT
```

the solution we propose, C mirrors the stack used by A for DFS. On seeing an existing node, C examines the mirroring stack and tells A not only to abort but also the identifier of a node π in the stack, which is the starting point of the next branch C expects to receive. A rewinds the stack up to π only if π has not been sent.

Algorithm 5 details the procedure. For convenience, we arbitrarily refer to either parent of a double-parent node as left; a single-parent node only has a parent on the left. The mirroring stack s' in the receiver is identical to the sender's stack, except that s' only keeps nodes not existing in the receiver's graph (Line 19, a's). On receiving an existing node, the receiver notifies the sender with the node identifier popped from s'; it also sets the *skipping* bit to avoid duplicate notifications (Line 5–8, a's). The sender rewinds the stack only if the received node is yet to be visited (Line 11, 12, b's).

SYNCG transmits $O(|V_b \setminus V_a| + |A_b \setminus A_a|)$ bytes over the network. It is obviously the optimal amount of information exchange. Reference [22] lists other complexities.

7. Related work

We address communication overhead in exchanging data structures that preserve causality for distributed computation. There is little previous work in this area. In the context of vector timestamps, Singhal and Kshemkalyani [23] propose to send to a site only those vector elements that have been changed since the last transmission to this site. Two additional vectors are maintained for each timestamp to calculate the change set. However, this algorithm is not suitable for concurrency control because it models local events and remote messaging in one causal relation, which is not the case in replication systems. Fowler and Zwaenepoel [24] describe a method to trace causality in which only one vector element is sent along messages. However, this method is too expensive for online causal detection in terms of computation, and is mostly useful for off-line analysis.

As discussed in § 2.2, attempts have been made to reduce the size of version vectors by either removing inactive sites [19, 20] or by intentionally introducing inaccuracy [17, 18]. These efforts are orthogonal and can be easily applied to any of BRV, CRV, and SRV. As element values are built upon integer numbers, several solutions [25, 26] are proposed to address their unbounded space requirement. They are also orthogonal to the chosen implementations of vectors.

8. Conclusions

We address communication overhead of concurrency control in optimistic replication systems. To detect conflicts, state-transfer systems attach certain data structures to each replica. We argue that version vectors have the least storage complexity among known schemes. We then present three vector implementations to minimize communication and computation in vector synchronization. BRV is an optimal implementation for systems without conflict reconciliation; CRV works well when conflict rates are low. With two additional bits per vector element, SRV achieves optimal communication and running time.

One vector per replica is not sufficient for operation-transfer systems. Instead, they use causal graphs to capture

inter-operation relations. We propose a causal graph exchange algorithm that sends only the difference between two graphs with optimal communication. Network pipelining is adopted in all the above algorithms. It accelerates algorithm completion at the cost of marginal increases in the amount of data being transmitted.

Acknowledgement

We would like to thank the anonymous reviewers for their feedback and Eric Y. Chen, Madalin Mihailescu, Jin Chen, and Saeed Ghanbari for their valuable discussions and comments.

References

[1] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[3] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News*, 28(5):190–201, 2000.

[4] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, 1994.

[5] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[6] G. J. Popek, R. G. Guy, Jr. T. W. Page, and J. S. Heidemann. Replication in Ficus distributed file systems. In *IEEE Workshop on Management of Replicated Data*, pages 20–25, 1990.

[7] D. Malkhi and D. Terry. Concise version vectors in WinFS. *Distrib. Comput.*, 20(3):209–219, 2007.

[8] A. Yip, B. Chen, and R. Morris. Pastwatch: a distributed version control system. In *NSDI*, 2006.

[9] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW*, pages 59–68, 1998.

[10] B. Du and E. A. Brewer. DTWiki: a disconnection and intermittency tolerant wiki. In *WWW*, pages 945–952, 2008.

[11] Jr. D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Distrib. Sys., Vol. II: distributed data base systems*, pages 306–312, 1986.

[12] B. B. Kang, R. Wilensky, and J. Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *ICDCS*, pages 670–677, 2003.

[13] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–182, 1995.

[14] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[15] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

[16] R. Baldoni and G. Melideo. On the minimal information to encode timestamps in distributed computations. *Inf. Process. Lett.*, 84(3):159–166, 2002.

[17] C. Valot. Characterizing the accuracy of distributed timestamps. *SIGPLAN Notices*, 28(12):43–52, 1993.

[18] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distrib. Comput.*, 12(4):179–195, 1999.

[19] D. Ratner, R. Reiher, and G. J. Popek. Dynamic version vector maintenance. Technical Report CSD-970022, UCLA, 1997.

[20] Y. Saito. Unilateral version vector pruning using loosely synchronized clocks. Technical Report HPL-2002-51, HP Lab, 2002.

[21] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, 7(3):149–174, 1994.

[22] W. Wang and C. Amza. On optimal concurrency control for optimistic replication. Technical report, http://www.weihanwang.com/icdcs09, 2009.

[23] M. Singhal and A. Kshemkalyani. An efficient implemenation of vector clocks. *Inf. Process. Lett.*, 43:47–52, 1992.

[24] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *ICDCS*, pages 134–141, 1990.

[25] A. Arora, S. Kulkarni, and M. Demirbas. Resettable vector clocks. In *PODC*, pages 269–278, 2000.

[26] Y. Huang and P. S. Yu. Lightweight version vectors for pervasive computing devices. In *International Workshops on Parallel Processing*, pages 43–48, 2000.