Brief Announcement: Collect in the Presence of Continuous Churn with Application to Snapshots and Lattice Agreement*

Hagit Attiya
Technion, Israel
hagit@cs.technion.ac.il

Sweta Kumari
Technion, Israel
sweta@cs.technion.ac.il

Archit Somani
Technion, Israel
archit@cs.technion.ac.il

Jennifer L. Welch
Texas A&M University
welch@cse.tamu.edu

ACM Reference Format:
Hagit Attiya, Sweta Kumari, Archit Somani, and Jennifer L. Welch. 2020. Brief Announcement: Collect in the Presence of Continuous Churn with Application to Snapshots and Lattice Agreement. In *Symposium on Principles of Distributed Computing (PODC '20), August 3–7, 2020, Virtual Event, Italy.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3382734.3405709

1 INTRODUCTION

A popular programming technique that contributes to designing provably-correct distributed applications is to use shared objects for interprocess communication, instead of more low-level techniques. Although shared objects are a convenient abstraction, they are not generally provided in large-scale distributed systems; instead, the processes keep individual copies of the data and communicate by sending messages to keep the copies consistent. Traditional distributed computing considers a *static* system, with known bounds on the number of fixed computing nodes and the number of possible failures. *Dynamic* distributed systems allow nodes to enter and leave the system at will, either due to failures and recoveries, moving in the real world, or changes to the systems' composition. Motivating applications include those in peer-to-peer, sensor, mobile, and social networks, as well as server farms.

The usefulness of shared memory programming abstractions has been long-established for static systems [3]. This success has inspired work on providing the same for newer, dynamic, systems.

In this paper, we promote the *store-collect* shared object [5] as a primitive well-suited for dynamic message-passing systems with an ever-changing set of participants, a phenomenon called "churn". Each node can store a value in the object with a STORE operation and can collect the latest value stored by each node with a COLLECT operation. We focus on the situation when nodes enter and leave, but the resulting network is always fully connected, which could be due to, say, an overlay network. We assume nodes can crash, as long as the number of crashed nodes is no more than a fixed fraction of the total number of nodes in the system.

A continuous-churn-tolerant store-collect object can be implemented fairly easily. We adopt the system model in [4], which allows never-ending churn as long as not too many churn events take place during the length of time that a message is in transit. To capture this constraint, there is an assumed upper bound D on the maximum message delay, but no lower bound. Nodes do not know D and have no local clocks, causing consensus to be impossible to solve [4]. During any time interval of length D, the number of churn events that can occur is a fraction of the number of nodes in the system at the beginning of the interval. Our algorithm for implementing a (non-linearizable) store-collect object is based on the (linearizable) read-write register algorithm in [4]. It is simple and efficient: once a node joins, it completes a store operation within one round-trip, and a collect operation within two round-trips. A by-product of our work in this paper is a revised proof of the churn management protocol that is much more accessible than that in [4].

The store-collect object specification is versatile, yielding efficient implementations for objects that are not linearizable. There are simple algorithms to implement (non-linearizable) max-registers, abort flags, and sets using store-collect [7].

Building an atomic snapshot on top of a store-collect object is easy! We present a simple algorithm with an elegant correctness proof, which is more efficient than one based on registers as the values can be collected in parallel instead of in series. In static systems, atomic snapshot objects can be used to build multi-writer registers, concurrent timestamp systems, counters, and accumulators, and to solve approximate agreement and randomized consensus. They have also been applied in dynamic systems.

Atomic snapshots yield a very simple implementation of *generalized lattice agreement* (*GLA*) [9] in a dynamic system. GLA is an extension of (single-shot) *lattice agreement*, well-studied in the static shared memory model [6]. Notably, [11] considers dynamic systems subject to changes in the composition due to reconfiguration and provides an implementation for a large class of shared objects, including conflict-free replicated data types, that can be modeled as a lattice. Unlike our work, it requires that changes to the system composition eventually cease in order to ensure progress.

While it is easy to layer linearizability on top of a store-collect object, not every application needs the costs associated with linearizability, and store-collect gives the flexibility to avoid them. Our approach is modular, hiding the underlying complications of the message-passing and continuous churn from higher layers.

The problem of implementing shared objects in the presence of *ongoing* churn and crash failures is studied by Baldoni et al., e.g., [8]. Unlike our results, that work assumes the system size is restricted to a fixed window and the system is eventually synchronous.

If the level of churn is too great, our store-collect algorithm is not guaranteed to preserve the safety property; that is, a collect might miss the value written by a previous store (see [4]). This behavior is in contrast to the algorithms in [2, 10], which never violate the safety property but only ensure progress once reconfigurations cease. In future work, we would like to either improve our algorithm

*Supported in part by ISF grant 380/18 and NSF grant 1816922. Full version in [7].

PODC '20, August 3–7, 2020, Virtual Event, Italy

ACM ISBN 978-1-4503-7582-5/20/08.
https://doi.org/10.1145/3382734.3405709

to avoid this behavior or prove that any algorithm that tolerates ongoing churn is subject to such bad behavior.

2 THE STORE-COLLECT ALGORITHM

Each node runs a client thread and a server thread. A shared *store-collect object* supports concurrent execution of *store* and *collect* operations, invoked by *client* threads. $\text{STORE}_p(v)$, where v is a value and p is a client, gets a response ACK_p. An invocation COLLECT_p gets a response $\text{RETURN}_p(V)$, where V is a *view*, that is, a set of client-value pairs without repetition of client ids. $V(p)$ is v if $\langle p, v\rangle \in V$ and $\perp$ if no pair in V has p as its first element. Informally, a *collect* operation should return a view containing the latest value stored by each client. We also require that a later collect operation (possibly by a different client) returns later values (see [7]).

Nodes track the system composition with an algorithm that is the same as the one in CCREG [4], except for a single line which updates the current view (see [7]). A node p maintains a set *Changes* of events concerning the nodes that have entered the system. When an ENTER_p event occurs, p adds *enter(p)* to its *Changes* set and broadcasts an **enter** message requesting information about prior events. When p finds out that another node q has entered the system, either by receiving an **enter** message directly from q or by receiving an **enter-echo** message for q from a third node, it adds *enter(q)* to its *Changes* set. When p receives an **enter** message from a node q, it replies with an **enter-echo** message containing its *Changes* set, its current estimate local view (*LView*) of the state of the simulated object. The first time that p receives an **enter-echo** in response to its own **enter** message from a joined node, it computes *join_threshold*, the number of **enter-echo** messages it needs to get before it can join and increments its *join_counter*.

A fraction γ is used to calculate *join_threshold*, the number of **enter-echo** messages that should be received before joining, based on the size of the *Present* set (nodes that have entered, but have not left). Setting γ is a key challenge in the algorithm as setting it too small might not propagate updated information, whereas setting it too large might not guarantee termination of the join.

Once p receives the required number of replies to its **enter** message, p adds *join(p)* to its *Changes* set and broadcasts a message saying that it has joined and sets local variable *is_joined* to true. When p finds out that another node q has joined, it adds *join(q)* to its *Changes* set. When a LEAVE_p event occurs, p broadcasts a **leave** message and halts. When p finds out that a node q left the system, it adds *leave(q)* to its *Changes* set.

Once a node has joined, its client thread can handle *collect* and *store* operations (Algorithm 1) and its server thread (Algorithm 1) can respond to clients. Our implementation adds a sequence number, *sqno*, to each value in a view, which is now a set of triples, $\{\langle p, v, sqno\rangle, \ldots\}$, without repetition of node ids. We use the notation $V(p) = v$ if there exists *sqno* such that $\langle p, v, sqno\rangle \in V$, and $\perp$ if no triple in V has p as its first element. A merge of two views, V_1 and V_2, picks the latest value stored by each node according to the highest *sqno*. Note that $V_1, V_2 \preceq merge(V_1, V_2)$.

Each node keeps a local copy of the current view in its *LView* variable. In a *collect operation*, a client thread requests the latest value of servers' local views using a **collect-query** message. When a server node p receives a **collect-query** message, it responds with its *LView* through a **collect-reply** message if p has joined the system. When the client receives a **collect-reply** message, it merges its *LView* with the *received view (RView)*, to get the latest value corresponding to each node. Then the client waits for sufficiently many **collect-reply** messages before broadcasting the current value of its *LView* variable in a **store** message. When server p receives a **store** message with a view *RView*, it merges *RView* with its local *LView* and, if p is joined, it broadcasts **store-ack**. The client waits for sufficiently many **store-ack** messages before returning *LView* to complete the *collect*; this threshold is recalculated to reflect possible changes to the system composition that the client has observed.

In a *store operation*, a client thread merges the value it wishes to store with its local view, and broadcasts the resulting view with a new sequence number. It updates its local variable *LView* to reflect the new value by doing a merge and broadcasts a **store** message. When server p receives a **store** message with view *RView*, it merges *RView* with its local *LView* and, if p is joined, it broadcasts **store-ack**. The client waits for sufficiently many **store-ack** messages before completing the *store*.

Algorithm 1 CCC—Client and server code, for node p.

Local Variables:
- *optype*: string, initially $\perp$ // which type of operation is pending
- *tag*: int, initially 0 // identify current operation
- *threshold*: int, initially 0 // no. replies required in current phase
- *counter*: int, initially 0 // no. replies received in current phase

Derived Variable:
- $Members = \{q \mid join(q) \in Changes \wedge leave(q) \notin Changes\}$

When COLLECT_p **occurs:**
1: *optype = collect; tag++*
2: *threshold* $= \beta \cdot |Members|$; *counter = 0*
3: **broadcast** ⟨**collect-query,** *tag*, p⟩

When RECEIVE_p ⟨**collect-reply,** *RView, t, q*⟩ **occurs:**
4: **if** $(t == tag) \wedge (q == p)$ **then**
5: *LView = merge(LView, RView)* ; *counter++*
6: **if** (*counter* $\geq$ *threshold*) **then**
7: *threshold* $= \beta \cdot |Members|$; *counter = 0*
8: **broadcast** ⟨**store,** *LView, tag,* p⟩

When $\text{STORE}_p(v)$ **occurs:**
9: *optype = store; tag++* ; *sqno++*
10: *LView = merge(LView,*$\{\langle p, v, sqno\rangle\}$)
11: *threshold* $= \beta \cdot |Members|$; *counter = 0*
12: **broadcast** ⟨**store,** *LView, tag,* p⟩

When RECEIVE_p ⟨**store-ack,** *t, q*⟩ **occurs:**
13: **if** $(t == tag) \wedge (q == p)$ **then**
14: *counter++*
15: **if** (*counter* $\geq$ *threshold*) **then**
16: **if** (*optype* == *store*) **then return** ACK
17: **else return** *LView*

When RECEIVE_p ⟨**store,** *RView, tag, q*⟩ **occurs:**
18: *LView = merge(LView, RView)*
19: **if** (*is_joined*) **then broadcast** ⟨**store-ack,** *tag, q*⟩
20: **broadcast** ⟨**store-echo,** *LView*⟩

When RECEIVE_p ⟨**collect-query,** *tag, q*⟩ **occurs:**
21: **if** (*is_joined*) **then broadcast** ⟨**collect-reply,** *LView, tag, q*⟩

When RECEIVE_p ⟨**store-echo,** *RView*⟩ **occurs:**
22: *LView = merge(LView, RView)*

The fraction β is used to calculate the number of messages that should be received (stored in local variable *threshold*) based on the size of the *Members* set (estimate of nodes that have joined and not left), for the operation to terminate. Setting β is also a key challenge in the algorithm as setting it too small might not return correct information from *collect* or *store*, whereas setting it too large might not guarantee termination of the *collect* and *store*.

Fortunately, there exist values for the parameters γ and β that allow the algorithm to be correct. In the extreme case of no churn, the failure fraction can be as large as 0.33; in this case, it suffices to set both γ and β to .67. As the churn rate increases up to 0.04, the failure fraction must decrease approximately linearly until reaching 0.03; in this case, it suffices to set γ to .75 and β to .78.

3 ATOMIC SNAPSHOTS

We employ the store-collect algorithm to implement *atomic snapshots* [1]. A *snapshot view* is a set of (node id, value) pairs, without duplicate node ids. An *atomic snapshot* provides two operations: Scan() returns a snapshot view, while Update(v) takes a value v, changes the value associated with the invoking node to be v, and returns Ack. Our algorithm uses repeated collects to identify an atomic scan when two consecutive collects return the same collected views. Updates help scans to complete by embedding an atomic scan that can be borrowed by overlapping scans they interfere with.

We use a store-collect object, whose values are five-tuples: *val* holds the argument of the most recent update invoked at p; *usqno* holds the number of updates performed by p; *ssqno* holds the number of scans performed by p; *sview* holds a snapshot view that is the result of a recent scan done by p, used to help other nodes complete their scans; *scounts* holds a set of counts of how many scans have been done by the other nodes, as observed by p.

To Scan, Algorithm 2 increments the scan sequence number and performs a store on the shared store-collect object with all the other components unchanged. Then, the first view is collected. In a while loop, it collects an additional view. If the two most recently collected views are equal (*successful* double collect), this view is returned. Otherwise, the algorithm checks whether the last collected view contains a node q that has observed its own *ssqno* (in *scounts*). If this condition holds, the snapshot view of q is returned.

An Update first obtains all scan sequence numbers from a collected view and assigns them to a local variable *scounts*. Next, an embedded scan is performed and the returned view is saved in a local variable *sview*. Then *val* is set to the argument value and the update sequence number is incremented. Finally the new value, update sequence number, collected view, and set of scan sequence numbers are stored; the node's scan sequence number is unchanged.

An operation terminates within $O(N)$ collects and stores, where N is the number of nodes active when it starts (see [7]).

4 GENERALIZED LATTICE AGREEMENT

Let $\langle L, \sqsubseteq \rangle$ be a lattice, where L is the domain of lattice values, ordered by $\sqsubseteq$, with a join operator, $\sqcup$, that merges lattice values. A GLA object supports a Propose operation whose input and response are both lattice values. The input to p's ith Propose is denoted v_i^p and the response is w_i^p. The following conditions are required:

Algorithm 2 Atomic snapshot: code for node p.

Local Variables:
- *ssqno*: int, initially 0 // no. scans p has invoked so far
- *scounts*: set of (node id, integer) pairs without duplicate ids; initially $\emptyset$
- *val*: element of Val_{AS}, initially $\perp$ // argument to p's recent update
- *usqno*: int, initially 0 // no. updates p has invoked so far
- *sview*: snapshot view, initially $\emptyset$ // result of last embedded scan
- V_1, V_2: store-collect views, both initially $\emptyset$

When $\text{SCAN}_p()$ **occurs:**
1: ***ssqno*++**
2: $\text{STORE}_p(\langle -, -, \textbf{\textit{ssqno}}, -, - \rangle)$
3: $V_1 = \text{COLLECT}_p()$
4: **while** ***true*** **do**
5: $V_2 = V_1$; $V_1 = \text{COLLECT}_p()$
6: **if** $(V_1 == V_2)$ **then**
7: **return** V_1.***val*** // **direct scan**
8: **if** $\exists q$ **such that** $\langle p, \textbf{\textit{ssqno}} \rangle \in V_1(q).\textbf{\textit{scounts}}$ **then**
9: **return** $V_1(q).\textbf{\textit{sview}}$ // **borrowed scan**

When $\text{UPDATE}_p(v)$ **occurs:**
10: ***scounts*** = $\text{COLLECT}_p()$.***ssqno***
11: ***sview*** = $\text{SCAN}_p()$ // **embedded scan**
12: ***val*** = v ; ***usqno*++**
13: $\text{STORE}_p(\langle \textbf{\textit{val}}, \textbf{\textit{usqno}}, -, \textbf{\textit{sview}}, \textbf{\textit{scounts}} \rangle)$
14: **return** Ack

(a) Validity: Every response value w_i^p is the join of some values proposed before this response, including v_i^p, and all values returned to any node before the invocation of p's ith Propose. *(b) Consistency:* Any two values w_i^p and w_j^q are comparable.

Our GLA algorithm (see [7]) uses an atomic snapshot object, in which each node stores a single lattice value (*val*). A Propose operation is simply an Update of a lattice value which is the join of all the node's previous inputs, followed by a Scan returning the analogous values for all nodes, whose join is the output of the Propose. Validity and consistency are immediate from the atomic snapshot properties, as is the complexity.

REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.
[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7, 2011.
[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
[4] H. Attiya, H. C. Chung, F. Ellen, Saptaparni, and J. L. Welch. Emulating a shared register in an asynchronous system that never stops changing. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):544–559, 2018.
[5] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Dist. Comp.*, 15(2):87–96, Apr. 2002.
[6] H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.
[7] H. Attiya, S. Kumari, A. Somani, and J. L. Welch. Store-collect in the presence of continuous churn with application to snapshots and lattice agreement, 2020. https://arxiv.org/abs/2003.07787.
[8] R. Baldoni, S. Bonomi, and M. Raynal. Implementing set objects in dynamic distributed systems. *Journal of Comp. and Sys. Sci.*, 82(5):654– 689, 2016.
[9] J. M. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.
[10] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Dist. Comp.*, 23(4):225–272, 2010.
[11] P. Kuznetsov, T. Rieutord, and S. Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, pages 31:1–31:17, 2019.