Distributed Garbage Collection for General Graphs

Hari Krishan
Louisiana State University
hari1106@gmail.com

Steven R. Brandt
Louisiana State University
sbrandt@cct.lsu.edu

Costas Busch
Louisiana State University
busch@csc.lsu.edu

Gokarna Sharma
Kent State University
sharma@cs.kent.edu

Abstract

We propose a scalable, cycle-collecting, decentralized, reference counting garbage collector with partial tracing. The algorithm is based on the Brownbridge system but uses four different types of references to label edges. Memory usage is $O(\log n)$ bits per node, where n is the number of nodes in the graph. The algorithm assumes an asynchronous network model with a reliable reordering channel. It collects garbage in $O(E_a)$ time, where E_a is the number of edges in the *induced subgraph*. The algorithm uses termination detection to manage the distributed computation, a unique identifier to break the symmetry among multiple collectors, and a transaction-based approach when multiple collectors conflict. Unlike existing algorithms, ours is not centralized, does not require barriers, does not require migration of nodes, does not require back-pointers on every edge, and is stable against concurrent mutation.

Keywords Distributed Garbage Collection; Distributed Termination Detection; Reference Graph; Cycles; Strong, Weak, Phantom Count; Reference Count

ACM Reference Format:
Hari Krishan, Steven R. Brandt, Costas Busch, and Gokarna Sharma. 2018. Distributed Garbage Collection for General Graphs. In *Proceedings of 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3210563.3210572

1 Introduction

Garbage-collected languages are widely used in distributed systems, including big-data applications in the cloud [5, 8, 15]. Languages in this space include Java, Scala, Python, etc., and platforms include Hadoop, Spark, etc. Bruno et al. [5] studied necessity and significance of garabge collection in big data environments. Garbage collection is seen as a significant benefit by the developers of these applications and platforms because it eliminates a large class of programming errors, which translates into higher developer productivity.

One significant place where the convenience of garbage collection does not exist, however, is in distributed, asynchronous multitask systems (AMT). Frameworks in this space include Charm++ [12], UINTAH [18], HPX [11], etc. The current state of the art with these high-performance scientific codes is to use simple reference counting and to manually free edge references to reclaim memory. From private conversations, the authors know that these research groups have struggled with memory issues and would like to have an option for garbage collection.

What is needed is something distributed, decentralized, scalable, that can run without "stopping the world," and has good time and space complexity.

There are two main types of garbage collectors, namely Tracing and Reference Counting (RC). Tracing collectors track all the reachable objects from the root (in the reference graph) and reclaim all the unreachable objects. RC collectors count the number of objects pointing to a given block of data at any point in time. When a reference count is decremented and becomes zero, its object is garbage, otherwise it *might* be garbage and some tracing of edges through a subset of the graph is necessary (note that the cost can be reduced if tracing is not done aggressively). Unfortunately, cycles among distributed objects are frequent [22]. Previous attempts at distributed garbage collection, e.g. [9, 13, 14, 16, 17, 25], suffer from the need for centralization, global barriers, the migration of objects, or have inefficient time and space complexity guarantees.

We note that our algorithms fits the restrictions of the $\mathcal{CONGEST}$ model [7], commonly used to analyze distributed algorithms. This model describes a distributed system where the topology of the network is restricted such that each node can only send messages along its edges to its immediate neighbors. Communication and computation is synchronous in $\mathcal{CONGEST}$, meaning that computation is divided into rounds during which each node can send a message to each of its neighbors, and each message is of size $O(\log(n))$ bits. The model allows, however, that nodes

ISMM'18, June 18, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5801-9/18/06…$15.00
https://doi.org/10.1145/3210563.3210572

and communication links experience no faults and nodes have unique IDs. In the present work, this model is used for determining time complexity.

Contributions: We present a hybrid reference counting algorithm for garbage collection in distributed systems that is described by the $\mathcal{CONGEST}$ model [7] and requires that messages are neither lost nor duplicated. Our algorithm collects both non-cyclic and cyclic garbage in distributed systems of arbitrary topology by advancing on the three reference count collector which only works for a single machine [3] based on the Brownbridge system[1] [4]. The advantage for such systems is that they can frequently determine that a reference count decrement does not create garbage without doing any non-trivial work (they only need to check that the "strong count" is not zero).

This work builds on the algorithm described in [3] (herein called SWP because it uses three types of references: Strong, Weak, and Phantom). The multi-collector algorithm described in the present work (herein called SWPR because it uses four types of references: Strong, Weak, Phantom, and Recovered) work extends SWP in three significant ways. First, SWPR never locks more than one object at a time. The SWP algorithm had to lock two nodes to properly update an edge, and an arbitrarily large number of nodes could be locked during a collection. Thus, SWPR is more suitable for use in a distributed setting and better able to exploit parallelism. Second, SWP requires two bits of data to be tracked per edge in the graph, whereas SWPR requires none.

Our proposed algorithm is scalable because it collects garbage in $O(E_a)$ time using only $O(\log n)$ bits memory per node, where E_a is the number of edges in the affected subgraph G' (the set of nodes whose status as garbage is uncertain as a result of a given edge deletion) and n is the number of nodes of the reference graph G. Moreover, in contrast to previous work on distributed garbage collection, our algorithm does not need centralization, global barriers, or the migration of objects. Apart from the benefits mentioned above, our algorithm handles concurrent mutation (addition and deletion of edges and reclamation of nodes in the reference graph) and provides liveness and safety guarantees by maintaining a property called *isolation*. Theorems 11 and 12 prove that when a collection process works alone (i.e. in isolation), it is guaranteed to collect garbage and not to reclaim it prematurely. Theorem 16 proves that when multiple collection processes interact, our synchronization mechanisms allow each of them to act as if they were working alone, and Theorems 20 and 21 prove the correctness in this setting. To the best of our knowledge, this is the first algorithm for garbage collection in distributed systems that simultaneously achieves such guarantees.

[1] Note, the original algorithm of Brownbridge suffered from premature collection, and subsequent attempts to remedy this problem needed exponential time in certain scenarios [20, 23]. Those limitations were addressed in [3].

Related Work: Prior work on distributed garbage collection is vast; Jones et al. [10] provide overall study of automatic memory management and we discuss here the papers that are closely related to our work. The marking algorithm proposed by Hudak [9] requires a global barrier. All local garbage collectors coordinate to start the marking phase. Once the marking phase is over in all the sites, then the sweeping phase continues. Along with the marking and sweeping overhead, there are consistency issues in tracing based collectors [21].

Ladin and Liskov [13] compute reachability of objects in a highly available centralized service. This algorithm is logically centralized but physically replicated, hence achieving high availability and fault-tolerance. All objects and tables are backed up in stable storage. Clocks are synchronized and message delivery delay is bounded. These assumptions enable the centralized service to build a consistent view of the distributed system. The centralized service registers all the inter-space references and detects garbage using a standard tracing algorithm. This algorithm has scalability issues due to the centralized service. A heuristic-based algorithm by Le Fessant [14] uses the minimal number of inter-node references from a root to identify "garbage suspects" and then verifies the suspects. The algorithm propagates marks from the references to all the reachable objects. A cycle is detected when a remote object receives only its own mark. The algorithm needs tight coupling between local collectors and time complexity for the collection of cycles is not analyzed. Gog et al. [8] introduced region-based memory management in big data environments where each region contains a collection of similar objects with well-defined lifetimes. This region-based collection reduces the overhead of collection while offloading the decision about the choice of region for a particular object to the programmer.

Similar to the present work, Clebsch et al. [6] proposed an algorithm for concurrent garbage collection based on the actor paradigm. This algorithm identifies cyclic garbage using a dedicated actor for detecting cyles who has a global view of the system by collecting local snapshots from all the actors. This technique introduces a message size overhead that does not fit within the $\mathcal{CONGEST}$ model (since all messages cannot be bounded to $O(\log n)$ size). Even apart from message size, the centralized cycle detector will create performance bottleneck. and heavy load on central component.

Collecting distributed garbage cycles by backtracking is proposed by Maheshwari and Liskov [17]. This algorithm first hypothesizes that objects are dead and then tries to backtrack and prove whether that is true. The algorithm needs to use an additional data structure to store all inter-site references. An alternative approach of distributed garbage collection by migrating objects was proposed by Maheshwari and Liskov [16]. Both of the algorithms use heuristics to detect garbage. The former one uses more memory, the latter one increases the overhead by moving the objects between the sites. Recently, Veiga et al. [25] proposed an algorithm that uses asynchronous local snapshots to identify global garbage. Cycle detection messages (CDM) traverse the reference graph and gain information about the graph. A drawback of this approach is that the algorithm doesn't work with the $\mathcal{CONGEST}$ model. Any change to the snapshots has to be updated by local mutators, forcing current global garbage collection to quit. For a thorough understanding of the literature, we recommend reading [1, 21].

It is shown in [2, 24] that there is a strong relation between distributed termination detection and distributed garabge collection. Tel et al. have shown that a distributed termination detection algorithm (DTA) can be derived from any distributed garabge collector [24]. Blackburn et al. described several distributed garbage collectors by treating the DTA and the garbage collector separately [2]. This work follows that general methodology by making repeated use of a marking algorithm beginning at the node initiating a collection operation. We again make use of a DTA when reconciling multiple collection operations by ensuring that their interactions are eventually acyclic (see Lemma 13).

Paper Outline: Section 2 describes the garbage collection problem, constraints involved in the problem, and the model for which the algorithm is presented. Section 3 explains the single collector version of the algorithm (SWPR-1) and provides correctness, time, and space guarantees. Section 4 extends the results of Section 3 to the multiple collector scenario. Section 5 presents the test results of the algorithm through a simulator implemented in Java. Finally, Section 6 concludes the paper with a short discussion. The pseudocode for the algorithms may be found in the Appendix.

2 Model and Preliminaries

Distributed System: We consider a distributed system of nodes where each node operates independently and communicates with other nodes through message passing. The underlying topology of the system is assumed to be arbitrary but connected. Each node has a queue to receive messages, and in response to a message a node can only read and write its own state and send additional messages. We further assume that the nodes can communicate over a reliable reordering channel, i.e. that messages are never lost or duplicated. These properties make our system compatible with the $\mathcal{CONGEST}$ asynchronous network model with no failures [19].

Basic Reference Graph: We model the relationship among various objects and pointers in memory through a directed graph $G = (V, E)$, which we call a *reference graph*. The graph G has (at least) a special node R, which we call the *root*. Node R represents global and stack pointer variables, and thus does not have any incoming edges. Each node in G is assumed to contain a unique ID. All adjacent nodes to a given node in G are called *neighbors*, and denoted by Γ. The *in-neighbors* of a node $x \in G$ can be defined as the set of nodes whose outgoing edges are incident on x, represented by $\Gamma_{in}(x)$. The *out-neighbors* of x can be defined as the set of nodes whose incoming edges originate on x, represented by $\Gamma_{out}(x)$. Note that each node $x \in G$ does not know $\Gamma_{in}(x)$ at any point in time.

Distributed Garbage Collection Problem: All nodes in G can be classified as either *live* (i.e., not garbage) or *dead* (i.e., garbage) based on a property called *reachability*. Live and dead nodes can be defined as below:

$Reachable(y, x) = x \in \Gamma_{\text{out}}(y) \vee (x \in \Gamma_{\text{out}}(z) \mid Reachable(y, z))$

$Live(x) = Reachable(R, x)$

$Dead(x) = \neg Live(x)$

We allow the live portion of G, denoted as G', to be mutated while the algorithm is running, and we refer to the source of these mutations as the *Mutator*. The Mutator can create nodes and attach them to G', create new edges between existing nodes of G', or delete edges (and thereby reclaim nodes) from G'. Moreover, the Mutator can perform multiple events (creation and deletion of edges) simultaneously. The Mutator, however, can never mutate the dead portion of the graph $G'' = G \backslash G'$.

Axiom 1 (Immutable Dead Node)**.** *The Mutator cannot mutate a dead node.*

Axiom 2 (Node Creation)**.** *All nodes are live when they are created by the Mutator.*

From Axiom 1, it follows that a node that becomes dead will never become live again.

Each node experiencing deletion of an incoming edge has to determine whether it is still live. If a node detects that it is dead then it must be removed from the graph G.

Definition 1 (Distributed Garbage Collection Problem)**.** *Identify the dead nodes in the reference graph G and reclaim them.*

Classification of Edges in G: We classify the edges in G as *weak* or *strong* according to the Brownbridge method [4]. This classification does not need preprocessing of G and can be done directly at the time of creation of G. Chains of strong edges connect the root R to all the nodes of G and contain no cycles. The weak edges are required for all cycle-creating edges in G. The advantage of the Brownbridge method is that it permits less frequent tracing of the outgoing edges. Without a distinction between strong and weak edges, tracing is required after every edge deletion that does not

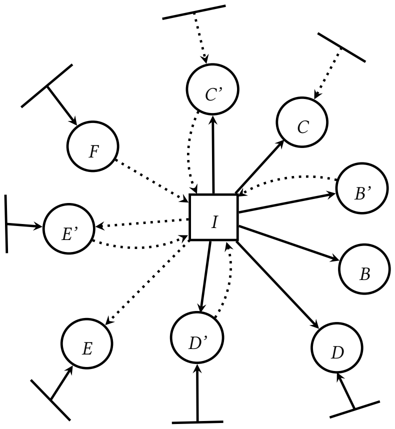


Figure 1. We depict all the ways in which an initiator node, I, can be connected to the graph at the beginning of a garbage collection trace. Circles represent sets of nodes. Dotted lines represent one or more non-strong paths (i.e. paths which contain at least one weak edge). Solid lines represent one or more strong paths. A T-shaped end-point indicates the root, R. If C', D', E' and F are empty sets, I is garbage, otherwise it is not.

Set Name	Nodes in Set	Comment
Purely Dependent	B B'	if I is garbage, so this set
Dependent	$B\ B'\ C\ C'$	Will phantomize
Supporting	$C'\ D'\ E'\ F$	If this set is not empty, I is not garbage
Independent	$D\ D'\ E\ E'\ F$	Will not phantomize
Build	$D'\ E'\ F$	If this set is not empty, recover is not necessary
Recovery	C'	The intersection of the dependent and supporting sets
Affected	$B\ B'\ C\ C'$ $D\ D'\ E\ E'$	

Figure 2. A summary of how nodes related to the initiator. See Fig. 1

bring the reference count to zero. In the Brownbridge method, if a strong edge remains after the deletion of any edge, the node is live and no tracing is necessary. Fig. 1 illustrates an example reference graph G with strong (solid) and weak (dotted) edges.

In this work, we make use of a third kind of edge in addition to strong and weak: the *phantom* edge. A phantom edge is in a temporary and indeterminate state, neither strong nor weak, that the garbage collection algorithm uses while trying to determine whether the edge is strong, weak, or garbage.

Weight-based Edge Classification for the Reference Graph G***:*** We assign *weights* to the nodes as a means of classifying the edges in G. Each node maintains two attributes *weight* and *max-weight*, as well as counts of strong and weak incoming edges. The weight of the root, R, is zero. For an edge to be strong, the weight of the source must be less than the weight of the target. The *max-weight* of a node is defined as the maximum weight of the node's *in-neighbors*. Messages sent by nodes contain the *weight* of the sender so that the *out-neighbors* can keep track of their strong count, weak count, and *max-weight* . When a node is created, its initial weight is given by adding one to the weight of the first node to reference it.

Lemma 1 (Strong Cycle Invariant)**.** *No cycle of strong edges will be formed by weight-based edge classification.*

Proof. By construction, for any node y which is reachable from a node x by strong edges, y must have a larger weight than x. Therefore, if x is reachable from itself through a cycle, and if that cycle is strong, x must have a greater weight than itself, which is impossible. □

Terminology: A *path* is an ordered set of edges such that the destination of each edge is the origin of the next. A path is called *strong* if it consists exclusively of strong edges. If it contains any weak edges, it is called *non-strong*. We say two nodes are *related* if a path exists between them.

A node, x, is a member of the *dependent set* of I if all strong paths leading to x originate on I (i.e. B, B', C, and C' in Fig. 1).

The *purely dependent set* of I consists of the nodes that have no path from R (i.e. B, and B' in Fig. 1). If I is garbage, then so are its purely dependent nodes.

A node, y, is a member of the *supporting set* of I if there is a path from R to y, and from y to I (i.e. C', D', E', and F in Fig. 1). If the supporting set is empty, then I is garbage, otherwise it is not.

A node, z, is a member of the *independent set* of I if z is related to I and there is at least one strong path from R to z (i.e. D, D', E, E', and F in Fig. 1).

A node, x, is a member of the *build set* of I if it is both a member of the supporting set and the independent set. Equivalently, a node, x is said to be in the *build set* if a strong path from R to x exists, and a non-strong path from x to I exists (i.e. D', E' and F in Fig. 1).

A node, x, is a member of the *recovery set* of I if it is both a member of the supporting set and the dependent set (i.e. C' in Fig. 1). During the process of phantomization, the dependent set will phantomize.

A node, x, is said to be in the *affected set* if there exists a path, p, from I to x, such that no node found along p (other than x itself) has a strong path from R (i.e. everything except F in Fig. 1).

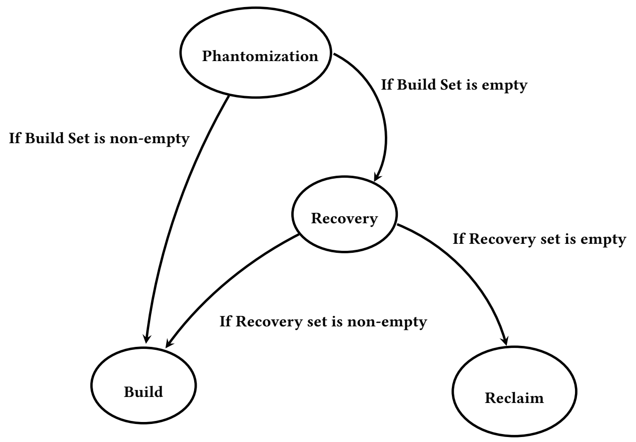


Figure 3. An illustration of the phase transitions performed by an initiator in the algorithm.

These sets are summarized in Fig. 2 for an example reference graph G given in Fig. 1.

3 Single Collector Algorithm (SWPR-1)

For the single collector, we assume that every mutation in G is serialized and that the Mutator does not create or delete edges in G during a *collection process* (i.e. the sequence of messages and state changes needed to determine the liveness of a node). When the last strong edge to a node, x, in G is deleted, but $\Gamma_{\text{in}}(x)$ is not empty, x becomes an *initiator*. The initiator starts a set of graph traversals which we call phases: *phantomization*, *recovery*, *building*, and *reclaiming*. We classify the latter three phases (*recovery*, *building*, and *reclaiming*) as *correction* phases. Fig. 3 provides a flow diagram of the phases of an initiator and Algorithm 1 provides a highlevel pseudocode of how the SWPR-1 algorithm works.

As illustrated in Fig. 1, a node $I \in G$ is dead if and only if its supporting set is empty. If I is discovered to be dead, then its purely dependent set is also dead. Even when the supporting set does not intersect the affected set (i.e. when C', D' and E' are empty), and thus none of the nodes supporting the initiator can receive messages from the initiator, our algorithm will still detect whether the supporting set is empty.

3.1 Phantomization Phase

For a node of G to become a phantom node, the node must (1) have just lost its last strong edge, and (2) has at least one weak edge, (3) not already be a phantom node. The initiator node is, therefore, always a phantom node, and phantomization always begins with the initiator. When a node w becomes a phantom node, it (1) *toggles*, i.e. it increases its *weight* to *maxweight* + 1, thus converting all its incoming weak edges to strong edges; and (2) it sends *phantomize* messages to all nodes in $\Gamma_{\text{out}}(w)$. From then on, all the outgoing edges from

Algorithm 1: single collector algorithm for a node $x \in G$. Required variables on each node: weight, max weight, strong count, weak count, and phantom count, wait count, parent pointer, and state $\in$ $(healthy, phantom, recover, build, dead)$

```
if no incoming strong edge to x ∧ $\Gamma_{in}(x) \neq \emptyset$ ∧ x is not an initiator
∧ x is not a phantom node then
    x becomes both an initiator and phantom node;
    $x.phase \leftarrow phantomizing$; send all the nodes in $\Gamma_{out}(x)$ a
    phantomize message and label all outgoing edges phantom;
if x is not a phantom node ∧ x receives a phantomize message then
    if no incoming strong edge to x then
        x becomes a phantom node (but not an initiator);
        $x.phase \leftarrow phantomizing$; send all the nodes in $\Gamma_{out}(x)$
        a phantomize message and label all outgoing edges
        phantom;
    else
        send a return message to the node it received
        phantomize message from;
if x is not an initiator ∧ $x.phase == phantomizing$ ∧ a
return message from all the nodes in $\Gamma_{out}(x)$ is received then
    send a return message to the node it received phantomize
    message from;
if x is an initiator ∧ $x.phase == phantomizing$ ∧ x received a
return message from all the nodes in $\Gamma_{out}(x)$ then
    if build set is not empty then
        $x.phase \leftarrow building$; send a build message to all the
        nodes in $\Gamma_{out}(x)$;
    else
        $x.phase \leftarrow recovering$; send a recover message to all
        the nodes in $\Gamma_{out}(x)$;
if x is not an initiator ∧ x receives a build message from y then
    if x is a phantom node then
        $x.phase \leftarrow building$; perform toggling to convert
        phantom edges to strong or weak edges; decrement the
        phantom count, increment the strong count; send a
        build message to all the nodes in $\Gamma_{out}(x)$;
    else
        perform toggling, if necessary; decrement the phantom
        count, increment either the strong count or the weak
        count depending on whether $y.weight < x.weight$; send
        a return message to the node it received build message
        from;
if x is not an initiator ∧ x receives a recover message then
    if strong count is zero then
        $x.phase \leftarrow recovering$; send recover message to
        $\Gamma_{out}(x)$;
    else
        $x.phase \leftarrow building$; send build message to $\Gamma_{out}(x)$;
if x is an initiator ∧ $x.phase == recovering$ ∧ x received a
return message from all the nodes in $\Gamma_{out}(x)$ ∧ strong count is
zero then
    send reclaim message to $\Gamma_{out}(x)$;
if x is not an initiator ∧ x receives a reclaim message then
    decrement the phantom count;
    if x has no incoming strong edge then
        send reclaim message to $\Gamma_{out}(x)$;
if all phantom, strong, weak counts are zero ∧ x is not waiting for
any return message then
    reclaim x;
```

w will be considered to be *phantom edges*, i.e. neither strong nor weak but a transient and indeterminate state. If a node, u, in $\Gamma_{\text{out}}(w)$ loses its last strong edge as a result of receiving a *phantomize* message, u also becomes phantom node, but not an initiator. Phantomization will thus mark all nodes in the dependent set. In addition, all the nodes in the affected set will receive phantomize messages.

Since the algorithm proceeds in phases, we need to wait for phantomization to complete before recovery or building can begin. For this reason, for every *phantomize* message sent from a node, x, to a node, y, a *return* message must be received by x from y . If y does not phantomize as a result of receiving the message from x, the *return* message is sent back to x immediately. If the *phantomize* message causes y to phantomize, then y waits until it has received *return* back from all nodes in $\Gamma_{\text{out}}(y)$ before replying to x with a *return* message. For this purpose, each phantom node maintains a single backward-pointing edge and a counter to track the number of *return* messages it is waiting for. While y is waiting for *return* messages, it is said to be *phantomizing*. Once all *return* messages are received, the phantom node is said to be *phantomized*.

Phantomization is, therefore, essentially, a breadth-first search rooted at the initiator. The traversal contains two steps. In the forward step, messages originate from the initiator and propagate to the affected nodes. After they reach all the affected nodes, *return* messages will propagate backward to the initiator. The reverse step is a simple termination detection process which uses a spanning tree in the subgraph comprised of the affected nodes (i.e., the single backward-pointing edge forms the spanning tree).

The outgoing edges of any phantom node are said to be *phantom edges*. Each node has a counter for its incoming phantom edges (in addition to its strong and weak counters). Every node in the affected subgraph will have a positive phantom count at the end of the phantomization phase. When phantomization is complete, the initiator then enters into the correction phase.

3.2 Correction Phase

When phantomization is complete, the initiator starts either the *recovery* or *building* phase depending on the build set.

If the build set is not empty, the initiator enters the build phase. In the build phase, the affected subgraph is traversed, phantom edges are converted to strong and weak edges, and the weights of the nodes are adjusted.

If the build set is empty, the initiator enters the recovery phase. In the recovery phase, the affected subgraph is traced until the recovery set is found (i.e. phantom nodes that have strong incoming edges). If and when this set is found, the build phase begins from the nodes in the recovery set. If the recovery is empty, then the initiator will reclaim all the purely dependent nodes.

Both build and recovery phases make use of wait counts, return messages, and the backward pointing edge in the same manner as phantomization.

Build Phase: If the initiator has any strong incoming edges after the phantomization phase, then the build set is not empty. In response, the initiator updates its phase to *building* and sends *build* messages to its *out-neighbors*.

If a node, y, sends a *build* message to a node, x, that is phantom node, the *max-weight* of x is set to the weight of y ($x.\textit{max-weight} \leftarrow y.\textit{weight}$), and the weight of x is set to the weight of y plus one ($x.\textit{weight} \leftarrow y.\textit{weight} + 1$). The node x then decrements the phantom count, increments the strong count, and propagates the *build* message to its *out-neighbors*.

If a node, y, sends a *build* message to a node, x, that is not phantom node, it *builds* the edge, i.e. it updates the *max-weight* of x if necessary, decrements the phantom count and increments either the strong or the weak count, depending on its weight.

After a node, x, builds its last phantom edge and replies to its parent in the spanning tree with *return*, the node x is then returned to a *healthy* state.

Lemma 2 (Dependent Phantomizes). *During phantomization, only the dependent set phantomizes.*

Proof. We only consider situations where I loses its last strong edge but still has incoming weak edges, as other cases are trivial. By definition, phantomization converts the outgoing edges of I to phantom edges. Since this takes away the last strong edge of all nodes in the dependent set, these nodes and their outgoing edges also phantomize. Since nodes outside the dependent set have strong paths from the root, they do not phantomize. □

Lemma 3 (Not Phantomized). *After phantomization, nodes in the build set are not phantom nodes.*

Proof. By Lemma 2, since the build set has no intersection with the dependent set, it does not phantomize. □

Lemma 4 (Phantomized). *After phantomization, nodes in the recovery set are phantom nodes.*

Proof. By Lemma 2, since the recovery set is a subset of the dependent set, it will phantomize. □

Lemma 5 (Build set). *After phantomization, if the initiator has strong incoming edges, then the initiator is not empty.*

Proof. From Lemma 3, we know that nodes in the build set can keep the initiator, I, alive. When the initiator became a phantom node, it converted its weak incoming edges to strong. However, since nodes in the recovery set all became phantom nodes by Lemma 4, they will not contribute strong support to the initiator. Since nodes in the build set do not phantomize by Lemma 3, the edges connecting them to the initiator remain strong. □

Lemma 6 (Recovery set). *After phantomization, if a phantom node contains at least one strong incoming edge, it belongs to the recovery set.*

Proof. Every node in the recovery set will phantomize by Lemma 4. Also, every node in the recovery set has a non-strong path from R prior to phantomization. Phantomization, however, will cause the nodes to toggle, converting the non-strong path from R to a strong path for at least one node in the recovery set. Therefore, the recovery set will contain at least one node with at least one strong incoming edge. □

Lemma 7 (Non Empty Recovery Set). *After phantomization, if the recovery set is not empty, then the initiator is not dead.*

Proof. The recovery set is an intersection of dependent set and supporting set. Since the dependent set's strong paths come from the initiator, it is phantomized by the phantomization process. When a node is a member of the supporting set and the dependent set, it means that node has an alternative path from root to node that does include the initiator. That alternative path will become strong after phantomization. □

Recovery Phase: If the initiator has no strong edges after the phantomization phase, then the build set is empty and it updates its phase to *recovering* and sends *recover* messages to its *out-neighbors*.

When a node receives a *recover* message and its strong count is zero, it simply updates its state to *recovering* and propagates the *recovery* message to *out-neighbors*. If, instead, it has a positive strong count, it propogates *build* to its *out-neighbors* .

Note that it is possible for a build message to reach a node, x, that is still in the forward step of the recovery process. To accommodate this, when x receives all its *return* messages back from $\Gamma_{\text{out}}(x)$, x checks to see if it now belongs to the recovery set (i.e., if it has positive strong count). If it still does not belong to the recovery set, then the return message is sent back to the parent as usual. If it has a positive strong count, the node starts the build process. Once the phantom count is zero, the strong count is positive, and the node is no longer waiting for return messages, it is marked healthy.

Reclaim Phase: If the recovery phase finishes and the initiator, x, still has no incoming strong edge, it issues *plague* messages to $\Gamma_{\text{out}}(x)$. As the name suggests, this message is contagious and deadly. Any node receiving it decrements its phantom count by one and (if the recipient has no incoming strong edges) it sends the *plague* message along its outgoing edges.

Once a node has no edges (i.e. phantom, strong, and weak count are zero) and is no longer waiting for return messages, it is reclaimed. Unlike the other phases, return messages and wait counts are not used in *plague* messages.

Lemma 8 (Time Complexity). *SWPR-1 finishes in $O(Rad(i, G_a))$ time, where G_a is the graph induced by affected nodes, where i is the initiator, and $Rad(i, G_a)$ is the radius of the graph from i.*

Proof. In each time step, *phantomization* spreads to the *out-neighbors* of the previously affected nodes, increasing the radius of the graph of phantom nodes by 1. In O(Rad(i,G_a)) time, all the nodes in G_a receive a *phantomize* message, since all the nodes in G_a are at distance less than or equal to Rad(i,G_a) from i. In the reverse step, the same argument can be applied backward. So phantomization completes in O(Rad(i,G_a) time.

Note that this calculation is an ideal bound as it assumes there are more computational units than nodes.

In the correction phase, during the forward step, in r time, r neighborhoods of i received *recovery* or *build* or *plague* message, until the affected subgraph is traversed. In the reverse step of *build* or *recovery*, however, a *return* message might initiate the build process. While the build process nodes send return messages, the nodes will become healthy thereby reducing the Rad(i,G_a). So in the worst case, all nodes that received recovery might build. Because each node will have only one parent, the return step cannot take more than O(Rad(i, G_a) time. □

Corollary 1. *SWPR-1 finishes in $O(E_a)$ time, since Rad(i, G_a) can be O(E_a).*

Lemma 9 (Communication Complexity). *SWPR-1 sends O(E_a) messages, where E_a is the number of edges in the graph induced by affected nodes.*

Proof. In the forward step of the phantomization, all the nodes in the dependent set send the *phantomize* message to their *out-neighbors*, and each node can do this at most once (after which the phantom node flag is set). So the forward step of the algorithm uses only O(E_a) messages. In the reverse step, the *return* messages are sent backward along the edges of the spanning tree. So the reverse step sends O(V_a) messages, where V_a is the number of nodes in the affected subgraph. So Phantomization uses O(E_a) messages.

In the forward step of the recovery/build, either a *recovery* message, a *build* message, or both traverses every edge, so the forward step of the algorithm uses O(E_a) messages. In the reverse step of the algorithm, the *return* messages are sent back to the parent a maximum of two times (once for recovery, once for build), traversing a subset of the edges in the reverse direction. In every forward step of the plague, all outgoing edges are consumed, and therefore it takes O(E_a) messages. □

Lemma 10 (Message Size Complexity). *SWPR-1 sends messages of $O(\log(n))$ size, where n is the total number of nodes in the graph.*

Proof. The messages have to hold a value at least as large as the count of nodes in the system which are O(log(n)) size. Apart from the ids, the message also contains the weight of node which is also of size O(log(n)). In the return message, our algorithm only uses the id of the sender and receiver. So all the messages in the SWPR-1 are of O(log(n)) size. □

Theorem 11. *All dead nodes will be reclaimed at the end of the correction phases.*

Proof. Assume a node y, is a dead node, but flagged as live node by the correction phase. If y becomes live, then it must have done so because its edges were rebuilt during building. If so, then either y has a strong count, or there exists a node x with a strong count which also has a path to y. However, any node with a strong count at the end of phantomization is a live node by Lemma 5 and Lemma 6, and because a path from x to y exists, x is also live. This contradicts our assumption. □

Theorem 12. *No live node will be reclaimed at the end of the correction phases.*

Proof. Assume a node, x is live, but it is reclaimed. If x is the initiator and is live then the supporting set is not empty. If the build set isn't empty, then x will have a strong count, so it won't be reclaimed. If the build set is empty and the recovery set is non-empty, then correction will build a strong edge to x, so it won't be reclaimed. This contradicts our assumption. Now assume x is dead, but it causes some other node to be reclaimed. If x is dead, then the supporting set is empty and the purely dependent set is dead. The independent set has a strong count before the collection process, so it won't be reclaimed. The nodes in the dependent set that had a non-strong path from R before phantomization, will have a strong path from R after toggling and recovery/build, so they will not be reclaimed. This also contradicts our assumption. □

4 Multi-Collector Algorithm (SWPR)

We now discuss the case where mutations in G are not serialized and the Mutator is allowed to create and delete edges in G during the collection process. As a consequence, multiple edge deletions might create collection processes with overlapping (induced) subgraphs, possibly trying to perform different phases of the algorithm at the same time.

To break the symmetry among multiple collectors, we assign unique identifiers to each one. The collection identifier (CID) is a tuple consisting of three numbers, a *major id*, an *object id* (which uniquely identifies an object), and a *minor id* $(I_{major}, I_{object}, I_{minor})$. Tuples are ordered in a lexicographical order first by major id, then if major id's are the same, object id's, then if both major and object id's are the same, they are ordered by minor id. Initially, both the major and minor id are zero.

In addition, to help a node keep track of what phase it should be in, nodes remember the message which caused their backward-pointing edge (i.e. parent edge) to be set.

Appendix A contains the detailed pseudocode of the multi-collector algorithm.

Definition 2 (Isolation). *A collection process is said to proceed in isolation if either (1) the affected subgraph does not mutate, or (2) mutations to the affected subgraph occur, but they do not affect the decisions of the initiator to reclaim or build, and do not affect the decisions of the recovery set to build by the correction phases.*

If there are two or more (collection) processes acting on the same shared subgraph, the collection process with the higher id (higher priority) will acquire ownership of the nodes (of the shared subgraph) upon receipt of a *phantomize* message. As a consequence, any *initiator* node which receives a *phantomize* message from a higher priority collection marks itself with the higher id and loses its status as an initiator.

In addition, we have a new type of message called the *claim* message which, like *phantomize*, takes ownership of the nodes it touches. If a node that is already phantomized receives a *phantomize* or *claim* message from a higher priority collection, that node sends a *claim* message, to Γ_{out}. The *claim* message propagates through the tree in the same manner as *phantomize*, *recover*, and *build*, by waiting for return messages and sending a *return* back to the parent once all of them have been received. The claim message ensures that the reachable subgraph is under the management of the higher priority collection process.

Collection Process Graph: The set of nodes marked by a CID is called a *collection process.* Each collection process can be represented as a single node, and the set of collection process nodes eventually forms a directed acyclic graph.

Lemma 13. *The collection process graph is eventually acyclic.*

Proof. If a cycle exists between two process graphs, A and B, then recovery or build messages must eventually propagate from one to the other. Without loss of generality, assume A has higher priority than B. During the correction phases, messages propagate along all phantom edges, and in the process take ownership of any nodes they touch. Eventually, therefore, there should be no edges from A to B. □

Recovery Count: The recovery count tracks the number of *recovery* messages each node receives from its own collection process. Thus, a recovery edge is also a phantom edge, and a phantom edge will be converted to a recovery edge when a recover message is sent for that edge with the same CID. The recovery edge is the fourth edge type in the algorithm. The recovery count maintained on each node is stored in a variable named *RCC.*

A recovering node can neither send the *return* message to its parent nor start reclaiming until the recovery count is equal to the phantom count.

The following scenario illustrates the purpose of the recovery count. Consider the configuration of nodes in Fig. 4. Initially, we have root edges connected to both N_6 and N_1 and after these paths are reclaimed we have two collection processes.

Assume that these two collection processes have relative priority $I_6 > I_1$, and I_6 begins its recovery phase after the edge $N_2 \rightarrow N_4$ is phantomized. Process I_6 should identify node N_4 as belonging to the recovery set, but if process I_1 has not completed recovering and building, it will not. Therefore, without the requirement that the recovery count equal the phantom count, the recovery phase for I_6 will complete and prematurely reclaim N_6. With the requirement, I_6 will stall until $N_2 \rightarrow N_4$ is rebuilt. Once that is done, I_6 will rebuild $N_4 \rightarrow N_5$ and $N_5 \rightarrow N_6$. Return messages will then be sent until they get back to N_6, and finally $N_6 \rightarrow N_4$ will rebuild.

The effect of the recovery count requirement, therefore, is to ensure that the higher priority process makes its decision in isolation from the lower priority process. Thus, the higher priority processes will wait until either the lower priority processes is claimed by the higher one (if both are part of the same cycle), or until the lower priority process completes (if there is no path from the higher prioirity process to the the lower priority process).

Note that the recovery count also affects *plague* messages. These messages will not propagate unless the node receiving them has an equal recovery and phantom count.

We assumed in the discussion above that I_6 has higher priority than I_1. If the priority were reversed, I_1 would take over building node N_4, but this introduces additional complexity which we will now address.

Lemma 14. *If an entity A precedes an entity B topologically in the collection process graph, and A has a lower priority than B, entity A will complete before entity B and both will complete in isolation.*

Proof. Consider a node, x, that has incoming edges from both A and B. Process B will have ownership of the node during the recovery or build phase, but until the edges from A are either built or deleted, x will have a recovery count (i.e. RCC) equal to the number of edges from B, and a phantom count (i.e. PC) equal to the sum of the edges from $A + B$. So the recovery or build phase of B must take place after A, and so B will operate in isolation. Since there are no edges from B to A, and since B is not making progress, B does not affect A, and A is in isolation. □

Recovery Transactions: Recovery transactions are a mechanism to restart the recovery in certain special circumstances. Consider a *recover* message, m, arriving at node N_4 from N_2 (Fig. 4). This time, assume that the collection id of I_1 is higher

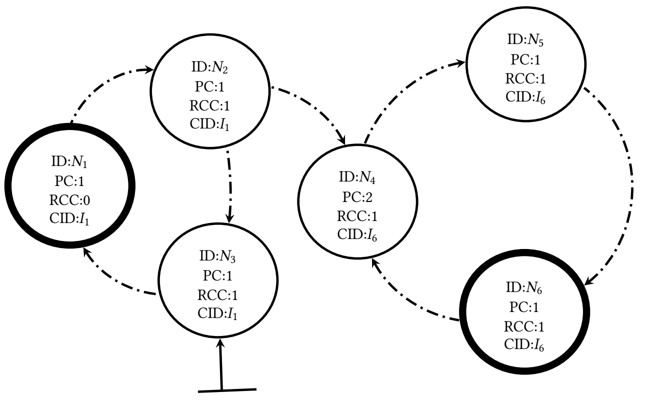


Figure 4. We depict two collection processes in the recovery phase. Each circle is a node. Node properties are: node id (ID), phantom count (PC), recovery count (RCC), collection id (CID). Bold borders denote initiators, and dot and dashed edges denote phantom edges.

than I_6. If N_4 is in the recovered state, the collection id on x is updated with the collection id in m ($x.CID \leftarrow m.CID$). If, however, x is recovering, i.e. it is waiting for *return* messages from its *out-neighbors*, a *return* message, r, is immediately sent to N_4's parent, and N_4's parent is set to point to N_2. In addition, a value called the *start-over CID* or *SOC* on x is set and propagated by return messages until it reaches the initiator. The *SOC* will cause recovery to restart on x once the recovery operation completes. The new recovery phase, however, will have incremented its minor id (the last number in the tuple) by 1.

In the example, the SOC propagates back to N_6, where the recovery count will be reset and the entire recovery phase will start over, but with collection id I'_6. Note that while I'_6 is higher than I_6, it is still lower than I_1. The higher priority (signified by the prime) doesn't change the relative priority of the collection process to the other independent collection processes because it only increments the minor id. After this change, the new recovery phase will not proceed until the phantom count is equal to the recovery count ($PC = RCC$). Since this cannot succeed, I'_6 stalls until I_1 takes over the entire graph.

This restarted recovery triggered by the SOC is similar to the rollback of a transactional system and hence the name.

Handling the actions of The Mutator: Creation of edges and nodes by the Mutator poses no difficulty. If an edge is created that originates on a phantomized node, the edge is created as a phantom edge. If a new node, x, is created, and the first edge to the node comes from a node, y, then x is created in the same state as y (e.g., phantomized, recovered, etc.).

If, however, the Mutator deletes a phantom edge $x \rightarrow y$ from collection x_1 and $\Gamma_{\text{in}}(y)$ is not empty, it is not enough to reduce the phantom count on y by 1. In this case, y is made the initiator for a new collection process, y_2, such that y_2 has higher priority than x_1 (we increment the major id on the tuple). If y is in the recovering state (i.e., it is waiting for *recover* messages to return), it sends *return* with $SRO = true$, and re-recovers y.

Lemma 15. *If an entity A precedes an entity B in the collection process graph C with respect to a topological ordering and A has higher priority than B, entity A will take ownership of entity B, and A will proceed in isolation.*

Proof. Because A has higher priority, it takes ownership of every node it touches and is thus unaware of B. So it proceeds in isolation. If B loses ownership of a recovered node, it will not affect isolation because a recovered node has received *recovery* messages on all its incoming edges ($RCC = PC$), and has already sent its return. If B loses a node that is building or recovering, B is forced to start over. In the new recovery or build phase, B will precede A in the topological order, and both will proceed in isolation. □

Theorem 16. *All collection processes will complete in isolation.*

Proof. By Lemma 13 we know that the collection process graph will eventually be topologically ordered, and the ordering will not violate isolation given by Lemma 15. Once ordered, the collection processes will then complete in order proven by Lemma 14. □

Corollary 17. *In a quiescent state (i.e. one in which the Mutator takes no further actions on G_a), with p active collection processes, our algorithm finishes in $O(Rad_{all})$ time where G_a is the affected subgraphs of all connected collection process and Rad_{all} is the sum of the radii of the affected subgraphs of all p initiators.*

Lemma 18 (Creation Isolation). *Creation of edges and nodes by the Mutator does not violate isolation.*

Proof. The addition of edges cannot affect the isolation of a graph because (1) addition of an edge cannot cause anything to become dead and (2) if an edge is created to or from any edge in a collection process, then the process was already live by Axiom 1. □

Lemma 19 (Deletion Isolation). *Edge deletion by the Mutator does not violate isolation.*

Proof. When the Mutator removes an edge $x \longrightarrow y$ from inside a collection process graph, a new, higher collection process is created and it is given ownership of y. By Theorem 16, the old and new collection process will proceed in isolation. □

Theorem 20 (Liveness). *Eventually, all dead nodes will be reclaimed.*

Proof. By Axiom 1, we know the Mutator cannot create or delete any edges to dead nodes. Theorem 11 proves that if a collection process works in isolation, all dead nodes will be reclaimed. Lemma 16 proves that indeed all collection processes complete in isolation. Theorem 11 proves all collection processes in isolation reclaim dead nodes. Thus, eventually, all dead nodes will be reclaimed. □

Theorem 21 (Safety). *No live node will be reclaimed.*

Proof. Lemma 12 proves that if a collection process works in isolation, no live nodes of G will be reclaimed. Lemma 16 proves that when multiple collection processes working on the same graph proceed in isolation. Lemma 18 proves that creation of edges does not violate isolation. Likewise, Lemma 19 proves that the deletion of an edge in the does not affect isolation. □

5 Testing

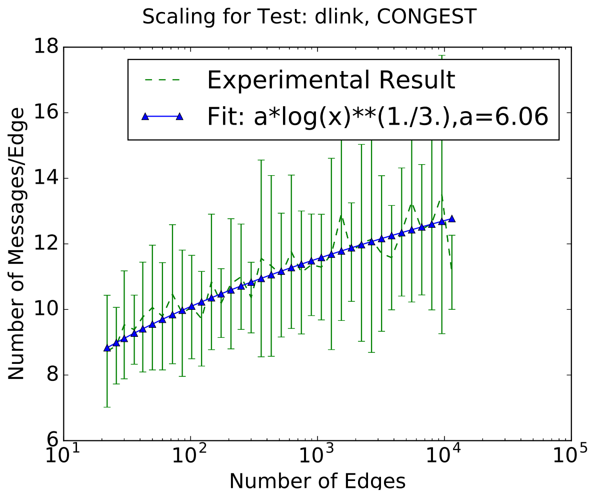
The algorithm presented in this paper was tested using a simulator created in Java. The simulator puts all messages into a common queue, then withdraws them one-at-a-time in random order for execution. We call this "random" mode. This simulates a distributed parallel system with highly varying message delivery times. Alternatively, we used a "CONGEST mode" in which each object with a pending message executes within a simulated time step or round.

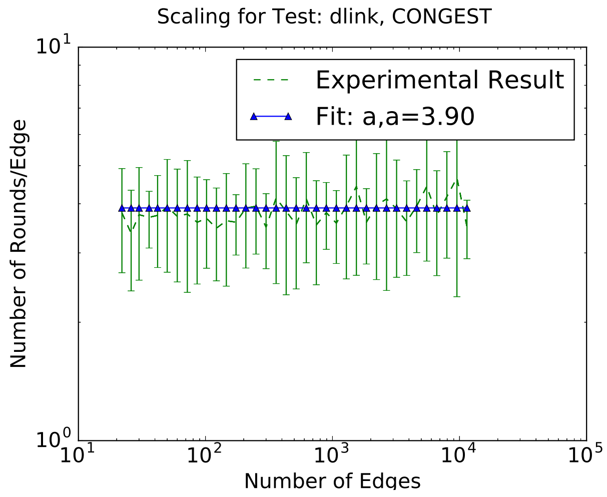
In either case, because the system is actually executing in a random sequence, we can replay any problems created by pathological message arrival times by simply starting the simulation with the same seed.

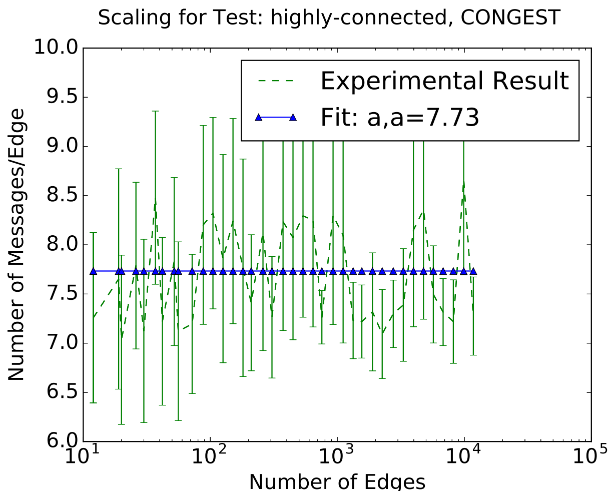
Using this framework, we ran hundreds of thousands of simulations with as many as a million concurrent collection processes. Initial configurations included doubly linked lists, highly connected graphs (including cliques), random graphs, and regular square grids of nodes. These graphs, initially, had root edges connecting to all the nodes and weak edges between the nodes. Once the graph was complete, we removed all root nodes and let garbage collection begin at every node simultaneously.

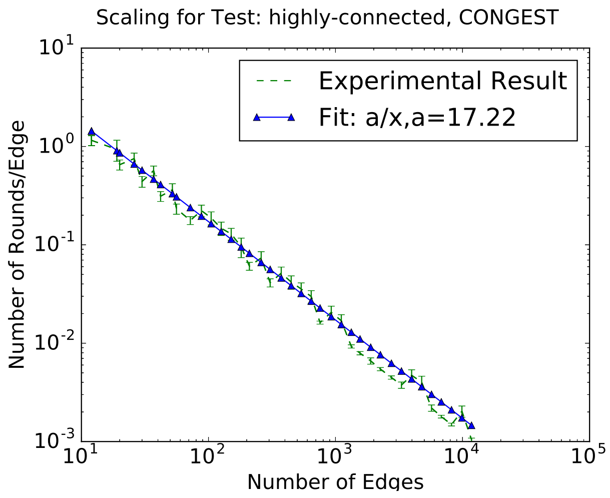
We summarize the empirical results of these experiments below. Additionally, we provide plots with error bars (representing standard deviations) for the cases listed below. In all of what follows, N represents the number of edges in the network.

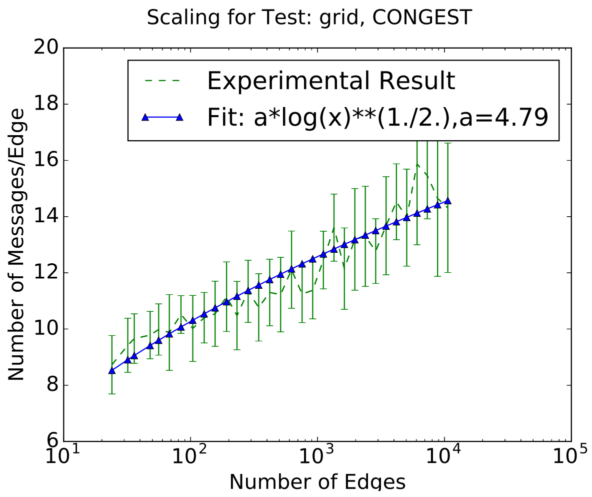
1. The doubly linked list $\propto 6.06 \log(N)^{1/3} N$ messages and $\propto 3.9N$ rounds to complete. The reason for the relatively poor performance in terms of rounds is explained by the low connectivity of the network.
2. The best performing were the highly connected graphs. In this case, the number of messages was $\approx 7.73N$, and the number of rounds was ≈ 17.22. This is explained by the fact that highly connected networks rapidly eliminate competing collections and the single, highest priorty collection can proceed with nearly perfect parallelization.

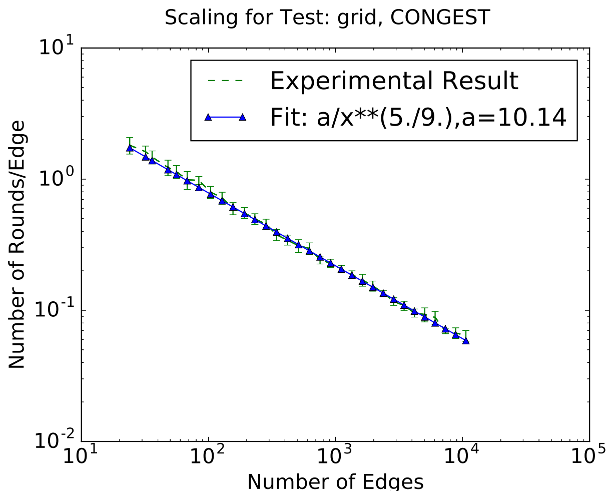












3. The grid required approximately $\approx 4.79 \log(N)^{1/2} N$ messages and $\propto 10.14 N^{4/9}$ rounds to complete

In general, the more connected the dependent graph is, the easier the messages can flow, and the faster (in terms of rounds) the algorithm can proceed toward a state with no phantom edges/nodes.

The test simulator is publicly available at https://github.com/stevenrbrandt/DistributedGarbageCollectorSimulator.git

6 Conclusions

We have described a hybrid reference counting algorithm for garbage collection in distributed systems. The algorithm improves on the previous efforts by eliminating the need for centralization, global barriers, back references for every edge, and object migration. Moreover, it achieves efficient time and space complexity. Furthermore, the algorithm is stable against concurrent mutations in the reference graph.

We have described a test framework used to verify the correctness of the algorithm and summarized the performance of the algorithm under various scenarios.

We believe that our techniques might be of independent interest in solving other fundamental problems that arise in distributed computing.

In future work, we hope to address how to provide an order in which the dead nodes will be cleaned up, permitting some kind of "destructor" to be run, and to address fault tolerance. In addition, we hope to implement the proposed algorithm in the HPX framework [11].

A Algorithms

Required variables on each node:

1. integers: object id, strong count, weak count, phantom count, rcc, wait count, weight, max weight
2. three int tuples: cid, recover cid, start over cid
3. enum with members healthy, phantom, recover, build, infected, dead: state, recv
4. object pointers: parent
5. boolean: incrRCC

```
Procedure OnEdgeCreation(sender weight,is phantom edge,sender cid):
if is phantom edge then
    if cd = NULL then
        cd = new CollectionData(sender cid)
        cd.parent = sender
    Increment cd.phantom count
    if cd.phantom count = 0 and cd.rcc = 0 then
        cd = NULL
else if sender weight < weight then
    Increment strong count
else
    Increment weak count
if sender weight > max weight then
    max weight = sender weight
```

```
Procedure OnEdgeDeletion(sender,sender weight,is phantom edge,sender cid):
if phantom flag then
    if cd = NULL then
        cd = new CollectionData(sender cid)
    Decrement cd.phantom count
    if sender cid.equals(cd.cid) then
        Decrement cd.rcc
else if sender weight < weight then
    Decrement strong count
else
    Decrement weak count
if cd = NULL then
    if strong count > 0 then
        return
    else if weak count > 0 then
        if has no outgoing edges() then
            toggle()
        else
            cd = new CollectionData(id)
            toggle()
            PhantomizeAll(sender)
    else
        delete outgoing edges(NULL)
        cd = new CollectionData(id)
        cd.state = DEAD
else
    return to parent force()
    cd.cid = new CID(cd.cid.majorId + 1, id, 0)
    cd.parent = 0
    if cd.wait count = 0 then
        action(sender)
```

```
Procedure Toggle():
if weak count > 0 and strong count = 0 then
    strong count = weak count
    weak count = 0
    weight = max weight + 1
```

```
Procedure Phantomize(sender,sender cid):
if w < weight then
    Decrement strong count
else
    Decrement weak count
do action = cidCheck(sender, sender cid)
if parent was set then
    cd.recv = PHANTOM
Increment cd.phantom count
if do action then
    if not is initiator() then
        action(sender)
return to sender(sender)
if cd.phantom count = 0 and strong count = 0 and weak count = 0 and cd.wait count = 0 then
    delete outgoing edges(NULL)
    cd.state = DEAD
```

```
Procedure PhantomizeAll(sender weight,is phantom edge,sender cid):
if phantom flag() then
    ClaimAll()
    return
cd.state = PHANTOM
for each edge do
    Send Message Phantomize → edge
    Increment cd.wait count
if cd.wait count = 0 then
    cd.state = HEALTHY
    return to parent()
    if strong count > 0 and cd.phantom count = 0 and cd.rcc = 0 then
        return to sender(sender)
        cd = NULL
    else
        cd.incrRCC = FALSE
else
    cd.incrRCC = FALSE
```

```
Procedure Claim(sender,sender cid):
do action = cidCheck(sender, sender cid)
if parent was set then
    cd.recv = PHANTOM
if do action then
    if not is initiator() then
        action(sender)
return to sender(sender)
```

```
Procedure ClaimAll():
cd.state = PHANTOM
for each edge do
    Send Message Claim → edge
    Increment cd.wait count
if cd.wait count = 0 then
    return to parent()
cd.incrRCC = FALSE
```

```
Procedure Recover(sender cid,sender,incrRCC,mandate):
incr = cd.cid < sender cid
do action = cidCheck(sender, sender cid)
if parent was set then
    cd.recv = RECOVER
if incrRCC and sender cid = cd.cid and cd.phantom count > 0 then
    Increment cd.rcc
if do action then
    cd.incrRCC = FALSE
    if not is initiator() then
        action(sender)
return to sender(sender)
```

```
Procedure RecoverAll():
if cd.incrRCC then
    ClaimAll()
    return
incrRCC = not (cd.recoverCid = cd.cid)
cd.state = RECOVER
cd.recoverCid = cd.cid
for each edge do
    Increment cd.wait count
    Send Message Recover → edge
cd.incrRCC = FALSE
if ready() then
    return to parent()
```

```
Procedure Build(sender weight,sender cid,sender,decrRCC,mandate):
if cd ≠ NULL and phantom flag() and strong count = 0 and weak count = 0 then
    weight = sender weight + 1
    max weight = sender weight
if sender weight < weight then
    Increment strong count
else
    Increment weak count
if sender weight > max weight then
    max weight = sender weight
Decrement cd.phantom count
do action = cidCheck(sender, sender cid)
if parent was set then
    cd.recv = BUILD
if in collection message and decrRCC then
    Decrement cd.rcc
if ready() and cd.parent ≥ 0 then
    action(sender)
    return to sender(sender)
    return
if do action then
    if not is initiator() then
        action(sender)
return to sender(sender)
```

```
Procedure BuildAll(sender):
if not phantom flag() then
    if not is initiator() then
        return to parent()
    return
decrRCC = cd.recoverCid = cd.cid
cd.recoverCid = NULL
cd.state = BUILD
for each edge do
    Increment cd.wait count
    Send Message Build → edge
cd.incrRCC = FALSE
if ready() then
    return to parent()
    return to sender(sender)
    cd = NULL
```

```
Procedure plague(sender,sender cid):
Decrement cd.phantom count
if sender cid = cd.cid then
    Decrement cd.rcc
if cd.phantom count = cd.rcc and cd.wait count = 0 then
    if strong count = 0 and weak count = 0 then
        InfectAll()
        return
    else if cd.phantom count = 0 then
        cd = NULL
```

```
Procedure InfectAll():
for each edge
Send Message Plague → edge
set edge to NULL
edges.clear()
if strong count = 0 and weak count = 0 and cd.phantom count = 0 and cd.wait
count = 0 then
    cd.state = DEAD
else
    cd.state = INFECTED
```

```
Procedure Return(start over cid):
Decrement cd.wait count
if start over cid ≠ NULL then
    if cd.start over = NULL or cd.start over < start over cid then
        cd.start over = start over cid
    if is initiator() and start over cid = cd.cid then
        mycid = cd.cid
        cd.cid = new CID(mycid.majorId, mycid.objId, mycid.minorId + 1)
if cd.wait count = 0 then
    if start over cid ≠ NULL and is initiator() then
        if not start over cid = cd.cid then
            start over cid = NULL
        c = cd.cid
        cd.cid = new CID(c.majorId, c.objId, c.minorId + 1)
        if cd.phantom count = 0 and strong count > 0 and ready() then
            if phantom flag() then
                BuildAll(-1)
            else
                return to sender(-1)
                cd = NULL
        else if phantom flag() then
            RecoverAll()
        else if strong count > 0 then
            BuildAll(-1)
        else
            toggle()
            PhantomizeAll(-1)
    else
        action(-1)
else
    mycid = cd.cid
    if mycid = start over cid then
        if is initiator() then
            cd.cid = new CID(mycid.majorId, mycid.objId, mycid.minorId
            + 1)
            cd.parent = 0
            if cd.wait count = 0 then
                action(-1)
```

```
Procedure cidCheck():
if cd = NULL then
    if sender cid = NULL then
        sender cid = new CID(0, this.id, 0)
    cd = new CollectionData(sender cid)
    cd.parent = sender
    parent was set = TRUE
else if sender = 0 then
    pass
else if sender cid = cd.cid then
    in collection message = TRUE
    if cd.wait count > 0 then
        return FALSE
    if cd.parent < 0 then
        cd.parent = sender
        parent was set = TRUE
else if sender cid < cd.cid then
    return FALSE
else if cd.wait count > 0 then
    if not is initiator() then
        return to parent force(cd.cid)
    cd.parent = sender
    parent was set = TRUE
    cd.cid = sender cid
    cd.start over = NULL
    if cd.state = RECOVER then
        cd.state = PHANTOM
    return FALSE
else if cd.cid < sender cid then
    if cd.parent > 0 then
        return to parent force(cd.cid)
    else
        cd.start over = NULL
    cd.parent = sender
    parent was set = TRUE
    cd.cid = sender cid
return TRUE
```

```
Procedure Ready(sender weight,is phantom edge,sender cid):
if cd = NULL then
    return TRUE
if cd.wait count > 0 then
    return FALSE
if cd.rcc < cd.phantom count then
    if is initiator() then
        if cd.state = RECOVER or BUILD then
            return FALSE
    else if cd.recv = RECOVER or BUILD then
        return FALSE
return TRUE
```

```
Procedure actionInitiator(sender):
if ready() then
    if cd.state = HEALTHY then
        if not phantom flag() and old weight < weight or strong count = 0 then
            PhantomizeAll(sender)
    else if cd.state = PHANTOM then
        if strong count > 0 then
            BuildAll(sender)
        else if has no outgoing edges() and weak count = 0 and cd.phantom count = 0 then
            cd.state = DEAD
        else if weak count = 0 then
            RecoverAll()
    else if cd.state = RECOVER then
        if strong count = 0 then
            InfectAll()
        else
            BuildAll(sender)
    else if cd.state = INFECTED then
        if cd.phantom count = 0 and strong count = 0 and weak count = 0 then
            cd.state = DEAD
    else if cd.phantom count = 0 then
        return to sender(sender)
        if strong count = 0 and weak count = 0 then
            PhantomizeAll(sender)
        else
            cd = NULL
else if phantom flag() then
    if cd.incrRCC then
        ClaimAll()
else if strong count = 0 then
    PhantomizeAll(sender)
else if not phantom flag() then
    if strong count = 0 then
        PhantomizeAll(sender)
```

Acknowledgements

We wish to acknowledge the following grants in support of this work: NSF 1737785

References

[1] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, September 1998.

[2] Stephen M. Blackburn, Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, David S. Munro, and John Zigman. Starting with termination: A methodology for building distributed garbage collection algorithms. *Aust. Comput. Sci. Commun.*, 23(1):20–28, January 2001.

[3] Steven R. Brandt, Hari Krishnan, Gokarna Sharma, and Costas Busch. Concurrent, parallel garbage collection in linear time. In *Proceedings of the 13th International Symposium on Memory Management*. ACM, 2014.

[4] D.R. Brownbridge. Cyclic reference counting for combinator machines. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 273–288. 1985.

```
Procedure action(sender):
if cd = NULL then
    return
toggle()
if old weight < weight then
    PhantomizeAll(sender)
else if is initiator() then
    action initiator(sender)
else if cd.recv = HEALTHY then
    if cd.phantom count = 0 and cd.rcc = 0 and cd.wait count = 0 then
        return to sender(sender)
        cd = NULL
else if cd.recv = PHANTOM then
    if not phantom flag() and (old weight < weight or strong count = 0) then
        PhantomizeAll(sender)
    else if cd.incrRCC and phantom flag() then
        ClaimAll()
    else
        return to parent()
else if cd.recv = RECOVER then
    if cd.state ≠ RECOVER then
        if strong count > 0 then
            BuildAll(sender)
        else if phantom flag() then
            RecoverAll()
        else
            PhantomizeAll(sender)
    else if strong count > 0 then
        BuildAll(sender)
    else if weak count > 0 then
        PhantomizeAll(sender)
    else
        return to parent()
else if cd.recv = BUILD then
    if phantom flag() then
        BuildAll(sender)
    else
        return to parent()
        if cd.phantom count = 0 then
            return to sender(sender)
            cd = NULL
else if cd.recv = INFECTED then
    if cd.state ≠ INFECTED then
        if phantom flag() then
            InfectAll()
    if ready() and strong count = 0 and weak count = 0 and cd.phantom count = 0 then
        cd.state = DEAD
```

```
Procedure Phantom Flag():
if cd = NULL then
    return FALSE
else
    return cd.state = PHANTOM or RECOVER or INFECTED
```

[5] Rodrigo Bruno and Paulo Ferreira. A study on garbage collection algorithms for big data environments. *ACM Comput. Surv.*, 51(1):20:1–20:35, January 2018.

[6] Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. *SIGPLAN Not.*, 48(10):553–570, October 2013.

[7] Sukumar Ghosh. *Distributed systems: an algorithmic approach.* CRC press, 2014.

[8] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[9] Paul Hudak and Robert M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 168–178, New York, NY, USA, 1982. ACM.

[10] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management.* Chapman and Hall/CRC, 2016.

[11] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

[12] Laxmikant V Kale and Sanjeev Krishnan. Charm++: Parallel programming with message-driven objects. *Parallel programming using C+*, pages 175–213, 1996.

[13] R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 708–715, Jun 1992.

[14] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 152–161, New York, NY, USA, 1998. ACM.

[15] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[16] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. *Distributed Computing*, 10(2):79–86, 1997.

[17] Umesh Maheshwari and Barbara Liskov. Collecting distributed garbage cycles by back tracing. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 239–248, New York, NY, USA, 1997. ACM.

[18] John McCorquodale, JD de St Germain, S Parker, and CR Johnson. The uintah parallelism infrastructure: A performance evaluation on the sgi origin 2000. *High Performance Computing 2001*, 2001.

[19] David Peleg. *Distributed Computing: A Locality-sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[20] E.J.H. Pepels, M.J. Plasmeijer, C.J.D. van Eekelen, and M.C.J.D. Eekelen. *A Cyclic Reference Counting Algorithm and Its Proof.* Internal report 88-10. Department of Informatics, Faculty of Science, University of Nijmegen, 1988.

[21] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 211–249, London, UK, UK, 1995. Springer-Verlag.

[22] N. Richer and M. Shapiro. The memory behavior of www, or the www considered as a persistent store. In *POS*, pages 161–176, 2000.

[23] J.D. Salkild. Implementation and analysis of two reference counting algorithms. Master thesis, University College, London, 1987.

[24] Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Trans. Program. Lang. Syst.*, 15(1):1–35, January 1993.

[25] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24a–24a, April 2005.