ways for collecting garbage cycles. One approach which may prove feasible is
to snapshot all the reachability information into one of the nodes, and then to
detect the connected components which do not include pointers from the root
set. We will elaborate on this direction in the full version.

The algorithm that is presented in this work was designed as part of a collabor-
ative effort for distributing Java on top of the MILLIPEDE system developed at
the Technion - Israel Institute of Technology [2]. MILLIPEDE is a strong virtual
parallel machine for distributed computing environments. It implements its own
page-based DSM, which supports several memory consistency protocols. The cur-
rent version of MILLIPEDE is multithreaded, is implemented in user-space on top
of the Windows-NT operating system, and supports dynamic thread migration
and a variety of optimization methods [8]. We refer the reader to MILLIPEDE web
site to find more about this system: www.cs.technion.ac.il/Labs/Millipede.

References

2. A. Schuster A. Itzkovitz and L. Woflovich. Supporting multiple programming
   paradigm on top of a single virtual parallel machine. In Proc. of the 2nd In-
   ternational Workshop on High-Level Parallel Programming Models and Supportive
   Architectures and Languages Europe, volume 258 of Lecture Notes in Computer
4. P. Ferreira and M. Shapiro. Asynchronous distributed garbage collection in a
   cached store. INRIA Rocquencourt, France, May 1996.
5. Richard E. Jones and Rafael D. Lins. Cyclic weighted reference counting without
   delay. In Parallel Architectures and Languages Europe, volume 694 of Lecture Notes
6. D. Lester. Distributed garbage collection of cyclic structures. 4th International
   Workshop on the Parallel Implementation of Functional Languages, September
7. P. Dickman M. Shapiro and D. Plainfossé. Robust, distributed references and
   acyclic garbage collection. In ACM Symp. On Principles of Distributed Computing,
   of locality in distributed shared memory systems. Technical Report CS LPCR
   #9701, Technion, January 1997.
9. R.W. Topor. Termination detection for distributed computations. Information
    architecture. In Parallel Architectures and Languages Europe, volume 258 of
11. W. Yu and A. Cox. Conservative garbage collection on distributed shared memory

This article was processed using the \LaTeX} macro package with LLNCS style
Proof: From Claim 1 we can conclude that if \( R_1 \cup R_2 \cup R_3 = \text{FALSE} \), then \( R_0 = \text{TRUE} \) is also maintained. \( R_3 = \text{FALSE} \) from Claim 7.

Claim 9 If node 0 hasn’t received a black token, \( S = \{0\}, \) node 0 is white and \( \sum_{\text{remote}(X) \text{resv}} = 0 \) then page \( X \) can be safely reclaimed.

Proof: We know that the page can be safely reclaimed if there are no pointers to page \( X \) in the local memory of every node and there are no messages with pointers to page \( X \) that have not been delivered yet. Claim 8 gives us that \( R_0 = \text{TRUE} \) and \( R_3 = \text{FALSE} \). From \( [R_0 = \text{TRUE} \cap S = 0 \cap \text{node 0 is white}] = \text{TRUE} \) we conclude that there are no pointers to page \( X \) in any of the local memories of the nodes. And from \( R_3 = \text{FALSE} \) we conclude that there are no messages with pointers to page \( X \) that have not been delivered yet. So page \( X \) can be safely reclaimed.

8 Conclusion

In this paper we have proposed an efficient approach to collecting garbage from the Distributed Shared Memory.

Most of the work in the algorithm is localized in the nodes. Each node of the network has two counters for indicating the number of local pointers to pages in DSM and for counting references traveling across node boundaries.

We assume that there is a static spanning tree consisting of the network workstations. The algorithm uses this spanning tree to send signals to the leaves of the tree through its nodes, and to send tokens in the direction of the root. Concerning a specific page, every node, after receiving a signal, waits for the tokens from all of its children in the tree, and sends its token only if it sees no local pointers to this page in its memory. Tokens and signals can be piggybacked on messages required for other computations. So no additional messages are sent and the GC tokens are added to messages only when it is very probable that a node no longer references the page.

The part of the algorithm that includes the exchange of tokens between nodes was actually developed as it is presented in this paper. First, the invariant components were constructed one by one. Then, for every component, a rule was formulated to maintain this component. So by the time the invariant was developed it was easy to see that the algorithm was correct and its correctness was indeed proved formally later.

Duplication of messages has no effect in the described algorithm; every node remembers all the tokens it has received after it gets a signal and before it sends its token. When a node sees that some token was reduplicated, it simply ignores it.

Because our algorithm knows how to manage reduplicated messages, it is safe to use idempotent messages in order to significantly reduce the effect of message lost.

The only disadvantage of the proposed algorithm is its inability to collect (page) cycles of garbage. In our future research we will work to find efficient
Node 0 hasn’t received a black token and it is white itself. So it is obvious from Rule 3 that no node not outside $S$ has turned black in the current wave of tokens. This means that no node $j$ in the interval of time $[t_0, t_j]$, i.e., the interval of time when node $j$ was not outside $S$, has increased its remote or local counter. Also we know that no messages were sent by node $j$ at this time (otherwise it would increase its remote counter) and no messages were received at this time by node $j$ (otherwise it would increase its local counter). And without sending or receiving messages at some node, remote and local counters at this node change by the same values at the same rate by definition, i.e.,

$$\text{remote}_{j(t_0)}(X) - \text{remote}_{j(t_j)}(X) = \text{local}_{j(t_0)}(X) - \text{local}_{j(t_j)}(X).$$ (5)

By Rule 2 node $j$ sends its token only if $\text{local}_j = 0$. So

$$\text{local}_{j(t_j)}(X) = 0.$$ (6)

By definition, the final sum that we receive at node 0 is:

$$\sum \text{remote}(X)_{resv} = \sum_{j=1}^n \text{remote}_{j(t_j)}(X)$$ (7)

From 5 and 6 we get:

$$\text{remote}_{j(t_j)}(X) = \text{remote}_{j(t_0)}(X) - \text{local}_{j(t_0)}(X)$$ (8)

From 7 and 8 we get:

$$\sum \text{remote}(X)_{resv} = \sum \text{remote}_{j(t_j)}(X)$$
$$= \sum \text{remote}_{j(t_0)}(X) - \sum \text{local}_{j(t_0)}(X)$$ (9)

If $R3 = \text{TRUE}$, then at time $t_0$ there was at least one message that was sent in one of the previous waves and hasn’t been received yet in the current wave. So from Claim 3 we can conclude that

$$\sum \text{remote}_{j(t_0)}(X) - \sum \text{local}_{j(t_0)}(X) > 0$$ (10)

From 9 and 10 we get:

$$\sum \text{remote}(X)_{resv} > 0$$ (11)

This contradicts the condition that $\sum \text{remote}(X)_{resv} = 0$. So the assumption that $R3 = \text{TRUE}$ was wrong, and the claim is correct.

**Claim 7** If node 0 hasn’t received a black token, $S = \{0\}$, node 0 is white and $\sum \text{remote}(X)_{resv} = 0$ then $R1 \cup R2 \cup R3 = \text{FALSE}$.

**Proof:** The correctness of Claim 7 follows directly from Claim 5 and Claim 6.

**Claim 8** If node 0 hasn’t received a black token, $S = \{0\}$, node 0 is white and $\sum \text{remote}(X)_{resv} = 0$ then $R0 = \text{TRUE}$ and $R3 = \text{FALSE}$.
The proof is by induction on the number of messages sent and received before there were no messages in the network.

**Base:** In the beginning local and remote counters at all nodes are initialized to 0.

**Assumption:** After \( n - 1 \) messages sent and received \( \sum \text{remote} = \sum \text{local} \).

**Step:** Let’s look at node \( i \) that sends message \( n \) to another node \( j \). Let’s assume that the message contains \( m \) pointers to page \( X \). Then, after the message was sent and before it was received, \( \text{remote}_i(X) \) was increased by \( m \) and so \( \sum \text{remote} \) was increased as well. After message \( n \) arrived at node \( j \) \( \text{local}_j(X) \) was increased and \( \sum \text{local} \) was increased as well by \( m \). Thus, message \( n \) had no effect on the difference between \( \sum \text{local} \) and \( \sum \text{remote} \). So Claim 2 is correct.

**Claim 3** If there are messages in the network that were sent but have not yet arrived, then \( \sum \text{remote} > \sum \text{local} \).

**Proof:** Supposing that the Claim is not correct, let’s check the first time \( T \) its correctness was violated, i.e., until that moment \( \sum \text{remote} > \sum \text{local} \). The relation could change only if a local counter at some node was increased and the remote counter was not. This could happen only in the case of receiving a message. Let’s denote \( T' \) as the time when the last message was sent or received before time \( T \). At \( T' < t < T \) \( \sum \text{remote} - \sum \text{local} \) was equivalent to some positive number \( M \) that is obviously equivalent to the number of pointers to page \( X \) in all the messages that were sent but haven’t yet arrived. Following the arrival at time \( T \) of some messages, \( \sum \text{local} \) couldn’t increase by a number greater than \( M \). So the claim is always correct.

**Claim 4** \( \sum \text{remote} \geq \sum \text{local} \) is always true.

**Proof:** The correctness of Claim 4 follows directly from Claim 2 and Claim 3.

**Claim 5** If node 0 hasn’t received a black token, \( S = \{0\} \) and node 0 is white, then \( R1 \cup R2 = \text{FALSE} \).

**Proof:** \( R1 \) is false, because at the moment the wave reaches node 0 the only node that is not outside \( S \) is node 0 itself, and it is white.

\( R2 \) is false by definition because \( S \) doesn’t contain black tokens.

Let’s denote \( \sum \text{remote} \) that was actually received by node 0, plus \( \text{remote}_{\text{node } 0}(X) \) as \( \sum \text{remote}(X)_{\text{resv}} \). We also denote by \( \text{remote}_{i(t)}(X) \) the value of the remote counter of page \( X \) at node \( i \) at time \( t \), and \( \text{local}_{i(t)}(X) \) the value of the local counter of page \( X \) at node \( i \) at time \( t \).

**Claim 6** If node 0 hasn’t received a black token, \( S = \{0\} \), node 0 is white and \( \sum \text{remote}(X)_{\text{resv}} = 0 \) then \( R3 = \text{FALSE} \).

**Proof:** Let’s assume that the claim becomes incorrect in some wave, which will hence forth be called the current wave.

In the proof we denote by \( t_0 \) the time when the current wave began and by \( t_j \) the time when node \( j \) (except node 0) sent its token.
- \( t_{err} \) - time when \( R \) became false;
- \( t_1 \) - time when node \( i \) sends its token;
- \( t_2 \) - time when node \( i \) first became dirty after \( t_1 \) \( (t_2 > t_1) \);
- \( t_3 \) - time when the wave of tokens first reached node 0 after \( t_1 \) \( (t_3 > t_1) \) (this can be regarded as the end of the wave).

In the previous waves we know that \( R \) was always true by the way that we selected “current wave”. Because \( i \) is the first node outside \( S \) that became dirty in this wave, then until \( t_1 \) all nodes outside \( S \) were clean by definition of \( t_1 \).

Node \( i \) is outside \( S \) at \( t_1 < t < t_3 \).

At \( t_1 < t < t_2 \) node \( i \) is clean by definition of \( t_2 \). So at \( t_1 < t < t_2 \), \( R0 = \text{TRUE} \), because node \( i \) is the first node outside \( S \) that became dirty. \( \Rightarrow \) at \( t_1 < t < t_2 \), \( R = \text{TRUE} \).

Now we know that \( t_2 < t_{err} < t_3 \).

The fact that node \( i \) was clean and became dirty later tells us that it has received a message containing some pointer to page \( X \). Until this message was sent, node \( i \) was clean, so \( R0 = \text{TRUE} \) implies \( R = \text{TRUE} \).

If the message was sent to node \( i \) in one of the previous waves, then \( R3 = \text{TRUE} \), because it’s true without dependency if it arrives at node \( i \) at \( t_2 < t < t_3 \) (node \( i \) was outside \( S \) at this time interval), or if it doesn’t arrive to it until \( t_3 \). If \( R3 \) becomes true in the middle of some wave it stays true until the end of this wave, hence \( R \) is true from the beginning to the end of the current wave.

Therefore the message that made node \( i \) dirty must have been sent in the current wave, either by a node that is outside \( S \), or by a node that is not outside \( S \). But we know the message couldn’t have been sent by a node outside \( S \), because all nodes outside \( S \) were clean until \( t_2 \). In addition, the message could only have been sent by a node with a local counter \( > 0 \). So the only possibility is that the message was sent by a node not outside \( S \). This node increased its remote counter before sending the message, and since it is not outside \( S \), it must become black by Rule 2. So, until the time when this node sends its token (let’s denote it \( t_5 \)) and becomes white, \( R1 = \text{TRUE} \). At \( t_5 < t < t_3 \), \( R2 = \text{TRUE} \), because after \( t_5 \) \( S \) contains a black token. Thus, we found no place for \( t_{err} \) in the interval \( (t_1, t_3) \). Thus \( R \) is always true, and is an invariant of the algorithm.

In the proof, \( \sum_{\text{local}} \) and \( \sum_{\text{remote}} \) are denoted as global values that we would receive if we summed up the respective counters at the nodes at a certain instant of time.

**Claim 2** If there are no messages that are in the network then \( \sum_{\text{remote}} = \sum_{\text{local}} \).

**Proof:** By definition local and remote counters are decreased in the same cases. The only difference between them is that local counters are increased when the message arrives by the amount of pointers to page \( X \) contained in that message, whereas remote counters are not changed. Another, final difference in counting is that remote counters are increased when sending messages, whereas local ones are not.
When node 0 starts a garbage scanning process for page X, it can assign the signals that it sends to its children the number 1; then, every other node that would receive a signal with number 0 would send to all its siblings signals with the same number. After receiving a signal with number 0, a node would know that the token that it must send (receive) would contain the number of the signal that it has received. At the end of the wave 0 the root will increment this counter if an additional wave is required, and the process will go on in the same manner. As a result, when node i receives a token with a wrong wave number, it will know to neglect this token. Thus we avoid errors that could occur if some duplicated token arrives not in its wave.

There are no ideal solutions for avoiding the effects of message loss and hardware failures, because if they occur, the lost garbage collection data cannot be restored without additional overhead. We propose the minimal overhead of piggybacking control messages from time to time. For example, if node i doesn’t receive a token from one of its siblings (say, node j) for a considerable period of time, then there are chances that either the token was lost or node j failed to send it because of hardware problems. In this case, node i waits for any message that goes to node j and piggybacks a control message on it. If node i doesn’t receive an acknowledgment (token) for its control message, then there is some hardware problem in the network or in the node j itself. If node j receives a control message from node i, then the chances are high that the last token was lost. In that case it sends to node i a token with the appropriate wave number. Thus the tree structure of nodes used in our algorithm allows us to avoid serious problems that can arise as a consequence of using the unreliable connection protocol.

7 Safety Proof

We start the proof by showing that \( R \) is indeed invariant. Then for page X we show that if

\[
S = \{0\}
\]

\( \cap \) node 0 is white and \( local_0(X) = 0 \)

\( \cap \) all tokens at node 0 are white

\( \sum remote(X) = 0 \)

\( \Rightarrow (R1 \cap R2 \cap R3) = \text{FALSE} \Rightarrow (R0 = \text{TRUE}) \) and \( (R3 = \text{FALSE}) \)

\( \Rightarrow \) page X is garbage \( \Rightarrow \) page X can be safely reclaimed.

Claim 1 \( R \) is an invariant.

Proof: Here we are going to prove that \( R = R0 \cup R1 \cup R2 \cup R3 \) is always true.

Let’s assume by negation that it is false, and look at the wave in which \( R \) first became false. Because \( R \) became false, then \( R0 \) is false too, and thus some node outside \( S \) is dirty. Let’s look at node i, which was the first node outside \( S \) to become dirty in the wave when \( R \) first became false. Regarding this wave we denote:
of pages. Any collection which is based on the reference counting is not able to reclaim cycles of garbage. This problem becomes more severe when pages are collected. Let’s suppose that object \( a \) in page \( A \) references object \( b \) in page \( B \). Also, there is object \( c \) in page \( B \) that references object \( d \) in page \( A \). Though there is no garbage cycle observed in the graph of objects references, there is a circle \( A-B-A \) in the graph of references between the pages. Thus, pages \( A \) and \( B \) won’t be reclaimed by our algorithm, even if all the objects in them are unreachable from the outside. On the other hand, page-based collection solves the garbage cycles problem when some cycle is located entirely inside one page. When there are no “external” references to such a page, this cycle will be safely discarded from the memory along with the page that contains it.

6 Fault Tolerance

If the algorithm is implemented using an unreliable connection protocol (like UDP), then there is no way to avoid loss and duplication of signals and tokens. As we mentioned earlier, this sometimes causes a serious problem for a considerable number of distributed GC algorithms. In this section we show that our GC algorithm avoids unsafe reclamation of garbage pages as a result of message duplication. Message loss does not affect the safety of the collection process, but may rather delay an object’s reclamation for a very long time. Our algorithm helps decrease the number of such unreclaimed objects (pages).

Let’s consider collection regarding some page \( X \) and assume that some token \( A \), sent from node \( i \) to node \( j \), was duplicated in some wave. Every node in our algorithm can be in one of two states: either it already sent its token, and is waiting for a signal (State 1), or it already sent its signal, and is waiting for a token (State 2). If node \( j \) is in State 1, when it receives the duplicated token, it can simply ignore it. It is obvious that this decision is correct, because until node \( j \) receives the signal it is waiting for and passes it to node \( i \), there can’t be any new tokens that are sent from node \( i \). Otherwise, node \( j \) can be in State 2 when it receives the duplicated token. In the case that this token was sent in the current wave, node \( j \) can ignore it as well. The reason to this is that node \( j \) hasn’t yet received the tokens from all of its siblings, or it hasn’t yet sent its own token. Thus, if \( j \) receives two or more tokens from \( i \) while its state doesn’t change, then it may conclude that only one of them is valid.

The situation is more difficult when the duplicated token was sent in one of the previous waves and node \( j \) is in State 2 in the current wave. Then this message can bypass the real token from node \( i \) and be accepted by node \( j \) erroneously. To avoid this, the algorithm should be slightly altered, adding one more rule that ensures it will work correctly in the case of message duplication.

**Rule 9** Each wave in the algorithm is given its ordinal number starting from 1. Every signal and every token in the algorithm is given an integer value, containing the corresponding wave number. A token or a signal, containing a wave number that differs from the current one, is ignored by the node that receives it.
The extension in the size of the required data structure in every node increases not only the memory overhead, but also makes the operations on these structures more time consuming. In particular, it increases the complexity of the search operation required for changing the value of some counter. That is why the local work of the page-based solution will be much faster. In addition, utilization of a page table for keeping garbage collector information for every page avoids spending valuable time on creation and removal of GC data structure elements (counters, state), whose place can be reserved at system start-up.

As mentioned in Section 2, there is a system function for converting a virtual DSM address to the corresponding entry in a page table that contains a record on the corresponding page. This function must be used in order to access a certain local or remote counter that is stored in every page table entry. In other words, every change to the garbage collection information for some page must be preceded by the call to this converting function. One might think that an algorithm which collects objects instead of pages makes any conversion unnecessary, thus gaining a considerable amount of execution time. However, consider the case of some memory cell that doesn’t reference the beginning of some object, but rather one of its fields in the middle. In that case, a function is required for converting a reference to the beginning of the corresponding object, so that the right counters in GC local data structure can be found. This function requires information concerning all the objects referenced locally. For example, a field \( f \) belonging to some object \( c \) that is created at node \( A \), is referenced in node \( B \). To increase the local counter of \( c \), one must find it in the objects counters table of node \( B \), according to its initial address. If node \( B \) contains no data about \( c \), \( A \) must send this information (initial address, object’s size). Thus, in addition to the subsequent increase in the memory overhead, one can observe an increase in the number of messages sent between the nodes. No such overhead is required in the case of the page-based GC, because the information for converting a virtual address to a page number must exist for any page-based DSM regardless of which kind of GC is used. The only case in which page-based GC will need additional data is when the size of some object is greater than the DSM page size. Then the algorithm would have to know which pages this object occupies, so that the counters corresponding to these pages could be changed simultaneously.

The memory reorganization process as an important part of every garbage collection is not considered in this paper. But it is very important to note that swapping the free memory space after page reclamation is cheaper than the reorganization of small empty “cells” left after discarding separate objects.

At the same time, page-based collection has its own drawback. Even if some object \( x \) in page \( X \) becomes unreachable, the collector will not reclaim it until all other objects in the page become garbage. But in practice every memory usage agrees with the memory locality principle, which says that memory access is localized in time and space. In other words, if some object \( x \) in page \( X \) becomes unreachable, it is very probable that all other objects in page \( X \) will soon become garbage too, and the page will finally be reclaimed.

There is another problem that is caused by collecting garbage in granularity
high that it won't become dirty with respect to that page again. This is the main idea by which the algorithm saves the additional network traffic the collecting process usually requires. When collecting some page X, it is most probable that there will be only one wave of tokens. Even in this case, all the repeat signals and tokens can be piggybacked, thus avoiding any additional messaging at all. Although cases exist in which a node that sent its token and was considered clean may become dirty again, the root never reclaims it. This is proven in Section 7.

Sometimes the best computation time is desired, which can be achieved by turning the garbage collection off. Most algorithms in the distributed literature do not allow to suspend the GC process, and in particular they do not allow suspension once the collection process has begun. In our algorithm, if node 0 doesn’t initiate “garbage scanning”, no messages are sent at all. All the work is localized in the reference counting process, which has little influence on computation time as compared to messaging process. Transmission of GC messages (tokens and signals), can be safely delayed, as we prove the safety of the algorithm in Section 7 without implying synchronous message passing. We assume that it may take an arbitrarily long time for some message to arrive. Thus, suspending the “garbage scanning” process at some point in time T is not different than delaying the arrival of GC messages for that time T.

5.2 Collecting Pages vs. Objects

In order to further reduce the messaging overhead, it was decided that the algorithm would collect pages instead of objects. Collecting objects in every GC algorithm involves local and remote operations for every object. Objects allocated by a user’s program can be extremely small. Thus, collecting each one of them separately causes a significant increase in the number of messages that are sent for gathering the distributed reachability information. For example, if the algorithm that was presented in Section 4 worked with objects instead of pages, the number of signal-token waves would increase according to the average number of objects in the page.

Moreover, in the object-based GC algorithm, counters would be required for every object. In the worst case, when every allocated object has the size of only one byte, the memory overhead required for counters would be at least 50% (if each counter requires one byte as well). In practice this overhead is even greater. Let’s take for example the algorithm from Section 4. For each page in every node, local and remote counters are necessary. Let’s assume every one of them takes two bytes. One more byte is needed for each page to hold the information regarding node color (black or white) and from which other nodes the tokens have already been obtained. So the algorithm requests 5 bytes for each page in every node. For our page-based algorithm, if the page size is 4K, then the required memory overhead is about 0.12%. But if the algorithm is changed to work with objects instead of pages, the necessity for holding these extra 5 bytes for each object in every node that references it will not be eliminated. In this case if the average number of objects per page is M, then the memory overhead is increased exactly M times.
Now the real reason for using $\text{remote}(X)$ will become clear. First we formulate Rule 5, which maintains $R3$, and then we shall explain its meaning.

**Rule 5** Each node sums up $\text{remote}$ counters that arrived to it via tokens from its siblings, adds to the sum its own $\text{remote}(X)$ counter and sends the result together with the token to the parent, according to Rule 1.

In the last example, when the wave of tokens reaches node 0 it doesn’t receive any black token, because all nodes were white during the wave. If we would sum $\text{local}(X)$ counters using the same method as in Rule 5, but without using $\text{remote}(X)$, then node 0 would receive the sum 0, informing it that none of the nodes points to page $X$. But actually, as we see, there is a pointer to that page that was sent in the “delayed” message, a fact which node 0 should notice. That is where $\text{remote}(X)$ comes into use. Let’s assume that the “problematic” message was sent from node $A$ to node $B$ (that is outside $S$). Clearly, $A$ increased $\text{remote}_A(X)$ before sending this message. If $B$ doesn’t decrease its $\text{remote}_B(X)$, then at node 0 we will see that $\sum \text{remote}(X) > 0$ (this fact will be proved in Section 7). Thus we derive the following rule for node 0 to determine if page $X$ is garbage:

$$R \cap S = \{0\}$$

- node 0 is white and $\text{local}_0(X) = 0$
- all tokens at node 0 are white
- $\sum \text{remote}(X) = 0 \implies$ page $X$ is garbage.

Otherwise, node 0 begins the repeat wave of the algorithm and directs “repeat” signals outwards. The algorithm for the outward signals complies with the following rules:

**Rule 6** When node 0 receives a token from each of its siblings, it waits until its $\text{local}(X)$ counter becomes 0, checks if it is black or received a black token or $\sum \text{remote}(X) > 0$, and then sends a repeat signal to all its children. Otherwise it discards all the replicas of page $X$ from the memory.

**Rule 7** An internal node that receives a repeat signal sends the signal to all its children.

**Rule 8** A leaf receiving a repeat signal gets a white token.

## 5 Effectiveness

### 5.1 Saving the Network Traffic

The rules derived in Section 4 imply that by the time a wave of tokens reaches the root of the spanning tree, every node in this tree has become clean at least once since the wave began. Usually, page usage by an individual node is localized in time. Thus, if some node is clean with respect to some page, the chances are
Rule 2 A node that increases local(X) or remote(X) becomes black.

Now, assume for the last example that local(X) in node j decreases to 0, and it was the only black node. Then j finally will be in S, and because localj(X) = 0, it will transmit the token to its parent. After that it will remain outside S until the new wave. Then R0 may be false because i may be dirty and R1 is false because there are no black nodes not outside S. Now we add an additional possibility: each token can be painted in black or white color. Let's consider all the tokens at leaves to be white. Then a token that is transmitted from a black node is black too. So we change R to R0 U R1 U R2, where

\[ R2 \equiv \text{some node in S has a black token.} \]  

(4)

R2 is maintained by the following rule:

Rule 3 A node that is black or has a black token transmits a black token, otherwise it transmits a white token.

To prevent a node that turned black from transmitting repeatedly black tokens at successive waves, let's "paint" a black node that sends a token white. This doesn't violate R because R doesn't depend on the color of the nodes outside S. So

Rule 4 A node that transmits a token becomes white.

Still we cannot be sure that R holds, because in some wave of tokens the following situation could occur:

In some wave i, node A not outside S sent message Z, containing a pointer to page X, to node B, outside S. Let's assume this was the only message sent in waves i and i+1. Then node A turned black, received the tokens from all of its siblings, and sent a black token to its parent when A's local(X) counter became 0. After that moment node A is white for good. Consider that message Z arrives at node B only in wave i+1, when B is outside S, and all the nodes of the tree are white because none of them sent messages in waves i and i+1. Then with regard to wave i+1

- R0 doesn't hold, because there is a page with a pointer to page X encapsulated in message Z, sent by node A in wave i, received by node B outside S in wave i+1, which then increases its local(X) counter and becomes dirty;
- R1 doesn't hold, because all the nodes are white;
- R2 doesn't hold, because in wave i+1 all the nodes were initially white and none of them turned black. Thus there aren't any black tokens in the tree.

So we change R to be R0 U R1 U R2 U R3

R3 \equiv \text{There is a message that was sent in one of the previous waves of tokens destined to some node outside S, containing a page with one of its objects pointing to page X and this message has arrived to its destination in the current wave or still hasn't arrived yet.}
At the start of "garbage scanning", node 0 sends signals to all the leaves of the tree to start sending tokens in the direction of the root through their ancestors. We call this moment the beginning of the wave. When a leaf receives a signal it sends a token that moves through the nodes of the tree to the root. When the root receives tokens from all of its siblings, it detects if page X is garbage. If yes, then it reclaims all the replicas of this page and the scanning ends. Otherwise, next wave begins. It is important to note that signals and tokens regarding some page are sent only when the chance that this page is garbage is very high. This will be shown in Section 5.1.

The set of nodes holding one or more tokens in the wave at a certain moment and the set of leaves that still haven’t received a signal from the beginning of the wave is called $S$.

**Definition 1.** A node $j$ is outside of $S$ if $j$ is not in $S$ and the path from $j$ to node 0 (the root) includes nodes from $S$.

Let the invariant $R$ be set initially to $R_0$.

$$R_0 \equiv \text{all the nodes outside } S \text{ are clean.} \quad (2)$$

Initially $R_0$ is true when node 0 sends signals to the leaves and there is no leaf which has already received one of these signals, so the set of nodes outside $S$ is empty in the start of the wave.

Tokens then move inward (towards the root) according the following rule:

**Rule 1** A leaf that has a local counter 0 transmits a token to its parent; an internal node with $\text{local}(X) = 0$ that has received a token from each of its children transmits a token to its parent; a dirty node ($\text{local}(X) > 0$) doesn’t transmit a token.

When a node transmits a token, it is left without any tokens.

A node that transmits a token moves “outside” of $S$ and its parent is added to $S$ if it wasn’t in $S$ previously.

Rule 1 works and is sufficient for determining if page X is garbage only on the condition that no messages were sent since the scan began.

In the presence of messages, the following situation could occur. Assume some node $j$ that is not outside $S$ has in its local memory a page $Z$, where one of its objects $z$ points to the object $x$ in a page $X$. At another node $i$ that is outside $S$ (and its local counter is 0) the following operation is executed: $y = z$. Node $j$ sends page $Z$ to node $i$, so that in the end $y$ points to $X$ and the local counter of page $X$ at node $i$ becomes 1 (the node becomes dirty). Thus $R_0$ becomes false.

As we have mentioned, each node has an additional, remote counter. In the last example we saw that when node $j$ sent its page $Z$ to node $i$ it incremented its remote counter. Now we add the possibility to paint each node in one of two colors: black or white. Let’s assume all nodes to be initially white. A node turns black immediately after it increments its remote or local counter. So we change $R$ to $R_0 \cup R_1$, where

$$R_1 \equiv \text{at least one node not outside } S \text{ is black.} \quad (3)$$

Accordingly, the rule is as follows:
local \(i\) is increased:

- when some reachable object at node \(i\), that is not in page \(X\) itself, is assigned a pointer (handle) to some object in page \(X\) (local \(i\)(X) is increased by 1);
- when receiving a message containing pages with pointers to page \(X\), local \(i\)(X) is increased by the total number of the pointers to page \(X\) in this message.

local \(i\) is decreased:

- when reassigning a pointer at node \(i\), that was previously pointing to page \(X\), to another value (local \(i\)(X) is decreased by 1);
- when deleting the object in page \(X\), which was pointed to from node \(i\) by another object, which is not in page \(X\) itself (local \(i\)(X) is decreased by 1);
- when some page \(Y \neq X\), which contains pointers to page \(X\), is discarded from the local memory of node \(i\) (local \(i\)(X) is increased by the number of pointers to page \(X\) in page \(Y\)).

A node with a local \((X)\) counter 0 is called clean regarding page \(X\). If a local \((X)\) counter at the node is greater then 0 then the node is denoted as dirty with respect to page \(X\). It is obvious that local \(i\)(X) \(\geq 0\) is always true.

### 3.2 Remote Counters

The remote counters are almost exactly the same as the local counters except for one thing. Remote counters are not increased when a message containing pages with pointers to page \(X\) is received; rather, they increase when such a message is sent. In other words, when some message containing pointers to page \(X\) is sent from node \(i\), remote \(i\)(X) is increased by the total number of these pointers in the message. Initialization of remote counters and other cases of their modification are the same as for local counters. It is thus easy to see that when all the messages containing pointers to page \(X\) which were sent have been received then \(\sum remote_i(X) = \sum local_i(X)\). This will be proven in Section 7.

### 4 Determination of a Page as Garbage

We develop an algorithm for determining if a certain page \(X\) is garbage using a sequence of approximations to help us find invariant \(R\) and condition \(C\), such that

\[ R \cap C \implies \text{page } X \text{ is garbage.} \quad (1) \]

Note that all actions here with regard to the collection algorithm concern the same page \(X\). In other words, page \(X\) is the default, when no other page is indicated.

Let some node be the root of a fixed spanning tree of the given graph of all the nodes participating in a process of calculation. Let’s refer to the root as node 0.
can have several processes running on it and each process can allocate memory from the DSM. We denote the unit of allocation as an object. The DSM consists of pages, containing one or more objects. Every page can be replicated at several nodes.

We say that there is a pointer at node \(i\) to some page \(X\) if some object in the local memory of node \(i\), that is not in page \(X\) itself, is pointing to some object in one of the replicas of page \(X\). We assume that, given a pointer or a handle to some object, it is possible to find which page contains this object.

As a root set we denote the data that is immediately available to a program, without following any pointers. Typically this would include local variables from the activation stack, values in machine registers, and global, static, or module variables.

An object becomes reachable after allocating memory for that object, receiving a pointer (handle) to that memory and assigning it to one of the previously reachable objects.

An object becomes garbage when all reachable objects that were pointing to it are pointing to another object or NULL.

A page becomes garbage when all objects contained in all of the replicas of this page are garbage themselves.

A node \(i\) is clean for page \(X\) when there are no pointers to page \(X\) in the local memory of that node. Otherwise, node \(i\) is dirty for page \(X\).

There are messages traveling between the nodes that can contain reachable objects pointing to pages of DSM. These are the only messages that interest us: when we mention the word message, we refer to a message containing pointers to the page that is being checked for garbage.

Each node has in its local memory page tables containing an entry for every page in the DSM.

### 3 Local Counting

Before introducing the Garbage Collection algorithm, we want to describe some local information that is gathered for each page even when the GC algorithm is not actively reclaiming garbage pages. This information consists of two counters attached to each entry of the page table in the local memory - local and remote counters.

#### 3.1 Local Counters

For every node \(i\) we want to count the number of pointers located at node \(i\) to each page allocated in the DSM. When page \(X\) is created, local counters for this page at every node are 0. At node \(i\) we denote such a counter as \(\text{local}_i(X)\).
but it scales well enough for collecting garbage information as well. The idea is as follows: one or more waves of tokens go to and from the root node of some spanning tree built for the purpose of garbage collection. These tokens contain the local information created by reference counting. This data is then analyzed at the root, which in turn decides whether or not to collect the page.

It is known that Topor’s algorithm doesn’t work in the case of asynchronous message passing. We overcome this problem by analyzing the messages being sent and updating the counters according to the number of references the messages contain. So even if some message containing a reference takes a long time to arrive at its destination, the node that sent the message already knows which references were sent. It passes this information via the token to the root. So the root knows that the message, which has not yet arrived, contains a reference, and doesn’t allow the corresponding page to be reclaimed.

The algorithm is resilient to hardware and message failures. Duplication of messages is ignored, because the data is sent in the spanning tree and every node is aware of which nodes have already sent their tokens.

But perhaps the most important advantage of our algorithm is that it sends messages that contain auxiliary data for garbage collection only when the chance that some page can be reclaimed is very high. There are no additional messages when there is no attempt to collect pages. The only overhead in the absence of collection is the supplementary expense for maintaining local data structures per page.

Our solution is also tolerant of the consistency protocol and doesn’t depend on some global order of operations in the system.

The algorithm proposed in this paper still doesn’t collect cycles of garbage. Additional research is required to check the efficiency of using global garbage collection as a last resort, as it was suggested by Yu-Cox [11].

The rest of this paper is as follows. In Section 2, we describe the system model and the basic terminology used in this paper. In Section 3, we show how local counting is carried out. We present two types of counters local and remote. In Section 4, we develop, step by step, an invariant that helps us to describe the global part of the algorithm and to prove its correctness. In Section 5, advantages and drawbacks of the algorithm are shown. In Section 6, we trace how the algorithm can be implemented to avoid the negative effect of message duplication, loss and hardware failure. In Section 7, we use the invariant developed in the Section 4 to prove the safety of the algorithm.

2 System Model

The algorithm is aimed to work on a Distributed Shared Memory, which presents an abstraction of a shared memory for some Parallel Virtual Machine. The real network structure and location of local memory spaces, distributed between the workstations, is totally hidden from the programmer.

We assume here that there are several separate processors using Distributed Shared Memory (DSM). We reference each one of them as a node. Each node
because all the nodes are in a consistent state, but it is a very ineffective solution with respect to performance considerations, and is thus used as a last resort. Yu - Cox and other algorithms based on weighted reference count ([5],[6],[11]) also do not solve the problem of messages duplication, as a result of which some global objects may be unsafely reclaimed. This happens when some node observes that the reference to some global object is unreachable and sends it back to the owner along with its weight. If this message is accidentally duplicated, then the owner erroneously counts this returned weight twice. In this case the owner may mark this global object unused before all the references to it become unreachable, which is unacceptable in a loosely coupled distributed system.

Other techniques that try to avoid race conditions on remote counters and are resilient to hardware and message failures use Reference Listing (SSP Chains [7], Garbage Collector for Network Objects [1]). Their most significant shortcoming is the increase in the amount of auxiliary messages sent between nodes.

It should be noted that Reference Counting and Reference Listing techniques have one common drawback. They do not allow collection of cycles of garbage. There are various solutions for collecting these cycles, as well as other unreachable data (Hybrid Garbage Collection [4]), but they all add a considerable amount of messaging to the implementation of their algorithms.

In Distributed Systems, communication causes bottlenecks in computation processes. Thus it is not permissible for a GC algorithm to exchange messages only for the purpose of decreasing the time required for reclaiming an object that has become unreachable. That is why, when we encountered various tradeoffs, we chose to reduce network traffic at the cost of a possible increase in required memory space and local work.

Our solution is based on a variation of Reference Counting as a local part of the algorithm that is executed by every node without sending messages. The counters are allocated for every page in the Distributed Shared Memory (DSM), rather than for every object, as in earlier solutions. This solution is bound up with the widely applicable page-based implementation of DSM. An address or a handle of some object in the page-based DSM can be easily converted to the page table entry number corresponding to the page which contains this object. When the algorithm observes a pointer to some object, it associates this pointer with the proper page. The memory overhead of counters is thus reduced proportionally to the average number of objects in a page. Although a page may contain some unreachable objects, there is no guaranteed time period during which they can be reclaimed, because other objects in this page may still be reachable for an unlimited time, and the unit of reclamation is a whole page. But it is preferable to wait for all objects in the page to become unreachable than to exchange messages in order to receive the collection decision regarding each separate object. Thus, page-based counting considerably reduces the overhead communication for GC as well.

To collect the reachability information about some page, which is distributed between local counters on the nodes of the network, we use the technique first proposed by R.W. Topor [9]. He used this method for Termination Detection,
Most distributed systems are asynchronous. A message only tells about the past of the sender. There is usually no single global order of events.

Failures do occur. Machines crash; processes get into infinite loops. Messages get lost and delivered out-of-order (even with TCP/IP, in the presence of crashes).

As a consequence of the above, data tends to be inconsistent. If the data being copied is the Garbage Collection (GC) status, then the collection algorithm may decide to reclaim an object based on incorrect information.

As a result of these problems, many of the simple approaches for implementing garbage collection in distributed systems do not work at all or make costly demands on the system, thus reducing efficiency and slowing down the work of other parallel processes.

The easiest GC algorithm to distribute is reference counting. One simple approach is to attach a counter to each object. Duplicating or killing a reference to a global object sends an increment or decrement message to the target object. This is very expensive (one message per pointer operation); more importantly, it does not work. Since there is no guaranteed delivery order for messages, an increment might be overrun by a decrement, and the target object unsafely reclaimed.

A general fix is to provide “casual delivery” of messages, but that’s too costly. A simpler fix is Weighted Reference Count (or some variation of it) [3]. When a reference to the global object \( v \) is first created, it is given some total weight. Each local and remote reference to the object \( v \) has a field containing the partial weight. When a reference is duplicated, its partial weight is divided in two and the partial weight of the duplicated reference is given the new value, which equals half of the previous partial weight. Another half of this partial weight travels to the new reference and the partial weight of this reference is given that value. When deleting a reference, its partial weight is sent to the owner of the object \( v \) and is subtracted from the total value. In other words, all the Weighted Reference Counting techniques maintain the following invariant for every object \( v \):

\[
\text{total weight}_v = \Sigma \text{partial weight}_v
\]

When splitting the partial weight of some reference, one may come to partial weight value 1, which cannot be further divided. This problem is known in the literature as weight underflow and is referred to as the most serious problem of the Weighted Reference Counting techniques. There are solutions to this problem [10], but they significantly increase the number of messages sent to update the counters. Weighted Reference Counting is neither resilient to the loss or duplication of messages, nor to hardware failures. Weight underflow, hardware failures and the loss of messages leave a certain number of unreachable objects in the memory, which the collection algorithm is not able to reclaim. The Yu - Cox algorithm [11], for example, avoids costly run-time solutions to these problems in the following way: when a node can not collect the necessary memory space from the garbage, it sends a signal to all the workstations to suspend their computations and runs global collection. Global collection can achieve ideal results
Collecting Garbage Pages in a Distributed Shared Memory with Reduced Memory and Communication Overhead*

Dmitry Kogan and Assaf Schuster

Department of Computer Science, Technion
kogan,assaf@cs.technion.ac.il

Abstract. We present a novel algorithm for Garbage Collection (GC) in Distributed Shared Memory systems (DSM). Our algorithm reduces the network traffic overhead (and the memory and computation overheads), essentially eliminating all communication when there is no active collection, and minimizing it when the collection process is turned on. Our algorithm works correctly for asynchronous environments where messages may experience arbitrary delays on the way to their destinations. It also tolerates arbitrary duplication of messages and is thus a suitable “add-on” for fault-tolerant communication protocols. It does not suffer from problems such as weight underflow (which arise in reference counting techniques). In addition, when applied in granularity of pages (which is the most relevant in page-based DSM systems), then the memory overhead is not inflated when the average allocation size is small, and the memory reorganization required due to the GC operations is simplified.

1 Introduction

Manual memory management is programmer-time consuming and error prone. Most programs still contain leaks of memory even after considerable effort by the programmer. Modern API gives an extended set of functions for memory allocation and deallocation. This complicates the possibility of controlling the freeing of the dynamic objects when they become unreachable (i.e., no other object points to them). The problem is even harder in the distributed environment, because a part of code in the thread, which is in charge of deallocating the storage, must verify that no other object still needs that storage. Thus, many modules must cooperate closely. This leads to a tight binding between supposedly independent modules.

Despite the well-developed techniques of uniprocessor garbage collection, distributed techniques still present some difficult problems for researchers. These problems are engendered by several distinctive features of distributed systems:

- To know the status of a remote workstation one must send and receive messages; messages are costly.