Distributed Garbage Collection Using Group Reference Counting

Yuuji Ichisugi
ichisugi@is.s.u-tokyo.ac.jp
Akinori Yonezawa
yonezawa@is.s.u-tokyo.ac.jp
Department of Information Science, University of Tokyo

Abstract

Reference counting schemes are suitable for garbage collection on distributed systems because they do not require global information to find garbage. However, reference counting schemes have a serious drawback: they cannot reclaim garbage which form cyclic structures. In this paper, we propose an extension of the traditional reference counting scheme, *group reference counting*, which alleviates the drawback. This scheme is able to reclaim circularly referenced objects which are contained in object groups. We describe how this scheme can be implemented on distributed object-oriented concurrent systems. The overhead of this scheme does not depend on the number of nodes. Thus this scheme works efficiently on distributed systems containing large number of nodes.

1 Introduction

1.1 Distributed Garbage Collection

Object-oriented concurrent languages are powerful in describing distributed algorithms. On such languages, automatic storage management mechanisms, like *garbage collection(GC)*, are indispensable. But it is often difficult to perform efficient GC on distributed environments. The following requirements are imposed on distributed GC.

1. The algorithm should be able to executed in parallel in many nodes. Otherwise GC will obstruct improvement of performance by parallel execution.

2. For many distributed systems, it is often the case that some nodes become unreachable from other nodes due to system or network troubles. Even in such cases, the whole system should not stop computation. If the GC algorithm requires synchronization of all the nodes in the system, the system will stop when some nodes become unreachable.

The GC schemes proposed for Distributed Smalltalk[1, 2] are divided into two parts, the local GC and the global GC. The local GC uses only local information within each node, and no internode communications are necessary. In order to collect garbage spanning on several or more nodes, the user must start the global GC which is based on the mark & sweep scheme.

Emerald [3] system also uses the local GC and the global GC. The global GC is based on mark & sweep scheme but improved so that each node do not have to stop its computation for a long period of time during GC.

In these systems, the cost of synchronization becomes very expensive if the number of nodes in the system is significant. In contrast, the overhead of our proposed GC scheme does not depend on the number of nodes.

1.2 Mark & Sweep, Copy Scheme

Usually, mark & sweep, copy or their variations are used for GC on non-distributed systems. Such schemes trace all living objects to find garbage. But these schemes are not suitable for distributed systems because they have the following characteristics.

- All nodes should synchronize at the beginning and the end of GC. If some nodes becomes unreachable for a while, garbage cannot be collected during the period. Furthermore, the cost of synchronization becomes very expensive if the number of nodes in the system is significant.

- Sometimes objects cannot be traced in parallel. In Fig. 1, each subtree in the three nodes can be traced in parallel, but in Fig. 2, only one node executes GC process at a time. The other two nodes must stop computation, or continue computation consuming extra memory space.

1.3 Reference Counting Scheme

There is another traditional GC scheme, reference counting. It is suitable for distributed systems because they do not require global information to find garbage.

Furthermore, reference counting schemes have the following advantages.

- Computation is not interrupted for a significant period of time.

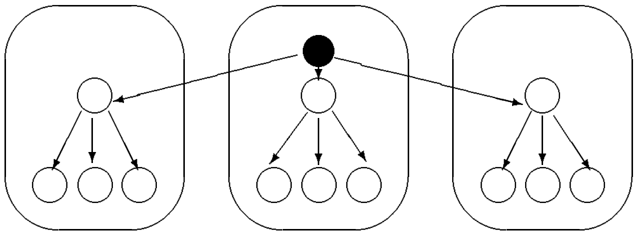

Figure 1: Objects which can be traced in parallel

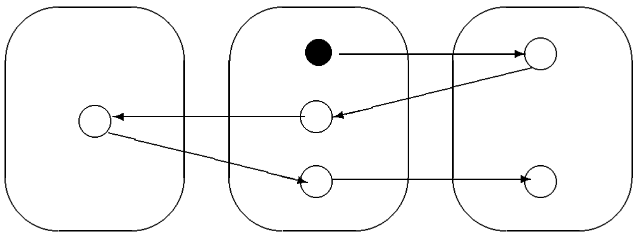

Figure 2: Objects which cannot be traced in parallel

- Overhead depends on the amount of garbage instead of the amount of data used.

However, reference counting schemes have a serious drawback: they cannot reclaim garbage which form cyclic structures. Some variations of reference counting schemes have been proposed which can reclaim cyclic structures in some special cases [5, 6, 7, 8]. In this paper, we propose a more general extension of reference counting schemes than described in [7].

[9, 10] suggest that grouping of objects is very powerful in development and execution environments of object-oriented concurrent languages. The garbage collection scheme described in this paper is effective in object-oriented concurrent systems which have mechanisms to group objects.

2 Distributed Reference Counting

Before describing our group reference counting scheme, in §3, we explain distributed versions of the traditional reference counting schemes. [12, 13] propose distributed reference counting schemes which requires only one message transmission for each internode reference. In this section, first we briefly explain the *weighted reference counting* [12] scheme and propose our new scheme, the *local reference counting*. Then the two schemes are compared.

2.1 Weighted Reference Counting

Each reference to an object have a *weight* as well as the address of the object. The reference counter of an object o is the sum of weights of references to o instead of the number of references to o. When a reference is copied, the weight of the reference is divided into two values. When a reference to an object o is discarded, *delete message* is sent to o with the weight of the reference. When o receives delete message, the reference count of o is decremented by the weight of the discarded reference. When the reference count of o becomes zero, o is garbage.

A reference count of a living object will never became zero even if messages are delayed in the network because there is only one message types, which decrements the reference counter. If a delete message is delayed, the value of the reference count will be larger than the real value, which is greater than zero.

2.2 Local Reference Counting

We propose a simpler scheme which requires the same amount of internode messages.

If an object on a node N refers to an object o on a node N_o, N has a *local reference counter (LRC)* of o. LRC contains the sum of the number of references passed to another node from node N and the number of references copied within the node N. N also remembers an id of a node which is the "first sender" of the reference to o. If N is the creator of the object, the first sender is N itself.

LRC is updated as described below.

- When a reference to o is copied within a node N, LRC of o on N is incremented.
- When a reference to o is discarded by an object on N, LRC of o on N is decremented.
- When a reference to an object o is sent from an object p on a node N_p to an object q on a node N_q, LRC of o on N_p is incremented.
- When a reference to o sent from N_p reaches N_q and there has not been LRC of o on N_q yet, N_q creates LRC of o with the initial value one and remembers N_p as the first sender of the reference to o. If there was LRC of o on N_q already, N_q sends a *discard message* of o to N_p immediately and LRC of o on N_q is incremented.
- When N receives a discard message of o, LRC of o on N is decremented.
- When LRC of o on N becomes zero, N sends a discard message of o to the first sender of the reference to o. If N is the creator of o, o can be regarded as garbage.

(See Fig.3)

A reference count of a living object will never became zero , as the Weighted Reference Counting method, because there is only one message types, which decrements the reference counter.

2.3 Local Reference Counting and Weighted Reference Counting

Both of the Local Reference Counting(LRC) and the Weighted Reference Counting(WRC) require only one message transmission for each internode reference. But the routes of messages are different. In the WRC scheme, a delete message for an object o is sent to the node on which o resides. In the LRC scheme, a discard message is returned to the node which sent the reference. In this case, the load of each node increased by GC is almost in proportion as ordinary computation. So, load balance mechanisms do not have to consider GC.

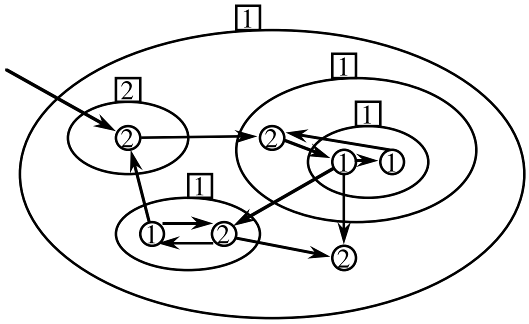

Figure 3: LRC on each node when o is sent from p to q

In the WRC scheme, reference weight underflow must be handled. In the LRC scheme, reference counter overflow can be avoided in many systems. The maximum value of a reference counter is lower than or equal to the sum of the following numbers.

1. the maximum number of references within one node

2. the number of nodes in the entire system

3. the maximum number of references contained in messages flying on the network

4. the maximum number of discard messages flying on the network

If the maximum value of a reference counter is limited, reference counter overflow can be avoided.

In the WRC scheme, the delete message of o is always sent to o directly , but in the LRC scheme, it may pass through other nodes. This means that the LRC scheme sometimes takes more time to reclaim garbage.

3 Group Reference Counting

The basic idea of the group reference counting scheme is very simple. If some objects are expected to form cyclic structures, the programmer defines a group including those objects and the system keeps a reference count for the group. The reference count of a group is updated when some member of the group is referred to or discarded by an object which is not a member of the group. When a member of a group is referred to or discarded by another member of the group, the reference count of the group will not be changed. If the reference count of the group becomes zero, all the members of the group can collectively be regarded as garbage. On the other hand, each object has

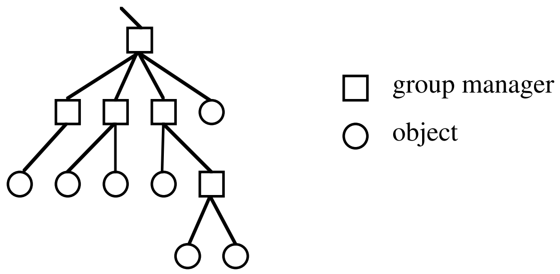

Figure 4: Group Reference Counter

a traditional reference counter to perform traditional object wise garbage collection. Some groups may possibly form cyclic structures. So we assume that groups may become members of another group. (See Fig.4)

According to our scheme, upon creation of an object, either the programmer or the system must specify a group, to which the object should belong. Currently, we are designing a language and its compiler so that groups are automatically specified by analyzing programs.

3.1 Characteristics of Groups

When an object o is a member of a group G, we say "o belongs to G." When an object o is a member of a group G_0, and G_i is a member of G_{i+1} for all i so that $0 \leq i < n$, we say "o is included by G_n."

In this paper, we assume that an object group has the following characteristics.

- When this garbage collection scheme is implemented on a distributed system, the group boundary is independent of the node boundary. In other words, some objects which reside on different nodes may belong to the same group.
- There exists one special group, the *root group*, by which all objects are included.
- All objects and groups except the root group belong to exactly one group.
- Groups are created dynamically during computation in the same way as objects. When a group is created, it initially has no members.
- When an object or group is created, a group to which it belongs is specified by the programmer. Once specified, the object or group never changes the group it belongs to.

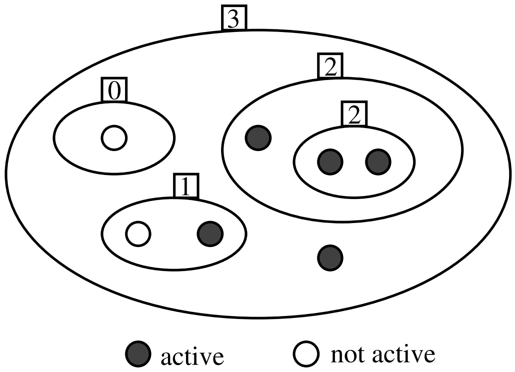

Figure 5: The tree of group managers

3.2 Examples

Objects which represent an abstract data type often refer to each other, so the programmer can specify a group corresponding to the abstract data type. Typical examples are two-way lists and ring buffers. Each cell which forms a two-way list may be collected if it is removed from the list, but the two-way list itself cannot be collected by the traditional reference counting schemes. If the two-way list has a group reference counter, it can be collected even if cells refer to each other.

In object-oriented operating systems, some modules which correspond to processes or process groups on UNIX can form groups. All storage used by a process can be reclaimed after the termination of the process if there are no interprocess references.

In programming languages which can construct static closures (e.g., Common Lisp), closures and its environment may refer to each other. In this case, closures and its environment can form a group. When all closures are discarded, closures and its environment can be reclaimed. This grouping can be done by compiler, so users of the language may not be concerned with garbage collection.

3.3 Implementation in Non-distributed Systems

3.3.1 Group Reference Counter

The system provides a *group manager* for each group. Each group manager has a *group reference counter (GRC)* and a list of its members. Each object (and group) has a pointer to a group manager. So, group managers and objects construct a two-way tree. The root of the tree is the group manager of the root group. (See Fig.5)

When an object o refers to or discards an object o', the system climbs two chains of group managers starting from o and o', and updates the GRCs on the chains until the shared group manager is found. If some GRC becomes zero, the system reclaims all members of the associated group.

3.3.2 Active Member Counter

In object-oriented concurrent systems, many objects become active simultaneously. Active objects can send messages to the other objects, so active objects should not be regarded as garbage even if they are not referred to. In the same way, "active groups"(groups which include more than or equal to one active member) should not be regarded as garbage. In order to determine whether a group is active or not, its group manager maintains an *active member counter (AMC)*, which counts the number of active member. When an object becomes active or inactive, AMCs of all groups which include the object are updated. (See Fig.6)

AMC is updated as described below.

- When an object o becomes active, the AMC of the group G_0 which contains o as a member is incremented. If the AMC of the group G_i changes zero to one, the AMC of the group G_{i+1} which contains G_i as a member is incremented.
- When an object o becomes inactive, the AMC of the group G_0 which contains o as a member is decremented. If the AMC of a group G_i changes one to zero, the AMC of the group G_{i+1} which contains G_i as a member is decremented.

The GRC of a group is incremented when the AMC of the group changes from zero to one, and decremented when the AMC changes from one to zero. So, clearly the GRC of an active group will never become zero.

Generally in an object-oriented concurrent language, an object is activated when it receives a message from another active object. If the sender and receiver of the message are included by the same group G, the AMC of G must not be zero before the message transmission. So, AMCs of all the groups which includes G will not be changed by the message transmission. This means that the overhead for updating the AMC is not affected by the number of groups which include G if message transmission takes place within G.

3.4 Implementation in Distributed Systems

3.4.1 Local Group Reference Counter

To implement the group reference counting scheme on distributed systems, the algorithm described in §2.2 can be applied. (The algorithms proposed in [12, 13] can also

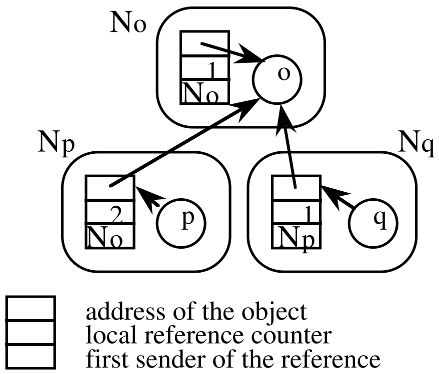

Figure 6: Active Member Counter

be applied.) Namely, when an object p on node N_p has a reference to o, N_p has *local group reference counters (LGRC)* of all the groups which include o.

When p on N_p sends q on N_q a reference to o, N_p also sends group-ids' of all the groups which include o but not include q, and LGRCs of such groups are incremented. When N_q receives group-ids' from N_p, N_q creates or updates each LGRC as described in §2.2.

Notice that when p on N_p sends q on N_q a reference to o, p must have references of both o and q. Therefore, N_p knows all the group-ids' of groups which includes o or q. So N_p can determine which group-id should be sent to N_q with the reference to o and which LGRC should be incremented only using the information local to N_p.

The "creator" of a group G is always exactly one node even if members of G is distributed among many nodes. If LGRC of G on the node which created G becomes zero, the node sends *reclaim messages* to all the members of G. When an object or a group receives a reclaim message, it can be regarded as garbage.

3.4.2 Local Active Member Counter

AMC may also be distributed by applying the same scheme described in §2.2.

Generally, in managing reference counters in a distributed system, both the sender node and receiver node of a reference need to be involved. For the same reason, managing distributed AMCs requires both the node for an activator object and the node for an activated object to be involved.

In an object-oriented concurrent system, an object becomes active only if the object receives a message from another object. Therefore, a node which makes an object active has a reference to the object. As described in §3.4.1, the node also has

LGRCs of all the groups which include the object.

If a node has LGRC of a group, the node also has a *local active member counter (LAMC)* of the group. LAMC has "the number of members which are activated by objects on the node". If LAMC is not zero, the node also remembers an *activator* of the group. The activator is an id of a node who changes LAMC from zero to one by sending a message to a member of the group.

LAMC is updated as described below.

- When an object p on a node N sends a message to an object o on the same node N, LAMCs are updated as described in §3.3.2. If an LAMC of a group G changes one to zero, the LGRC of G on N is incremented and N is remembered as an activator of G on N. Furthermore, if LGRC of G changes zero to one, N is remembered as a first sender of G.

- When an object p on a node N_p sends a message to an object o on a node N_o, LAMCs and LGRCs of all groups on N_p which include o is incremented. If an LAMC of a group G changes one to zero, N_p is remembered as an activator of G on N_p. If an LGRC changes one to zero, N_p is remembered as a first sender of G on N_p.

- When an object o on a node N_o receives a message from an object p on a node N_p and o changes its state from inactive to active, all LAMC is updated as described in §3.3.2. Furthermore, for all groups on N_o which include o,

  - If an LAMC of G changes zero to one, N_p is remembered as an activator of G and the LGRC of G is incremented. If the LGRC of G changes zero to one, N_p is remembered as a first sender of G. If the LGRC of G is incremented but its value was not zero, a discard message of G is sent to N_p immediately.
  - If G has been already active, an inactive message and a discard message of G are sent to N_p immediately.

- When an object o on a node N_o receives a message from a node N_q but o does not change its state (because it has been already active, or it does not accept the message), inactive messages and discard messages of all groups which include o are sent to N_p immediately.

- When an object o on a node N_o becomes inactive (in other words, o becomes message waiting mode), all LAMCs are updated as described in §3.3.2.

- When an LAMC of a group G on a node N changes one to zero, the LGRC of G on N is decremented. If the activator of G on N is not N, an inactive message are sent to the activator.

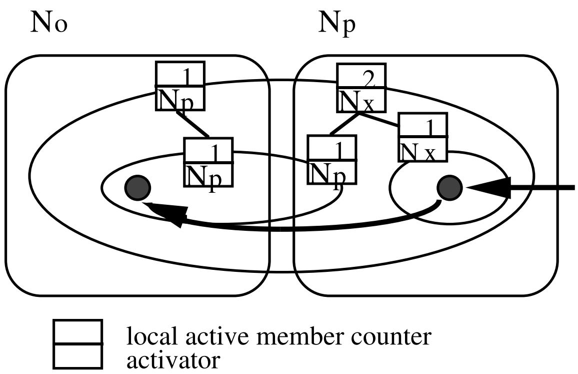

Figure 7: LAMC on each node at the time when o receives a message from p

- When a node N_p receives an inactive message of a group G, the LAMC of G on N_p is decremented.
- When an LGRC of a group G changes one to zero, a discard message or reclaim messages are sent as described in §3.4.1.

(See Fig.7)

4 Efficiency

4.1 The Amount of Internode Messages

The overhead of distributed group reference counting scheme depends on neither the number of nodes nor the amount of data used. It depends on the amount of the computation and the nesting levels of groups.

This scheme requires three types of extra messages for garbage collection: *reclaim, discard, inactive*. The number of reclaim messages are less than or equal to the number of members of groups which become garbage. The number of discard messages are less than or equal to the some of the product of the number of internode references and the maximum nesting level of groups and the product of the number of internode messages and the maximum nesting level of groups. The number of inactive messages are less than or equal to the product of the number of internode messages and the maximum nesting level of groups.

If a message transmission takes place within a group, the number of messages is less than the above worst case. It is expected that the computation of most distributed applications will be local to groups.

4.2 Improvement of Efficiency

The following techniques may reduce the amount of messages.

- Messages for garbage collection do not have to be delivered immediately. They can be delayed without affecting the computation speed. (Instead, garbage remain on the system for a longer period.) So messages for garbage collection can be buffered and delivered as packets which contain other important messages. Thus the cost of communication for garbage collection will be substantially reduced.

- A member which has the same lifetime of a group to which it belongs need not be reference-counted. For example, in a group which represents a queue, objects representing the head and the tail of the queue will never become garbage until the whole queue becomes garbage.

5 Problems and its Solution

5.1 Specifying Groups

It is very cumbersome for the programmer to specify a group every time when he creates objects. If a programming language is designed in such a way that it includes appropriate language constructs which form cyclic structures, the compiler can automatically detect such object groups. We are designing such a language constructs for an object-oriented concurrent language.

5.2 Unpredictable Cycles

It is often the case that cyclic structures are formed dynamically, so the programmer cannot predict which objects form a cyclic structure. The group reference counting scheme cannot reclaim such objects. In order to reclaim such objects, mark & sweep schemes may be combined to reference counting schemes. For example, if the programmer can specify that some groups should use a mark & sweep scheme instead of the reference counting scheme to manage its members, all unreachable members in the group can be reclaimed. Such a local mark & sweep scheme within one group does not require all nodes on the distributed system to cooperate.

6 Conclusion

We proposed new extensions of reference counting schemes, the local reference counting and the group reference counting.

Reference counting schemes on distributed systems do not requires synchronization of all nodes and the overhead does not depend on the number of nodes. So, it works efficiently on distributed systems containing large number of nodes. The local reference counting scheme is almost as efficient as other distributed reference counting schemes currently proposed, but it is easier to implement.

The group reference counting scheme inherits all the advantages of the traditional reference counting schemes and it can also reclaim cyclic structures if they are expected by the programmer. So it can be used on object-oriented systems which often create cyclic structures.

The distributed algorithm of the group reference counting scheme can be used for object-oriented concurrent language on distributed systems. In order to analyze traffic of messages and the number of objects that are really garbage but cannot be reclaimed, we are conducting experiments with some typical application programs. The result of our performance simulation will be reported in a subsequent paper.

Acknowledgments

The authors wish to thank Etsuya Shibayama and Takuo Watanabe for numerous helpful discussions. Also the authors thank Takayuki Nakagawa for writing the simulation program of this GC scheme.

References

[1] D.Decouchant: DESIGN OF A DISTRIBUTED OBJECT MANAGER FOR THE SMALLTALK-80 SYSETM, ACM OOPSLA, pages 444-450, 1986.

[2] John K.Bennett: The Design and Implementation of Distributed Smalltalk, ACM OOPSLA, pages 318-330, 1987.

[3] Eric Jul,Henry Levy, Norman Hutchinson,and Andrew Black: Fine-Grained Mobility in the Emerald System, ACM Trans. on Computer Systems, Vol. 6, No.1, pages 109-133, February 1988.

[4] J.Cohen: Garbage collection of linked data structures, *Computing Surveys*, 13(3) pages 341-367, September 1981.

[5] D.R. Brawnbridge: Cyclic reference counting for combinator machines, *In Proc. Functional Programming Languages and Computer Architecture*, pages 273-288, Springer-Verlag LNCS 201, September 1985.

[6] D.P.Friedman, D.S.Wise: Reference counting can manage the circular environments of mutual recursion,*Inf.Process.Lett.*,8(1) pages 921-930, January 1979.

[7] D.G.Bobrow: Managing Reentrant Structures Using Reference Counts, *ACM Trans. on Programming Languages and Systems*, 2(3) pages 269-273, July 1980.

[8] A.Deb: Parallel garbage collection for graph machines, *In Proc. of the Santa Fe Workshop on Graph Reduction*, pages 252-264, Springer-Verlag LNCS 279, September 1986.

[9] K.Shimizu, M.Maekawa, J.Hamano: Hierarchical Object Group in Distributed Operating Systems, *PROCEEDINGS The 8th International Conference on Distributed Computing Systems.* , pages 18-24, San Jose, California, June 1988.

[10] Y.Honda, A.Yonezawa: Debugging Concurrent Systems Based on Object Groups, In S.Gjessing and K.Nygaard, editors,*Proc. of European Conference on Object-Oriented Programming(ECOOP)*, LNCS 322, Springer-Verlag, 1988. *Revised version in* [14].

[11] Claus-Werner Lermen, Dieter Maurer: A Protocl for Distributed Reference Counting, *Proc. 1989 ACM Conference on Lisp and Functional Programming*, pages 432-443, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Masachusetts, August 1986.

[12] D.I.Bevan: Distributed garbage collection using reference counting, *In Proc. PARLE Parallel Architectures and Languages Europe*, pages 176-187, Springer-Verlag LNCS 259, June 1987.

[13] B.Goldberg: Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme, *SIGPLAN Conf. on Prog. Lang. Design and Imp.*, pages 313-321, June 1989.

[14] Yonezawa, A. (ed.): *ABCL: An Object-Oriented Concurrent System — Theory, Language, Implementation and Applications —*, The MIT-Press, March 1990.