Distributed Garbage Collection Algorithms

Stefan Brunthaler

***Abstract*— This seminar report presents adoptions of uniprocessor garbage collection techniques which are able to operate in a new context: distributed systems. We see that their application to distributed system poses new problems and that there is currently no algorithm available solving, or even addressing, all of them.**

The presented algorithms are structured according to their belonging family: reference counting, mark-and-sweep and stop-and-copy garbage collectors. Hybrid collectors that simultanously apply different techniques are also presented.

***Index Terms*— Distributed garbage collection algorithms, distributed reference counting, distributed tracing.**

I. Introduction

THE need for distributed systems has been growing in the past, and will be growing in the future. Their proliferation and success is out of the question. Several systems, frameworks and middleware products have been crafted in order to relief the burden of designing and implementing such systems. Due to their inherent complexity and partly opposing requirements, such as reliability and high performance–particularly difficult to achieve or guarantee in the context of networking–, no silver bullet has been found.

Uniprocessor garbage collection has been a success story. Programming languages have adopted this technique, and–after intial scepticism–programmers got used to the technique, since it considerably eases the tedious and error prone task of memory management. In distributed systems, the task of memory management is even more error prone, because of memory references between systems. Consequently the need for distributed garbage collection has emerged, and has been an active field of research over the past years. However, the number of proposed algorithms which only partly satisfy the expected functionality of a distributed garbage collection algorithm demonstrates that the field is still active, needs further investigation, and no generally applicable/accepted algorithm has been devised.

This seminar report draws its information mainly from three papers, namely Chapter 15 of Jones and Lins book on garbage collection [1], Plainfosse and Shapiro's survey of distributed garbage collection algorithms [2], and Abdullahi and Ringwood's garbage collecting the internet article [3]. Where appropriate, direct citations to other papers are made.

II. Background

A. Definitions

"A system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing." [4]

"A distributed system is a collection of independent computers that appear to the users of the system as a single computer." [5]

B. Problem Description

Distributed systems are applications which are deliberately designed to run on physically seperated systems, which are interconnected by a common network infrastructure, either a local area or a wide area network. The application of a distributed system architecture is motivated by the following advantages over "traditional" single-processor systems, among others:

- *Reliabilty*, i.e. satisfied by several machines which operate on replicated data sets or are in fact a replicated system, e.g. used in mission critical applicatgions, like control of nuclear reactors or aircrafts,
- *Performance*, i.e. multiple interconnected computers can reach the performance of a mainframe computer but cost only the fraction of one,
- *Functional distribution*, e.g. client/server systems,
- *Application domain*, e.g. automatic teller machines, cash register and inventory sytems, and roboter assembly lines.

What is important, however is–as a definition in the previous section suggests–that the distribution of some parts of a system is completely *transparent* to the user, and increasingly to the programmer too, especially since the advent of RPC (the *Remote Procedure Call*), CORBA (the *Common Object Request Broker Architecture*), and Java's RMI (the *Remote Method Invocation*).

According to Tanenbaum [5] there are several levels of transparency:

- Location transparency, where users cannot tell where resources are located,
- Migration transparency, where resources can move at will without changing their names,
- Concurrency transparency, where users can share resources automatically,
- Parallelism transparency, where activities can happen in parallel without users knowing,
- Replication transparency, where users cannot tell how many copies exist.

As we can see, a lot of concerns have to be taken into account for building distributed systems. This is what essentially makes garbage collection in distributed systems an interesting problem. Our view on distributed systems focuses on the system view of a distributed system, because garbage collection happens at this level. In order to properly describe the presented algorithms and problems, we use the following nomenclature, see also Figure 1:

- inter and intra, determining whether we are dealing with relationships within (intra) a given computer system or site, or between (inter) a set of given computer systems or sites,

- local and global systems, determining whether we are dealing with a single computer system within the distributed system (local), or all participating systems (global), intrasite relationships can only be local, whereas intersite relationships can only be global,
- local and global address space, determining whether we are dealing with a single computer system's or the virtual global address space built on top of the local address spaces of each participating computer system,
- local and global objects, determining whether we are dealing with objects local to their address space, or globally accessible objects, sometimes referred to as public objects,
- local and global references, local references are sometimes described as pointers, whereas the term references is generally used to denote global references, i.e. inter-site references from a global object to another, either global or local object at another site,
- import and export record, an import record imports a global, remote, reference to the local system, whereas an export record is used for exporting references from local public objects to the clients,
- local and global cyclic data structures (referred to as cycles),
- client and owner computers, and systems respectively, indicating whether a system references another system's public object (client), or a local object is referenced by other systems, causing it to be promoted to a global/public object; the latter is said to "own" the object, thus referred to as owner.

In the discussion of distributed garbage collection algorithms we are going to encounter recurring problems. These problems are two-fold: a) problems of the garbage collection domain, and b) problems from the domain of distributed systems. Problems of the category a mainly deal with the properties of a garbage collection algorithm [1], [3]:

- premature reclamation of accessible objects, the *unsoundness*,
- failure to reclaim all extant garbage, yielding floating garbage, the *incompleteness*,
- communication overhead, i.e. too many messages are exchanged,
- cycles, i.e. local and global cycles have to be reclaimed properly,
- synchronization, i.e. information about the state of global references have to be exchanged, since it is not possible to locally determine whether a reference is garbage or not,
- robustness, i.e. an algorithm should be able to recover from communication problems.

The major issue in category b are problems caused by unreliable networking, wich are:

- communication failure,
- message loss,
- message duplication,
- out-of-order delivery/non-causal delivery,
- delayed message delivery/latency.

Both problem sets are closely interrelated since problems of category b are always somehow causing problems in the a category.

III. Reference Counting Approaches

Because of their acyclic nature, most garbage collection algorithms are not able to reclaim cyclically referenced garbage. Some algorithms improve on this downside and extensions to the presented algorithms, which take care of this problem are explicitly mentioned.

A. *Uniprocessor Adoption*

The uniprocessor algorithm can fairly easy be adopted to suit distributed system requirements. The increment and decrement messages used in the uniprocessor algorithm just have to be changed in order to send corresponding messages for intersystem references. Consequently every creation of a new reference to a global object, duplication of an existing one and deletion of a present reference triggers a message, which is sent to the owner of the global object, causing it to increment, or decrement respectively, its reference count.

This straightforward adoption, however, has several drawbacks, mainly caused by messaging failures due to unreliable networking. Non-causal delivery of messages and message duplication can lead to unsoundness, whereas the problem of incompleteness arises when messages are lost. These problems can partly be eliminated by acknowledging the control messages used to indicate changing reference counts, however, this causes another disadvantage: communication overhead. As mentioned in the beginning of this section, the algorithm is not able to collect cycles.

B. *Weighted Reference Counting*

Original contribution in [6] and [7]. Weighted Reference Counting, henceforth abbreviated as WRC, is a major improvement over the original reference counting algorithm. This advantage manifests itself in a considerable reduction of communication overhead, due to the absence of *race conditions*. Race conditions originate by a non-causal delivery of increment and decrement messages, ultimately causing undsoundness, since an incoming decrement message can cause an object to be reclaimed, even if a chronologically earlier sent increment message is on its way, yet not received by the owner of the object.

The application of redundant acknowledge messages in the original algorithm cause the communication overhead. Here WRC emerges with a meaningful abstraction solving the problem of race conditions: weights. Instead of simply counting all references to a public object, the object now carries an additional weight, which is initialized to the maximum possible value of its designated storage area. If a reference is created, its weight field is also intialized to the maximum value, causing them to have equal weights, see Figure 2. On reference duplication, the weight of the source reference is halved. One half remains in the source reference, the other half is assigned to the duplicated reference, see

Figure 3. Finally, when a reference is deleted, its weight is used to decrease the weight of the global object, see Figure 4. Consequently, the weight of the global object is at all times equal to the sum of all weights of all the references to it. In order to maintain this invariant, considerably less messaging is required. One message is necessary for creating a reference, one for duplication, and one for deletion. Because control messages only decrement weights, no race conditions occur, which renders the additional acknowledge messages–necessary in the straightforward approach–superfluous.

Unfortunately, this algorithm has serious flaws too. Suppose the maximum possible weight is 2^n (using n bits of storage), and every duplication of a reference causes this weight to be halved, only n references can be made. One simple way to overcome this major disadvantage, is to use *indirections* [7]: Whenever the total weight drops to 1 (2^0), an indirection reference is created and used for duplicating. If a reference to an object contains an indirection reference, and this indirection is not co-hosted at the owner of the global object, however, two, instead of normally one messages are necessary to access the actual global object. Hence with every additional indirection layer, global object access requires more messages. In a worst-case scenario, a *domino* effect can occur, where a reference is followed along its indirections and finally linked back to its own local address space ([8], see also Figure 5).

Although this algorithm reduces communication overhead and avoids race conditions, it is still not applicable when used in conjunction with an unreliable network infrastructure.

Several extensions to this algorithm were made, in order to enable it to collect cycles, e.g. [9]; however, this algorithm uses a local mark and sweep collector, as suggested by Lins, which puts it in the family of hybrid collectors.

C. Indirect Reference Counting

Original contribution in [10], [8] and [11]. Indirect Reference Counting, henceforth abbreviated as IRC, is another interesting approach to the distributed reference counting problem. Besides the field for storing the reference to a global object, and the field for storing the current count value, IRC requires an additional field within its memory cell. This additional field is used for storing a reference to its parent object, i.e. the object which created the reference. Consequently, all state information is maintained within its references. Through the parent field, a so called *inverted diffusion tree* is created, which is used for garbage collection purposes only (Figure 6). Every new reference is the root of its own diffusion tree, with no copy count and no parent data available. When a reference is duplicated, the source reference becomes the parent of the duplicated one, and the source's copy count is increased by one; the object reference itself, however, points to the *same* global object in both references, the source and the duplicated one. This ensures that all references to a global object always use shortest, i.e. the most direct, path to it, thus keeping message exchange acceptable. Upon reference deletion, the copy count of its parent is decremented, and the reference is removed. Only references with copy count equal to zero can be reclaimed, which applies only to leaves in the inverted diffusion tree (Figure 7). Non-leaf references can also be removed from the diffusion tree, however, their reclamation is deferred until their copy count drops to zero (Figure 8 and 9 respectively). Ultimately this leads to floating garbage. The algorithm is reasonably efficient: both, reference duplication and deletion require only one message to be sent.

D. Trial Deletion

Original contribution in [12]. The algorithms idea is to emulate deletion of objects, suspected to be garbage. After this trial deletion, the reference count of the affected objects is incremented again, in order to continue proper operation. This procedure is able to collect garbage cycles. Despite the interesting approach, this algorithm has disadvantages: a) it cannot detect mutually referencing cycles, b) it needs a heuristic to detect which objects probably are garbage.

E. Reference Listing

The reference count of global references is kept in the according import record, i.e. every reference to a global object, is reflected in the reference count of the import record. Thus, the count associated with every import record should invariably reflect the number of export records referencing it.

As mentioned earlier, unreliable messages break this invariant, since, e.g. a duplicated message causes this import record's reference count to be decremented two times. This behaviour is appropriatly characterized in [2] as being *non idempotent*. If an algorithm was able to adequately deal with duplicated, or lost messages respectively, it would be *idempotent*. Instead of simply counting the references within the import record, a new import record is created for each new reference, either allocated or duplicated, see Figure 10. This adds the desired property of *idempotency* at the expense of some memory overhead.

Furthermore, the algorithm is able to reasonably react to crashed or unavailable systems. The algorithm is able to compute the set of active clients by prompting them to send live messages. Now the algorithm can decide what to do with the reference it holds to unavailable systems. It can either decide to:

1) keep the references and hope that an unavailable system recovers,
2) reclaim the objects which were references by this system, assuming that it has crashed.

IV. Tracing Approaches (Mark & Sweep)

This section deals with the tracing family of distributed garbage collection techniques. The heading of each section is named after its original authors and an alternate, however, expressive and useful name taken from [2] within parenthesis. Since tracing garbage collectors are inherently cyclic, they are also able to detect and reclaim cyclic data structures. This feature is exposed of all of the following techniques and not mentioned any further.

A. Uniprocessor Adoption

The straightforward adoption of the uniprocessor procedure results in scheme, where a global master site tells each participating system to suspend mutation and to start the mark phase. Local garbage can easily be reclaimed without synchronization. Global garbage however, requires each client system, to send a mark message to the corresponding owner in order to mark distributed data structures. (Any system can be a client of multiple owners, which means that this system is bound to send a mark message to each owner it holds client references from). This global marking phase is complete, when the master site is informed that all marking messages were received, and there are no more messages in transit. After marking the distributed system can enter the sweep-phase, which needs no distributed counterpart, since all sweeping can be done locally.

B. Hudak and Keller (Mark-Tree)

Original contribution in [13]. The Hudak and Keller algorithm is based on the concurrent on-the-fly collector presentend by Dijkstra *et al*. in [14], which allows for concurrent mutation and collection. The mark-tree algorithm was conceived for functional languages, and assumes that there is a single root in the distributed computation graph of objects. The modification of Hudak and Keller to the original algorithm allows the distributed system to concurrently mutate and collect on each participating system. Only the collection phases of *identification* and *reclamation* have to globally synchronized. Concurrent modification and collection of the distributed computation graph uses two queues: one for mutator and one for collector tasks. A task is the smallest autonomous unit of processor activity. Each task locks the objects it intends to modify, such that no change is lost.

Dijkstra's algorithm uses a recursive marking scheme for tracing the computation graph, which marks the nodes using a tricolor scheme. In order to concurrently mark the distributed computation graph, the recursive marking is replaced by so called *marking tasks*. For each child of a given node, a new mark task is created. Since each task runs locally, a global child reference requires to spawn a child task on the referenced system. Each child holds a reference to its parent task, and notifies its parent by a so called *uptree task*, which means that its own marking phase is completed. Hence marking starts by spawning a mark-task on the root of the computation graph, and correspondingly ends when an uptree task is spawned from the root.

Before marking, all cells have the color *white*, which denotes that it has not yet been visited. A memory cell becomes *gray*, if it has been identified, and seperate mark tasks have been created for each of its children. A gray cell can become *black* when it has received an uptree task from all of its children. Concurrent mutator activity changes the distributed computation graph, if, e.g. a new memory cell is allocated, it automatically is black. After marking is complete, white memory cells are considered to be garbage.

Additionally, the algorithm detects other kinds of garbage: *irrelevant* tasks and *dormant* subgraphs. Irrelevant tasks occur due to "a curious relationship between the computation graph and task queues" [13]: Because of speculative parallelism, it is possible to have active tasks referencing white cells, i.e. if marking is complete, these irrelevant tasks have to be considered garbage too. Dormant subgraphs owe their existence to the same curious relationship. They describe nodes, or subgraphs respectively, which happen to be reachable from the root node, but their semantics dictate that no task can ever propagate work there. By tracing from the tasks themselves instead of the root, dormant subgraphs can be detected. Eventually both of them are considered to be garbage, and have to be reclaimed.

C. Hughes (Tracing with Timestamps)

Original contribution in [15]. Hughes' algorithm uses a simple, yet powerful, observation to identify distributed garbage cycles: instead of simply marking nodes, the algorithm propagates current timestamps, i.e. live objects and references will always have a "recent" timestamp, whereas garbage objects timestamps will always remain constant, since no new time is propagated to them. Thus reclamation collects all objects, which timestamps indicate that they are "older" than the current time with a variable threshold. In order to use timestamps, a synchronized global clock is necessary. This requires a reliable network infrastructure.

The algorithm applies a structure as defined in [16]. Each participating system in a distributed network does its own *local* mark and sweep, upon completion of its local garbage collection phase, it informs the other systems about the references to global objects it retains. This procedure is unable to identify and collect distributed cycles. Hughes' timestamps solves this problem. The local clock is intialized by a global synchronized time. The local mark and sweep propagates the timestamps from its roots down to the leaves of the computation graph. After completion of local garbage collection phase, the systems exchange the timestamps of their references to each other, i.e. the timestamps of the local export reference table are sent along the way to their corresponding import records. If their timestamps are below the received ones, they are updated. Ultimately this leads to the global clock advancing for live objects ("ticking"), and constant for dead ones. In order not to prematurly reclaim live objects, the threshold T for reclamation is set to the oldest global time fragment issued at the beginning of the global garbage collection phase.

One major drawback of the algorithm is that is *not robust*, since a failed participating system prevents increasing the global timestamp, consequently blocking garbage collection on all other spaces. This is also true for slow systems, unwilling to initiate a local garbage collection. Even worse is that this behaviour is also exposed when the slow or failed system does not even have any global references.

D. Liskov and Ladin (Logically Centralized Reference Service)

Original contribution in [17]. As the name "Logically Centralized Reference Service" suggests, the Liskov and Ladin algorithm uses a completely opposing approach, when compared

to the previous ones. They use a centralized service, which is physically replicated on every participating system to achieve fault tolerance and high availablity, at the cost of redundance. As in the structure suggested in [16] the algorithm uses local mark and sweep garbage collectors for collecting the local heap. They also report the bookkeeping information kept in their import/export record tables to the central service, which uses this information to build a graph of global references. Now, a mark and sweep garbage collector is run against this graph of global references. Finally, the results of this ”global” garbage collection is distributed again, such that the just identified global garbage can be collected locally on each node.

Using this original description, Rudalics [8] came up with a counterexample, which proved that this algorithm in some cases is not correct. This behaviour is due to the fact that the correctness depends on the traversal order of a computation graph by the local garbage collector. Rudalics proposes two solutions to solve this problem. However, his suggestions are costly and complex, which is probably why the original authors chose to adopt Hughes algorithm, which solved their problem too.

E. Lang, Queinnec, Piquer (Tracing with Groups)

Original contribution in [18]. This algorithm belongs to the hybrid family of distributed garbage collectors, since it uses a tracing garbage collector for local collection and a reference counting algorithm for collecting distributed garbage. Moreover it is robust, since it relies on groups for garbage collecting. These groups can be reorganized such that failed or unavailable computer systems are omitted. The algorithm depends on timouts and message acknowledgements to determine whether a system is cooperative or not. Additionally it is also possible for a system to be part of multiple groups, such that groups can be nested. To seperate the groups from each other, every group uses its own unique identifier for collection purposes. Like suggested in Ali [16] the algorithm uses a a tracing garbage collector for local collection issues, however, no further details of Ali’s algorithm are applied.

The algorithm is by far the most complex presented in this report. The following steps summarize its operational behaviour:

1) **group negotiation**
2) **initial marking**: Import records transmit their marks to their corresponding export records. Import record marks can either be *hard* or *soft*. Export record marks can either be *hard*, *soft*, or *none*. A *hard* import record implies that it is referenced outside of the current group, whereas a *soft* import record indicates that it is–if at all–only accessible from within the group from a non-root object. (Compare figures 15, and 16 respectively)
   The initial marks are computed locally to the group by means of reference counters:
   a) backup reference counters of all import records (on every system),
   b) for every export record on every system within the group, a decrement message is sent to the corresponding import record, if, and only if, the system carrying the import item is belonging to the current group,
   c) after all messages in transit have been received, every system traverses its import records, which are marked *hard*, if their reference count is greater than zero, and marked *soft* else,
   d) the initial reference counters are restored from their backup copies of step 1, 2a.
3) **local propagation**: Local propagation uses two marking phases. Export records are initally marked *none*. After completion of the local tracing, all *none* marked export records are garbage, and can thus be safely reclaimed. The reclamation causes the reference counting algorithm to send decrement control messages to the corresponding import records. If their reference counts drop to zero, they are reclaimed by the distributed garbage collection algorithm, regardless of their mark. (Compare figure 17, and 18 respectively)
   a) First marking is performed using references from *hard* import records and roots. Any export record traced by this first step is marked *hard*,
   b) The second marking proceeds from *soft* import records, which marks previously unvisited export records *soft*.
4) **global propagation**: A previously marked *hard* export record has to propagate its mark to the corresponding import record. (Figure 19)
   If a new global reference is established, its import record is always marked *hard*, since it is obviously accessible from a root.
5) **stabilization**: The group is stabilized when there are no more messages in transit, that carry marking messages (can only be hardening messages). Additionally it is required that every system has propagated all relevant *hard* marking messages.
6) **cycle reclamation**: All accessible import records are *hard*. Any *soft* import records indicate that they are not accessible anymore, hence, their established references to their local objects should be dropped in order to enable the local garbage collector to reclaim the local objects. Consequently, any *soft* import record removes its reference to the local object, by referencing *null* instead. (Figure 20)
   Since the *soft* import records are not accessible anymore, their corresponding export records are going to be marked *none* during the next garbage collection, ultimately causing them to be reclaimed. This causes the reference counting algorithm to send a decrement control message to the import record, causing its reference count to drop to zero, and in consequence being collected by the reference counting mechanism. In case this import record belonged to a cycle, eventually every import record belonging to the cycle receives a decrement control message from their equally inaccessible export records. Finally all their reference counts drop to zero, and let the reference counting reclaim them.

(Figures 21 and 22)

7) **group dismission**

V. Copying Approaches (Stop & Copy)

Nothing but just a few distributed garbage collectors using this techique have been developed. Therefore, only one algorithm is presented here representatively. The algorithm (as described in [19]) is an adaption of an incremental scavenger as used by Baker in 1987 [20].

The memory of a system is divided as following:

1) a small space for storing references from import records, the so called *root space*,
2) the remaining space is split into two semispaces:
   a) the *fromspace*,
   b) the *tospace*.

Each of these semispaces is used for storing both, local and remote references, i.e. references to export records. However, these references are not mixed arbitrarily, but stored at different locations within the semispace:

- the *lower area* of a semispace is used for local references exclusively,
- the *upper area* is used for storing remote references exclusively.

In order to store remote references in the upper area of a semispace, the allocation process on the upper side of the semispace mirrors exactly the process of allocation on the lower side, with changed signs of course, otherwise the global references would be allocated in the tospace, which is of course undesireable.

Like the mark-tree algorithm (see section IV-B) of Hudak and Keller, this algorithm was conceived for a functional programming language, viz. LISP, therefore assumes that there exists a global root in the distributed computation graph.

Uniprocessor generation scavenging techniques are not able to detect cycles by default. Since the algorithm is based on the idea, it is also not able to reclaim distributed garbage cycles.

VI. Conclusion

As mentioned in the introduction, no 100 percent complete approach to distributed garbage collection is available. Consequently it is going to be an active research area for the time being. Among the presented algorithms, surely the algorithm of Lang *et al*'s algorithm (IV-E) provides the most advanced solution, with the only disadvantage being the large amounts of floating garbage it can accumulate. Otherwise it is very well suited to the inherent requirements of distributed systems.

It is clearly demonstrated, that some of the design characteristics of garbage collection algorithms, as well as the requirements for them, are opposing. Improving an algorithm along one dimension, more often than not causes it to deteriorate along another–possibly also important–dimension.

Most of the algorithms presented bear a noticeable resemblence to their uniprocessor counterparts, indicating that perhaps a completely new approach of the techniques to the area of distributed systems might yield better results.

Appendix I
Figures

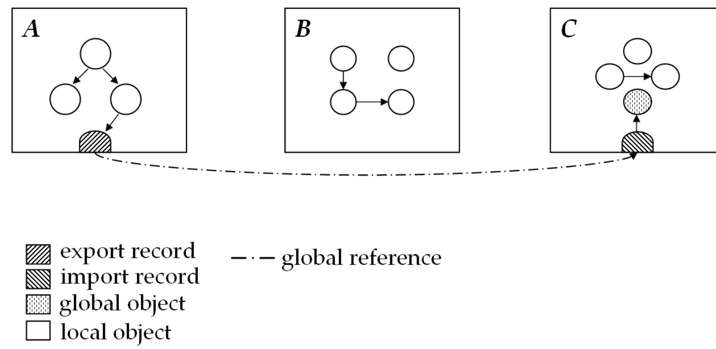

Fig. 1. Key for figures. Squares denoted by letters A, B, and C respectively are systems.

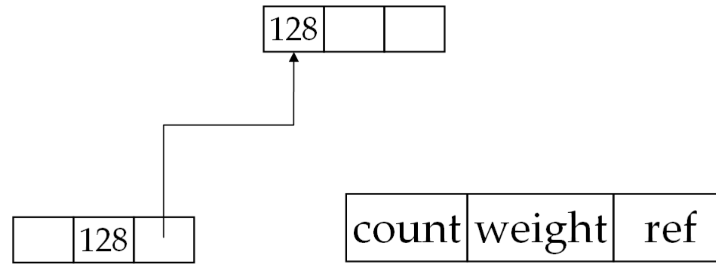

Fig. 2. Creation of a new reference using Weighted Reference Counting. Memory cell layout.

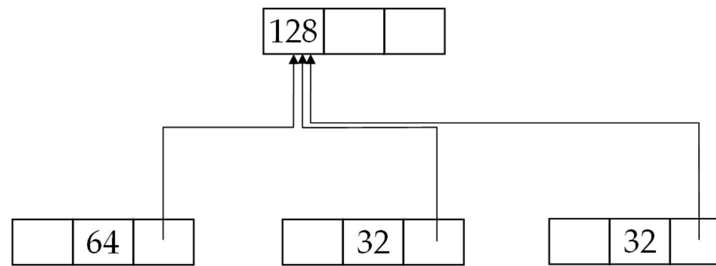

Fig. 3. Duplication of an existing reference using Weighted Reference Counting.

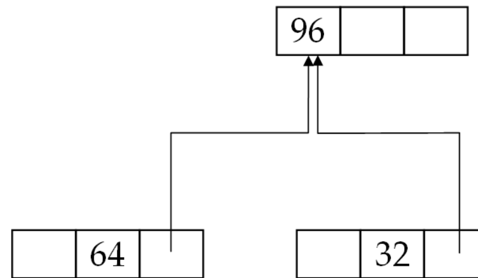

Fig. 4. Deletion of a reference using Weighted Reference Counting.

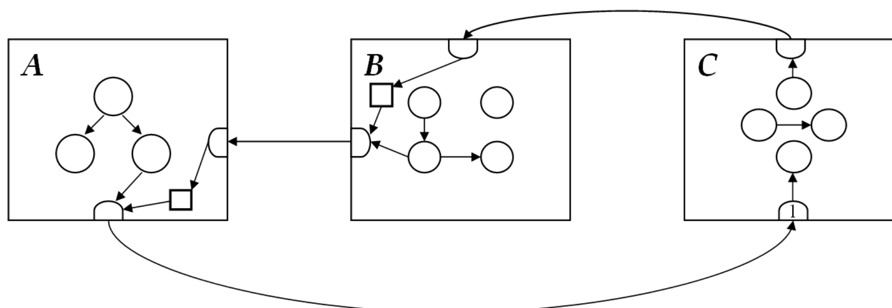

Fig. 5. Domino effect as observed by Rudalics [8]. The small squares between import and export records in systems A and B denote indirection memory cells.

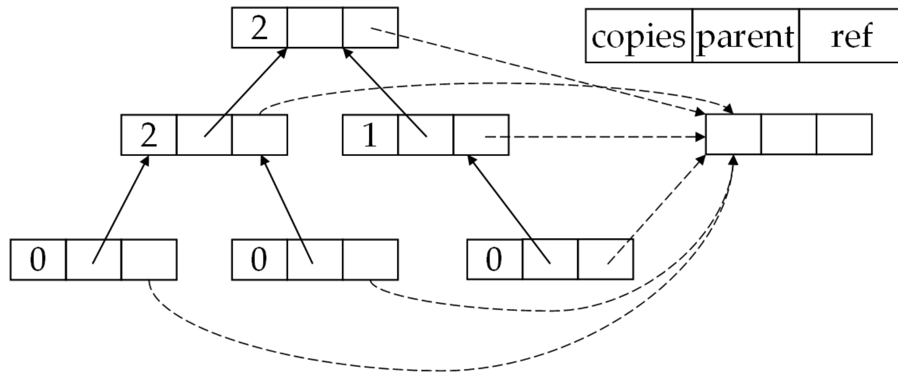

Fig. 6. Inverted diffusion tree. Resulted by additionally storing parent references in the memory cells.

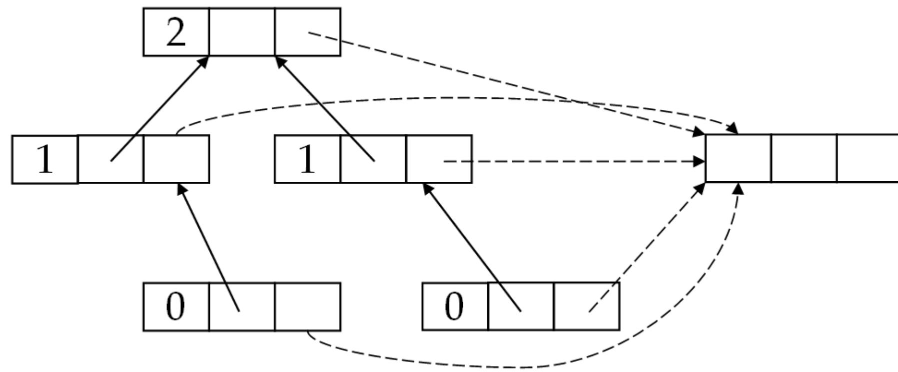

Fig. 7. Inverted diffusion tree after left leave node has been deleted. Copy count of its parent has been properly adjusted.

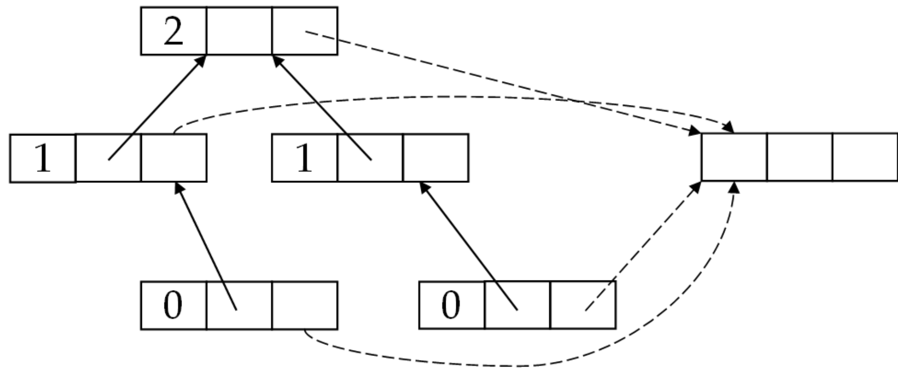

Fig. 8. Inverted diffusion tree after deletion of the reference from the parent node of the right outer leaf to the right reference cell. Cannot be reclaimed since its copy count is greater than zero, leads to floating garbage in consequence.

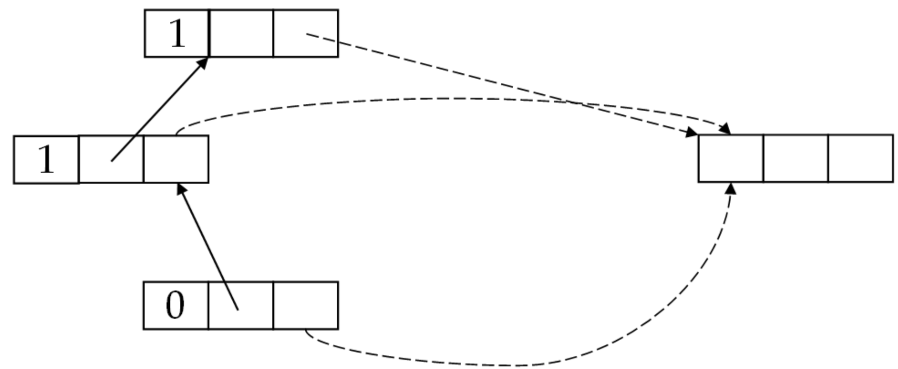

Fig. 9. Inverted diffusion tree after deletion of the outer right leaf node.

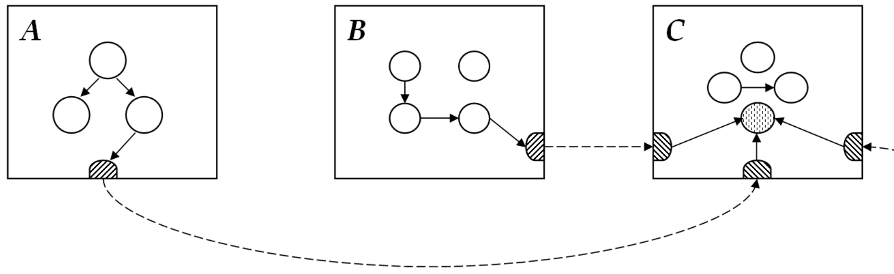

Fig. 10. Reference Listing. Every global reference, from A to C, and B to C respectively, gets its own import record.

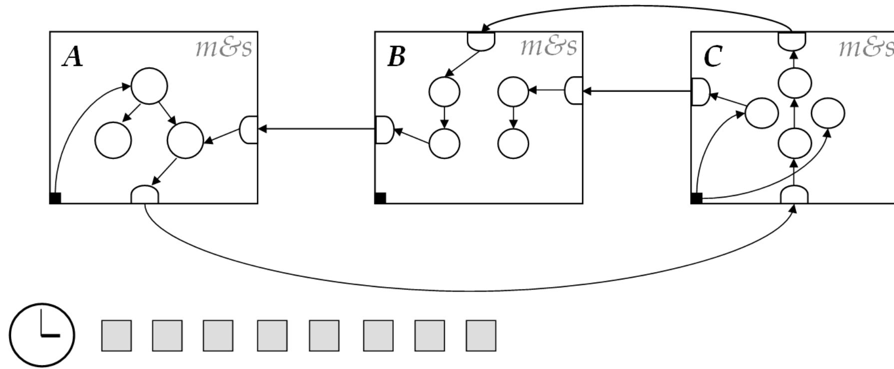

Fig. 11. Hughes' tracing with timestamps. Initially all objects are unmarked.

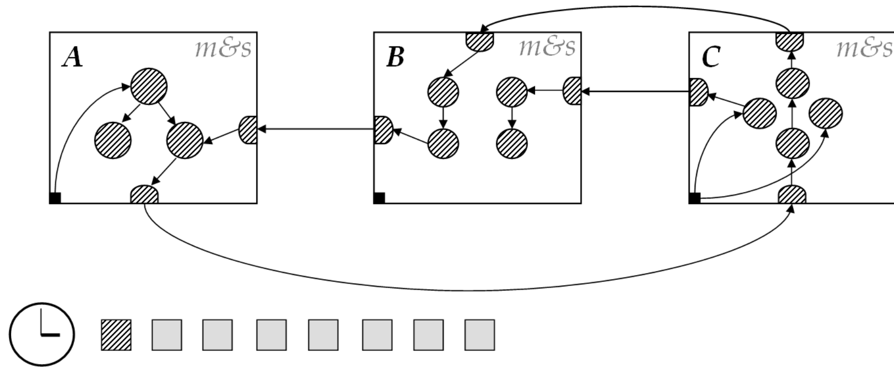

Fig. 12. Initial timestamp is propagated along the distributed computation graph.

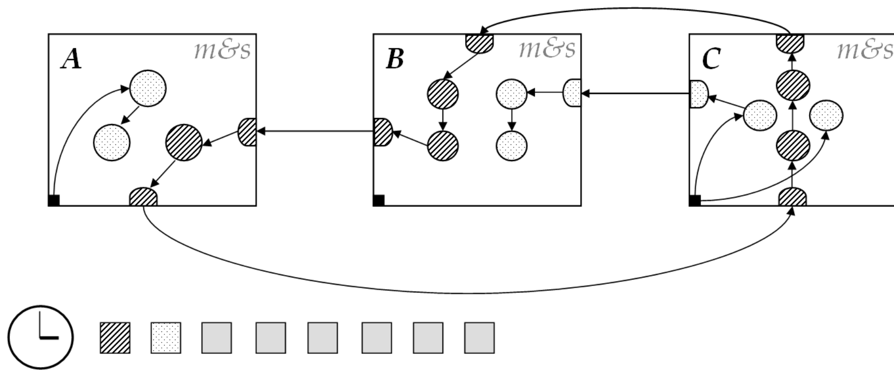

Fig. 13. Deleting a reference in system A, creates a large distributed garbage cycle. The next timestamp is propagated through the distributed computation graph.

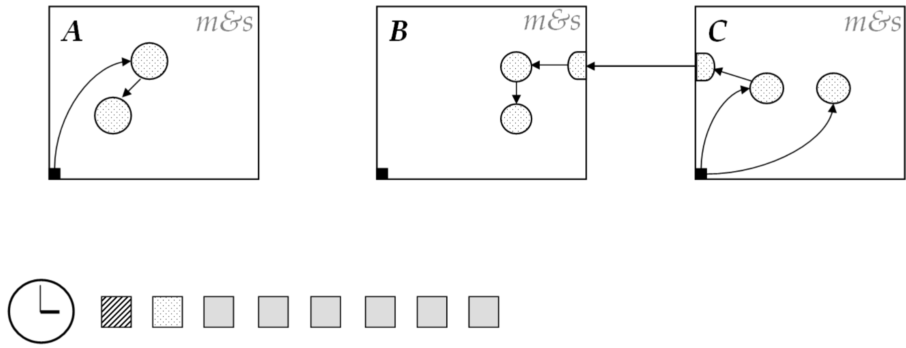

Fig. 14. After the system reclaimed the distributed garbage cycle.

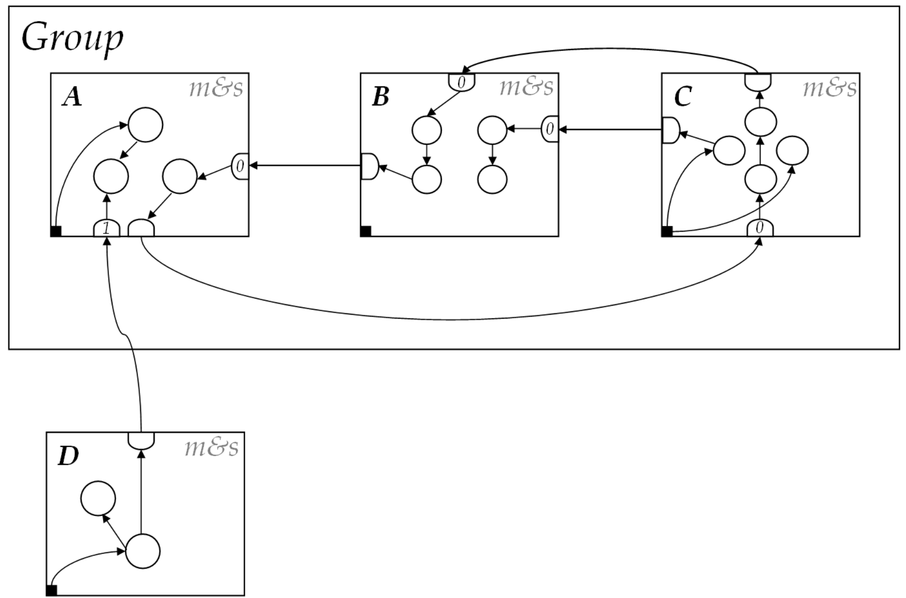

Fig. 15. Piquer's algorithm. Initial graph layout and inital reference counts.

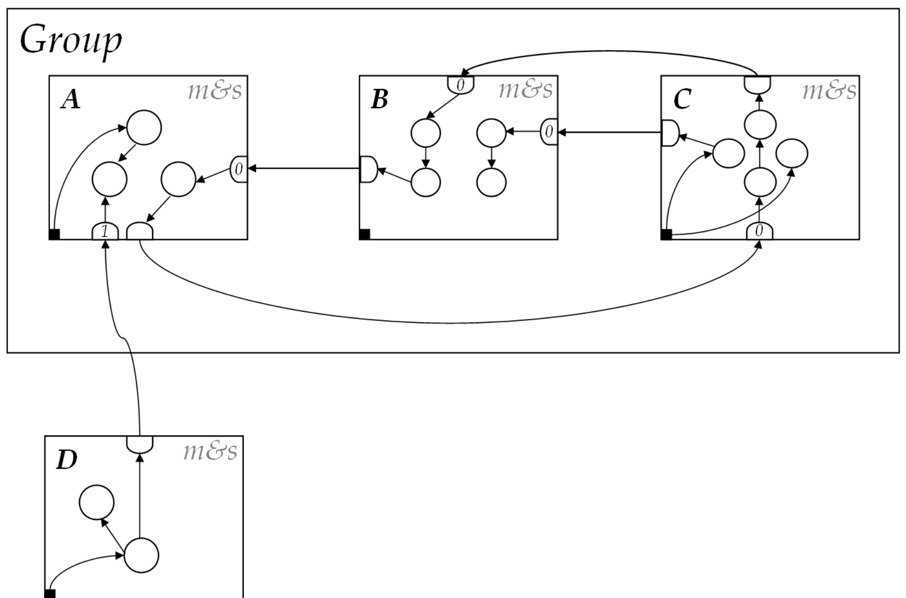

Fig. 16. Piquer's algorithm. After initial marking phase, deciding whether import records are hard or soft, part 1.

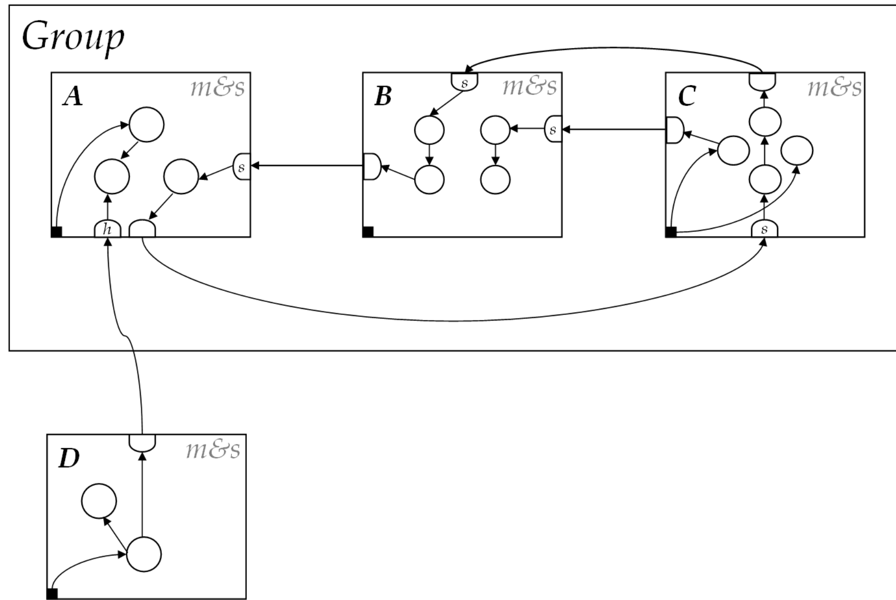

Fig. 17. Piquer's algorithm. After initial marking phase, deciding whether import records are hard or soft, part 2.

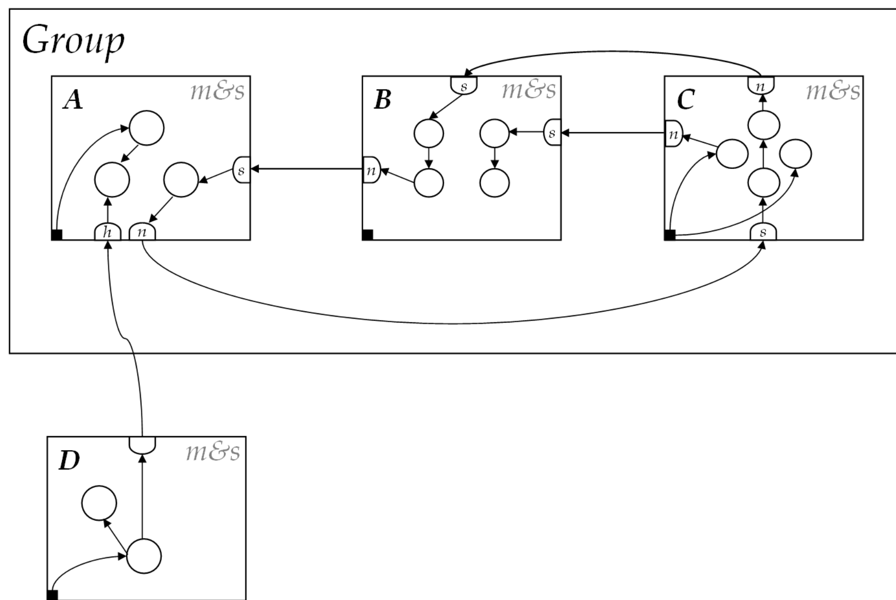

Fig. 18. Piquer's algorithm. After marking export records none.

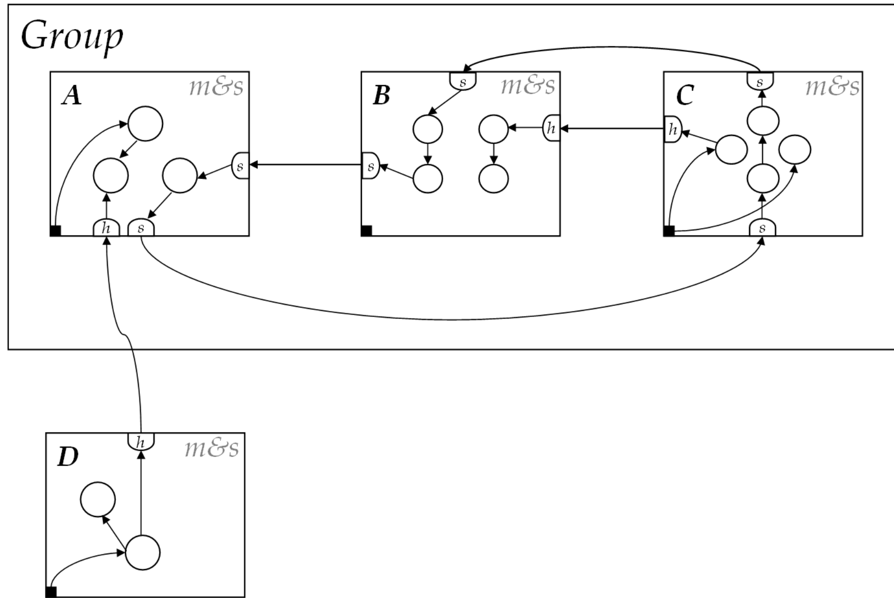

Fig. 19. Piquer's algorithm. Marking export and import records hard if they are reachable by a local root, System C.

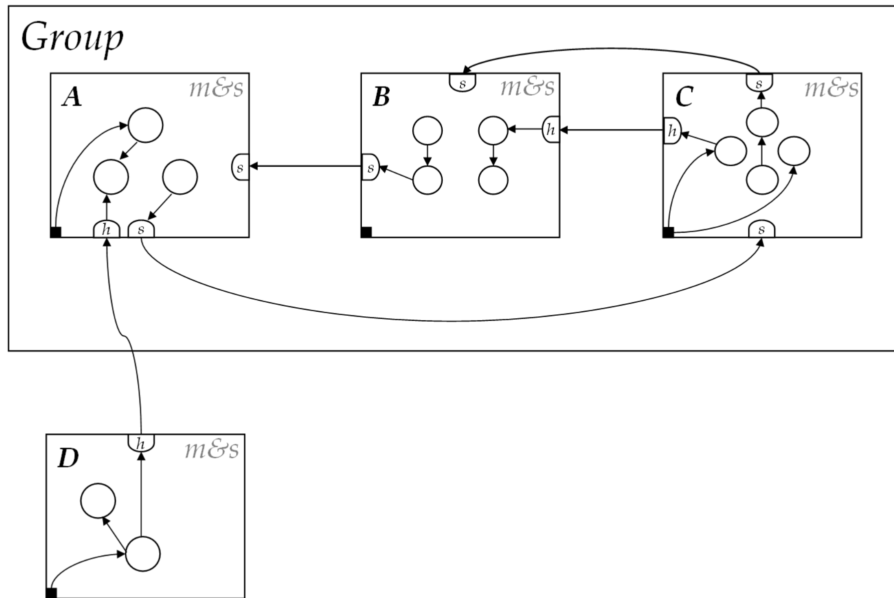

Fig. 20. Piquer's algorithm. Cutting references from import records to their corresponding local objects.

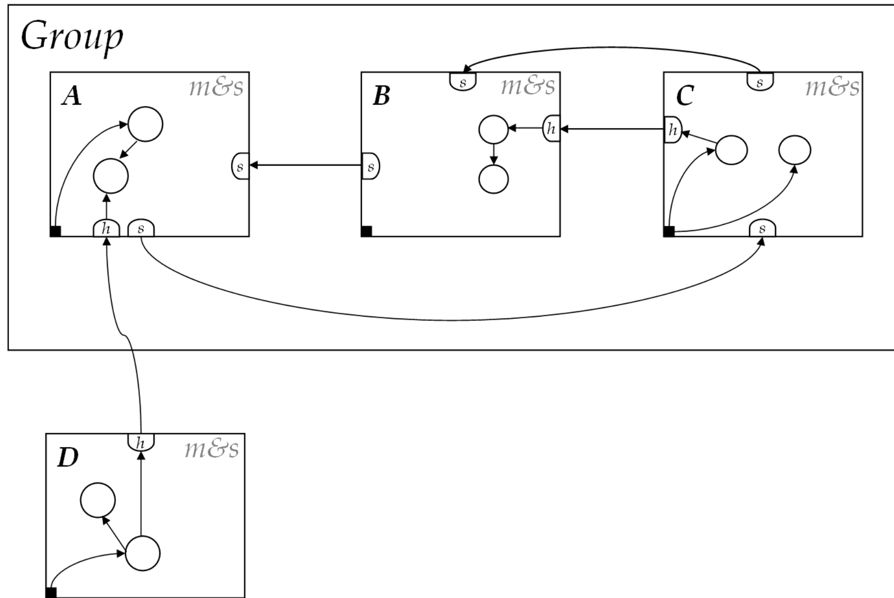

Fig. 21. Piquer's algorithm. After local garbage collection of unreachable and unreferenced local objects. Export records would have been also reclaimed in the last step, directly causing the decrement messages to be sent to the corresponding import record. Using this figure in the figure above and beyond suits illustrative purposes only.

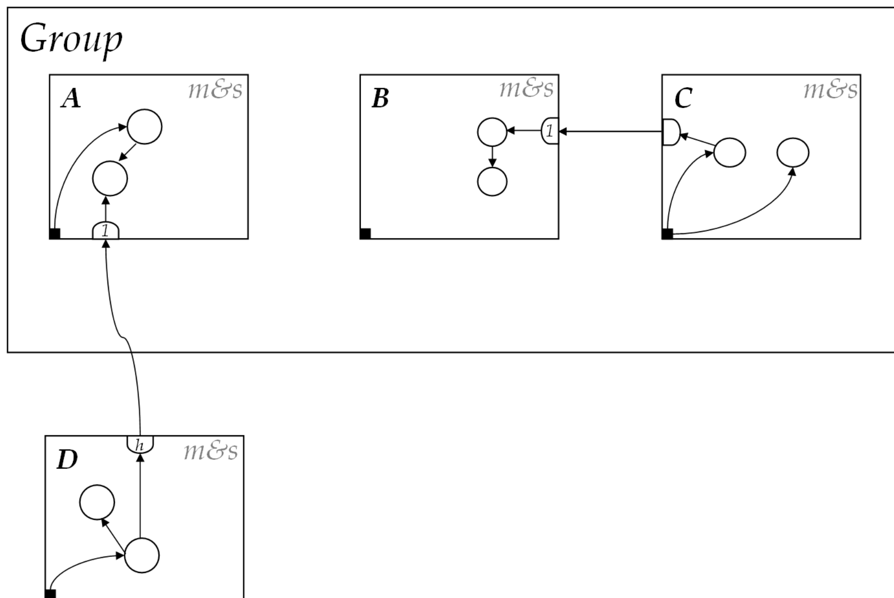

Fig. 22. Piquer's algorithm. After distributed reference counting algorithm eliminated the remaining unreferenced import and export records.

REFERENCES

[1] R. Jones and R. Lins, *Garbage collection: algorithms for automatic dynamic memory management*. New York, NY, USA: John Wiley & Sons, Inc., 1996, ch. 15.

[2] D. Plainfosse and M. Shapiro, "A survey of distributed garbage collection techniques," in *Proc. Int. Workshop on Memory Management*, Kinross, Scotland (UK), Sept. 1995.

[3] S. E. Abdullahi and G. A. Ringwood, "Garbage collecting the internet: a survey of distributed garbage collection," *ACM Comput. Surv.*, vol. 30, no. 3, pp. 330–373, 1998.

[4] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems (3rd ed.): concepts and design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] A. S. Tanenbaum, *Distributed operating systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[6] D. I. Bevan, "Distributed garbage collection using reference counting," in *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*. London, UK: Springer-Verlag, 1987, pp. 176–187.

[7] P. Watson and I. Watson, "An efficient garbage collection scheme for parallel computer architectures," in *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*. London, UK: Springer-Verlag, 1987, pp. 432–443.

[8] M. Rudalics, "Correctness of distributed garbage collection algorithms," Linz, Austria, Tech. Rep. TR 90-40.0, 1990.

[9] R. E. Jones and R. D. Lins, "Cyclic weighted reference counting without delay," in *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*. London, UK: Springer-Verlag, 1993, pp. 712–715.

[10] Y. Ichisuki and A. Yonezawa, "Distributed garbage collection using group reference counting," Position paper for OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems., 1990.

[11] J. M. Piquer, "Indirect reference counting: a distributed garbage collection algorithm," in *PARLE '91: Proceedings on Parallel architectures and languages Europe : volume I: parallel architectures and algorithms*. New York, NY, USA: Springer-Verlag New York, Inc., 1991, pp. 150–165.

[12] S. C. Vestal, "Garbage collection: an exercise in distributed, fault-tolerant programming," Ph.D. dissertation, Seattle, WA, USA, 1987.

[13] P. Hudak and R. M. Keller, "Garbage collection and task deletion in distributed applicative processing systems," in *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*. New York, NY, USA: ACM Press, 1982, pp. 168–178.

[14] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation," *Commun. ACM*, vol. 21, no. 11, pp. 966–975, 1978.

[15] R. J. M. Hughes, "A Distributed Garbage Collection Algorithm," in *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985, pp. 256–272.

[16] M.-A. KA, "Object-oriented storage management and garbage collection in distributed processing systems," Ph.D. dissertation, Stockholm, Sweden, 1984.

[17] B. Liskov and R. Ladin, "Highly available distributed services and fault-tolerant distributed garbage collection," in *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 1986, pp. 29–39.

[18] B. Lang, C. Queinnec, and J. Piquer, "Garbage collecting the world," in *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1992, pp. 39–50.

[19] M. Rudalics, "Distributed copying garbage collection," in *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*. New York, NY, USA: ACM Press, 1986, pp. 364–372.

[20] J. Henry G. Baker, "List processing in real time on a serial computer," *Commun. ACM*, vol. 21, no. 4, pp. 280–294, 1978.