Cooperating distributed garbage collectors for Clusters and beyond

Michael Philippsen
Computer Science Department, University of Karlsruhe
Am Fasanengarten 5, D-76128 Karlsruhe
phlipp@ira.uka.de

Abstract

The contribution of this paper is twofold. First a distributed garbage collector (DGC) is presented that is optimized for remote method invocation in reliable networks, such as current clusters of workstations. Since the algorithm does not require extra acknowledgement messages, even while collecting, it does not increase the latency of a remote call.

Then it is discussed how several DGCs can cooperate in networks that consist of different areas with respect to communication, i.e., of areas with different reliability properties. Proper placement and use of bridge objects allow to select an optimized DGC for every area.

1 Introduction

With the growth of Java, automatic garbage-collection got into the main stream. And since Java offers library support for distributed objects as well, it is only natural that Java includes a distributed garbage collector (DGC).

In the past, DGCs have predominantly been developed for experimental languages or for languages with a small number of users. A lot of research has been done on DGCs but most of the work targets underlying connection networks that suffer from message loss, duplication, delay, out-or-order delivery, disconnected nodes, or network partitioning. To deal with those kinds of challenging network problems, in general complicated acknowledgement schemes have been designed that unfortunately require additional messages on the critical path of a remote call.

Java's RMI (remote method invocation, [12, 14]) uses such a DGC that is suitable for wide-area TCP/IP networks.

However, the use of distributed Java is not restricted to wide-area applications, but there is demand to use Java also for parallel and distributed high-performance applications that require special-purpose high-performance communication networks [4, 13]. For such applications, acknowledgement schemes that slow down a remote method invocation are no longer acceptable. Moreover, RMI programmers tend to pass references to remote objects more frequently than they did in other languages before, because Java's remote method invocation is very transparent and feels much like a regular (local) method invocation. Thus, inefficiencies of an acknowledgement scheme can easily add up. On the other hand, on clusters of workstations with special purpose communication hardware, the network can offer certain guaranteed properties that no longer require the amount of acknowledgement traffic needed by earlier DGCs.

Unfortunately, current RMI does neither allow to use a special purpose DGC nor does it allow to switch to non-TCP/IP networks.

In this paper we review the related work on distributed garbage collectors (section 2) first. In section 3 we then present a new distributed garbage collector that is well suited for both clusters of workstations and fast remote method invocations. With KaRMI [6, 7], we have redesigned and re-implemented JavaSoft's RMI so that both a different distributed garbage collector can be plugged in and non-TCP/IP networks can be used.

There is no quantitative performance analysis in the paper. The absence of acknowledgement messages of any kind on the critical path is a significant qualitative improvement that allows for remote method invocations in only 80 μs on a network of DEC Alpha stations connected by Myrinet (compared to several milliseconds when standard RMI, a standard DGC, and Ethernet are used.)

In section 4 we study how KaRMI makes different DGCs cooperate in heterogeneous networks, for example where a cluster with a high-performance DGC is part of a regular TCP/IP network with a RMI-like DGC.

2 Distributed Garbage Collectors

For this paper we assume a stable communication network where a node cannot be disconnected and where there is no network partitioning. Moreover, no messages are lost, duplicated, or delayed. When two messages follow one another from a single node to the same target node, their relative order remains unchanged by the network. However, no assumption can be made about two messages that are sent from two different nodes to a single recipient. Their relative order of arrival is unknown.

The following discussion of known DGCs ignores network problems and only deals with problems that can occur on such a stable network. A more detailed discussion of DGCs can be found in [9]. Unfortunately, most of the algorithms that can be found in the literature cannot handle migrating objects.

2.1 Reference Counting

All reference counting collectors associate with each remote object – an object that can be accessed remotely from other nodes – a count of the number of remote references to it.

Basic Reference Counting. A race condition can occur in the following situation: When a client (a node that has a remote reference to an object) passes this reference on to another new client, the new client informs the referred object of the fact that a new remote reference to it came into existence, i.e., a so-called increment message is sent. Assume that slightly after this message has been sent, the local collector of the first client determines to claim the remote reference. This will result in a so-called decrement message to be sent to the referred object, but this message originates from a different sender. See figure 2.

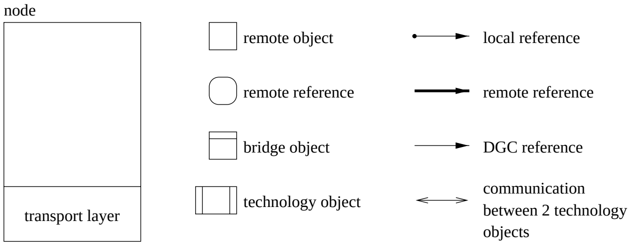

Figure 1. Legend of symbols used in the following figures

Since there is no guaranteed relative order of arrival at the referred object, assume that the decrement message arrives first. The reference counter will be decremented; it might reach zero and thus trigger the local collector to claim the object prematurely, although a remote reference still exists and an increment message is about to arrive.

To avoid that sort of race condition, a decrement message may only be sent after an increment message has been processed, hence every increment message must be acknowledged in reference counting DGCs. The acknowledgement is on the critical path of a method invocation.

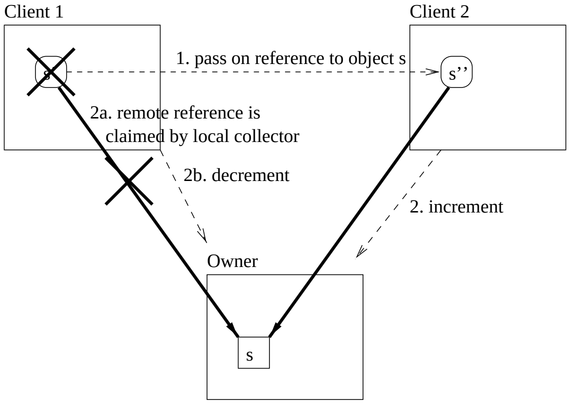
Client 1
Client 2
s'
1. pass on reference to object s
s''
2a. remote reference is claimed by local collector
2b. decrement
2. increment
Owner
s

Figure 2. Race condition of basic reference counting collectors: Client 1 passes on a reference to object s to client 2. Then the local collector of client 1 claims the stub s' and sends a "decrement message" to the owner of s. The new client 2 creates a new stub s'' and sends the "increment message" to the owner. If the decrement message is processed first, the owner's local collector might claim object s although a remote reference (stub s'') still exists.

Weighted Reference Counting. The aim of weighted reference counting [1, 15] is to get rid of the increment message and hence to avoid the acknowledgement message. To achieve that, the object stores a so-called total weight. Each client stores a partial weight; the invariant is that the sum of all partial weights spread over the network is equal to the total weight stored at the owner. When a client forwards a reference to a new client, half the partial weight is passed on as well. See figure 3.

It depends on the implementation what the receiving client does in case it already has a reference to the remote object. It can either add the receiving partial weight to the weight that is already present or it can return the partial weight in an asynchronous response. Often only the exponent k of a partial weight 2^k is stored in weighted reference counting collectors. Then in general, the receiving client must return the partial weight.[1]

When collecting, clients return their partial weight to the owner in decrement messages. The owner deducts the returned weight from the total weight. When the total weight reaches zero, the local collector takes over.

Although weighted reference counting seems to avoid additional messages on the first glance, extra traffic is

[1] Note that in the meantime the first client could have passed a reference to yet another client so that it becomes impossible to accept the returned weight, because it cannot be stored in the 2^k format. In this case, the partial weight is lost and returned to the owner.

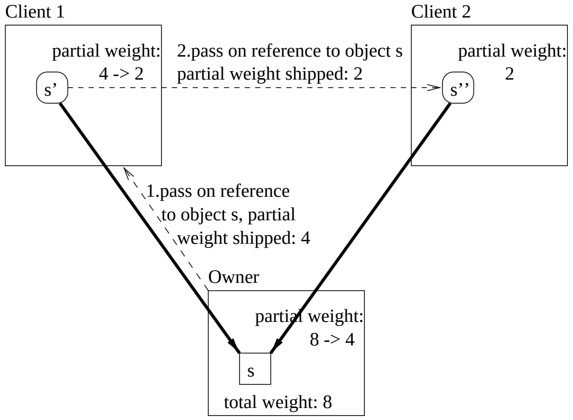

Figure 3. Object s starts with a total weight of 8 and a partial weight of 8. When the first reference is passed on to client 1, half the weight is given to client 1, a partial weight of 4 remains at the owner. Similarly, when client 1 passes the reference to s on to client 2, half the partial weight (2) is given to client 2, the other half remains at client 1. The invariant is that the sum of all partial weight in the network is equal to the total weight stored at the owner.

needed when a client's partial weight reaches a minimal value that cannot be split. Several techniques have been proposed to overcome this problem. The trivial solution is to increase both the total weight at the owner and the partial weight at the client by the same amount. But this requires additional messages on the critical path. Other approaches [11, 3, 8] add some sort of hierarchy where the client gives out new weight without contacting the owner. But the new client has to return the weight to the issuing client instead of the owner. Hence, there is no additional message on the critical path but the collection process itself is nested across the network.

2.2 Reference Lists

Reference lists [2] are similar in spirit to reference counters. Instead of a counter the owner stores a list of items with one entry for every client. The owner not only knows how many clients there are but in addition, the client knows the identity of the clients.

Since DGCs based on reference lists suffer from the same race conditions as reference counting collectors, the same acknowledgements have to be used for every increment message.

Reference lists are preferred over reference counters because they have a better behavior in case of lost or duplicated messages. Moreover, because of the knowledge about client identity, the owner can easily ping the clients to learn about network partitioning or client termination. Bundled with a leasing-based scheme, reference lists are ideal for instable networks.

RMI uses a DGC based on leasing and reference lists.

2.3 Tracing and hybrid DGCs

The general problem with reference counting and reference lists is that cyclic garbage cannot be detected. This can only be achieved by mark-and-sweep algorithms that trace every single reference. Tracing is very costly on slow networks and will not be discussed any further.

Hybrid DGCs employ both a DGC based on reference counting or reference lists and a tracing based DGC to get rid of any cyclic garbage left over. The latter is triggered only in long intervals.

2.4 Comparison

The following table compares the properties of the well-known categories of DGCs. It is obvious that the algorithms either require additional messages on the critical path of a remote method invocation or are costly because of many messages that are sent asynchronously or while collecting. This asynchronous or collection traffic does not slow down individual remote method invocations directly, but if a lot of such traffic is generated, the network's bandwidth may become a critical resource that prevents remote method invocations from being processed at full speed.

	basic ref. count	weight ref. count	ref. list	trace
add. msg. on critical path	yes	—	yes	—
add. concurrent msg.	—	yes	yes	—
add. msg. while collecting	—	yes	—	many
memory consumption	small	small	medium	small
amount of computation	small	small	small	medium

3 Max Counter DGC

We now present our max counter DGC that is specifically optimized for clusters with stable networks. It avoids all additional messages. Moreover, it works correctly even for migrating objects. Unfortunately, this DGC uses a lot of memory: each stub needs an array of integers that has a slot per node in the network. Compared to JavaSoft's RMI stubs, for clusters of 32 or more nodes, the extra array needs more memory than the stub itself. Moreover, the max counter DGC need longer messages than other DGCs – but this is acceptable on networks where latency and not bandwidth is the bottleneck.

3.1 General Idea

The max counter DGC is inspired by ideas of reference counting, weights, and reference lists but stores its information in a decentralized way. The general idea of the max counter DGC is that wherever an operation is executed that is relevant for the collection, that information is kept in as much detail as possible. Every node monitors how often a reference to a given object is passed on to other nodes (max counter array). Similarly, every node accepting a reference knows how often that reference has been passed to it (sum counter). When references are sent over the network, they are accompanied with the max counter array and the sum counter so that nodes can learn from their peers' states. The recipient knows how to combine the local counters with the ones in the message. Upon local collection, a node simply sends all its information to the owner who knows how to combine this array with its own and how to determine whether a local collector can be triggered.

More formally, every node A stores an information tuple $(m_A[1..n], s_A)$. The index of A in the max counter array m_A tells us how often node A has passed on a reference to other nodes. The sum counter s_A on the other hand tells us how often node A node has received a reference from other nodes. Two tuples are combined as follows:

$$(m_A, s_A) \oplus (m_B, s_B) = \\ ([\max(m_A[1], m_B[1]), \ldots, \max(m_A[n], m_B[n])], s_A + s_B)$$

Since $\oplus$ is associative and commutative, we can define $\sum \Gamma$ to be the application of $\oplus$ to all tuples of nodes in a set Γ. Furthermore, $m \sum \Gamma$ denotes the max counter array of the total, $s \sum \Gamma$ is the sum counter of the total.

The invariant of the max counter DGC algorithm is:

$$s \sum \Gamma = \sum_{i=1}^{n} (m \sum \Gamma)[i]$$

The total sum of references passed out to other nodes is identical to the number of references received by all the nodes. The object can be claimed by the local collector of the owner O if there is no local reference to it and if

$$s_O = \sum_{i=1}^{n} m_O[i]$$

During its lifetime, the owner learns about references passed on to other nodes (by every other node). This information is collected in its max counter array. The owner's local collector can claim the object if the total number of these references, i.e., the sum of the elements of the max counter array, is equal to the number of references returned to the owner. As long as there are remote references to the object, s_O will be smaller than the sum of the max counter elements. (At this point it becomes obvious that "decrement messages" must use the same $\oplus$-mechanism.)

3.2 Operations

Each of the following operations of the max counter DGC is an atomic unit, properly kept in isolation by the implementation. In the previous section, only nodes had information tuples. But messages need to have information tuples as well. The invariant covers both nodes and messages.

1. Reference Creation at node A. Node A is about to send a reference to an arbitrary recipient. The node's max counter m is incremented at position A, since a reference is issued by that node. The sum counter remains unchanged (because node A does not receive a new reference.)

Then a new message is created. The node's max counter is copied into the message to inform the recipient of all the reference passing node A is aware of. The message's sum counter is set to 1, since the message has received a reference exactly once.

2. Receive Reference at node B where B is already client. Node B receives a message with a reference to a remote object. Node B already has a stub that refers to the remote object, and B stores the necessary information tuple.

Node B applies the $\oplus$-operation to merge the information tuple of the message to its own information tuple. Since the sum counter of the message is 1 (see operation 1 above) B increments its sum counter by 1 (i.e., B got a new reference.) The max-operation ensures that B has the most accurate and combined information available on how often references have been passed on by all other nodes.

Finally, the message is discarded.

3. Receive Reference at node B where B is not yet client. Node B creates a stub that refers to the remote object according to the addressing information in the message. The new information tuple is initialized to ([0..0],0), then the information tuple of the message is merged by means of the $\oplus$-operation.[2] The message is then discarded.

4. A stub at node A is claimed by the local collector. The complete information tuple is sent to the last known owner of the object.

5. Claim information arrives at node B. When a message about a claimed stub arrives at node B, node B can or cannot be the actual owner of the referred remote object since the object might have migrated.

[2] Actually, the information tuple is not initialized to all zeros and hence the $\oplus$-operation is reasonable. Instead there is an additional global integer per global object ID not mentioned before. This counter that is kept even if the corresponding stub is claimed by a local collector contributes to the initialization of B's information tuple. We skip a detailed discussion of this counter for brevity.

However, as long as there still is a stub at B, the information tuple of the message is simply merged to the information tuple stored at B. The message is then discarded.

Only if B itself no longer has a stub, the message is forwarded to the assumed new owner.

6. Node A sends a migration message. Node A is the current owner of the object. No other node can send a migration message. The node's max counter m is incremented at position A, since a reference is issued by that node. Its sum counter is incremented by 1.

Then a new message is created. The node's max counter is copied into the message. The message's sum counter is set to 0.

Note that the main difference to operation 1 is the way sum counters are handled. In contrast to operation 1, the migration message itself appears to be the remote object ($s = 0$). Node A seems to have received a reference to it by having its counter incremented.

In contrast to all other operations, sending of migration messages needs an acknowledgement. Only after the recipient has properly installed the object the local collector may claim the stub at node A.

7. Migration message arrives at node B where B is already client. Node B becomes the new owner of the object. The information tuple of the message is merged to the tuple that is stored locally. Then the message is discarded and an acknowledgement is sent to the originator of the migration message.

8. Migration message arrives at node B where B is not yet client. Node B becomes the new owner of the object. The new information tuple is initialized to ([0..0],0), then the information tuple of the message is merged by means of the $\oplus$-operation. The message is then discarded and an acknowledgement is sent to the originator of the migration message.

4 Examples

The following examples are based on a shorthand notation. The **create** operation indicates that an object (not identified here) is created. When a reference is sent to node **y** (operation 1), we write **ref.s(y,idx)**. When the corresponding message is received from node **x** (operations 2 or 3), we write **ref.r(x,idx)**. The index **idx** is identical for a send-receive-pair and unique otherwise.

Similarly, **del.s(y,idx)** and **del.r(x,idx)** represent the sending and receiving of messages caused by local collection (operations 4 and 5). Finally, **mig.s(y,idx)** and **mig.r(x,idx)** stand for the sending and receiving of migration messages.

4.1 Basic Example

Let us first consider a network of three nodes. In the first line of table 1 (t=1), an object is created. The active node is node 1 (see column marked A). The corresponding information tuple is initialized to ([0,0,0],0). Since there is no remote reference to this object, the sum of the elements of the max counter array is equal to the sum counter. In the table, this fact is indicated by "=".

A reference to this object is then passed on to the second node (lines 2 & 8) which in turn passes on the reference to the third node (lines 4 & 5). This scenario is similar to the one depicted in figures 2 and 3. Afterwards both clients have their stub collected by the local collector (lines 6–9).

Table 1 shows all the modifications of information tuples. If a node does not have an information tuple, this fact is indicated by means of "—". Empty cells ("·") indicate that the information tuple has not changed from the previous line.

In line 2, the reference is sent out to node 2. Hence, the information tuple of node 1 is changed and a new message is created. The information tuple of the message is shown in the last column of the table. After creation of the message, the local collector of node 1 can no longer claim the object – formally the sum of the elements of the max counter array is higher than the sum counter (>).

In line 3, the message is consumed by node 2. Accordingly, node 2 installs an information tuple of its own. Similar for lines 4 and 5.

The subsequent delete operations are triggered when the local collectors on nodes 2 and 3 remove the stubs. Then the information tuple is sent to the owner (node 1). The first claim message that arrives at the owner informs the owner that both node 1 and 2 have sent out references. Hence, two claim messages with sum=1 are needed to release the object. From the sum counter in node 1 it can be seen that both existing references must cease to exist before a "=" is reached again.

The invariant holds for every line in the table. To check that, combine all information tuples of a line by means of the $\oplus$-operation. The sum of the elements of the total max counter is always equal to the total sum counter.

Moreover, it is easy to see that the race condition that is typical for reference counting DGCs does not occur.

4.2 Advanced Example

In this example, every operation discussed in section 3.2 actually occurs. Table 2 gives all the information tuples.

First, an object is created on node 1. References to this object are then sent to nodes 2 and 3. In line 5, this reference is returned to node 1: One can see what happens if a reference is sent to a node that already is a client (in

Table 1. Information tuples for Basic Example

t	A	op	node				message
			1	test	2	3	tuple
1	1	create(0)	([0,0,0],0)	=	—	—	—
2	1	ref.s(2,1)	([1,0,0],0)	>	—	—	([1,0,0],1)
3	2	ref.r(1,1)	·	>	([1,0,0],1)	—	—
4	2	ref.s(3,2)	·	>	([1,1,0],1)	—	([1,1,0],1)
5	3	ref.r(2,2)	·	>	·	([1,1,0],1)	—
6	2	del.s(1,3)	·	>	—	·	([1,1,0],1)
7	1	del.r(2,3)	([1,1,0],1)	>	—	·	—
8	3	del.s(1,4)	·	>	—	—	([1,1,0],1)
9	1	del.r(3,4)	([1,1,0],2)	=	—	—	—

Table 2. Information tuples for Advanced Example

t	A	op	node					message
			1	2	3	4	5	tuple(s)
1	1	create(0)	([0,0,0,0,0],0)	—	—	—	—	—
2	1	ref.s(2,1)	([1,0,0,0,0],0)	—	—	—	—	([1,0,0,0,0],1)
3	1	ref.s(3,2)	([2,0,0,0,0],0)	—	—	—	—	([2,0,0,0,0],1), ([1,0,0,0,0],1)
4	2	ref.r(1,1)	·	([1,0,0,0,0],1)	—	—	—	([2,0,0,0,0],1)
5	3	ref.r(1,2)	·	·	([2,0,0,0,0],1)	—	—	—
6	3	ref.s(1,3)	·	·	[(2,0,1,0,0),1]	—	—	([2,0,1,0,0],1)
7	1	ref.r(3,3)	[(2,0,1,0,0],1)	·	·	—	—	—
8	1	ref.s(4,4)	[(3,0,1,0,0],1)	·	·	—	—	([3,0,1,0,0],1)
9	3	ref.s(4,5)	·	·	([2,0,2,0,0],1)	—	—	([2,0,2,0,0],1), ([3,0,1,0,0],1)
10	4	ref.r(3,5)	·	·	·	([0,0,2,0,0],1)	—	([3,0,1,0,0],1)
11	4	ref.r(1,4)	·	·	·	([3,0,2,0,0],2)	—	—
12	3	del.s(1,6)	·	·	—	·	—	([2,0,2,0,0],1)
13	1	del.r(3,6)	([3,0,2,0,0],2)	·	—	·	—	—
14	4	ref.s(3,7)	·	·	—	([3,0,2,1,0],2)	—	([3,0,2,1,0],1)
15	4	ref.s(2,8)	·	·	—	([3,0,2,2,0],2)	—	([3,0,2,2,0],1), ([3,0,2,1,0],1)
16	2	ref.r(4,8)	·	([1,0,0,2,0],2)	—	·	—	([3,0,2,1,0],1)
17	3	ref.r(4,7)	·	·	([0,0,2,1,0],1)	·	—	—
18	1	mig.s(5,9)	([4,0,2,0,0],3)	·	·	·	—	([4,0,2,0,0],0)
19	5	mig.r(1,9)	·	·	·	·	([4,0,2,0,0],0)	—
20	3	del.s(1,10)	·	·	—	·	·	([0,0,2,1,0],1)
21	1	del.r(3,10)	([4,0,2,1,0],4)	·	—	·	·	—
22	4	ref.s(1,11)	·	·	—	([3,0,2,3,0],2)	·	([3,0,2,3,0],1)
23	1	ref.r(4,11)	([4,0,2,3,0],5)	·	—	·	·	—
24	1	del.s(5,12)	—	·	—	·	·	([4,0,2,3,0),5)
25	4	del.s(5,13)	—	·	—	—	·	([3,0,2,3,0],2), ([4,0,2,3,0],5)
26	5	del.r(4,13)	—	·	—	—	([4,0,2,3,0],2)	([4,0,2,3,0],5)
27	5	mig.s(4,14)	—	·	—	—	([4,0,2,3,1],3)	([4,0,2,3,1],1), ([4,0,2,3,0],5)
28	5	del.r(1,12)	—	·	—	—	([4,0,2,3,1],8)	([4,0,2,3,1],0)
29	4	mig.r(5,14)	—	·	—	([4,0,2,3,1],0)	·	—
30	5	del.s(4,15)	—	·	—	·	—	([4,0,2,3,1],8)
31	4	del.r(5,15)	—	·	—	([4,0,2,3,1],8)	—	—
32	2	del.s(4,16)	—	—	—	·	—	([1,0,0,2,0],2)
33	4	del.r(2,16)	—	—	—	([4,0,2,3,1],10)	—	—

this case the owner itself). From lines 8 to 11, two nodes send references to node 4. Node 3 deletes its stub in line 12 but later receives a new reference from node 4 in line 17. (Please note the max counter vector of line 17 takes into account the global counter of node 3 that is not discussed in this paper.)

In lines 18 & 19 the owner migrates from node 1 to node 5. It is shown in line 21 how messages are handled that arrive after the owner has gone but when a stub still exists. The stub is finally removed in line 24. Starting in line 24 all stubs are claimed by local collectors. This process is mixed with another migration of the object in lines 27 & 29. Finally, all stubs are gone and node 4 is the owner.

Again, the last column holds the information tuple(s) of messages that are out in the network. The invariant holds in every line of the table.

Only in lines 1 and 33, the sum of the elements of the max counter array is equal to the sum counter. Hence, only in lines 1 and 33, a local collector is allowed to claim the object.

4.3 Proof of correctness

The proof of correctness is very technical with lots of cases. Since it does not give any new insights, we simply refer the interested reader to [10] for the details.

The max counter DGC has been implemented in the JavaParty environment [5, 6]. The implementation relies on Java's weak object references. All of the operations mentioned in section 3.2 are properly synchronized.

5 Bridging several DGCs

Todays hardware infrastructure does not consist of isolated clusters of workstations alone. Instead, those clusters are often part of TCP/IP networks, or can even be accessed from the web.

Each node can have different transport mechanisms available, one for each type of communication network the node belongs to. A typical scenario might consist of nodes that are at the interior of a cluster and can only be accessed through the cluster's high-performance network, i.e., they only use a single transport technology while other nodes link the cluster into a bigger TCP/IP-network. Those need to be able to communicate by means of two types of technology, for example through Myrinet boards and through Ethernet cards. External nodes can only use Ethernet.

Obviously, in different areas, such a network has different properties so that different DGCs are best suited.

This section presents a mechanism that allows to use specifically tuned DGCs in different areas of such a network and make them cooperate correctly.

We first discuss bridge objects for the purpose of communication without taking garbage collection into account. A rule is derived that decides whether and where to create a bridge object. Then (see section 5.2) we refine this rule to ensure correct behavior if several DGCs are used concurrently.

5.1 Bridge objects for communication purposes

If the sender and the corresponding receiver of a message do not have access to the same communication technology, a so-called bridge object that can use both technologies needs to be created transparently at one of the nodes on the path. See figure 4.

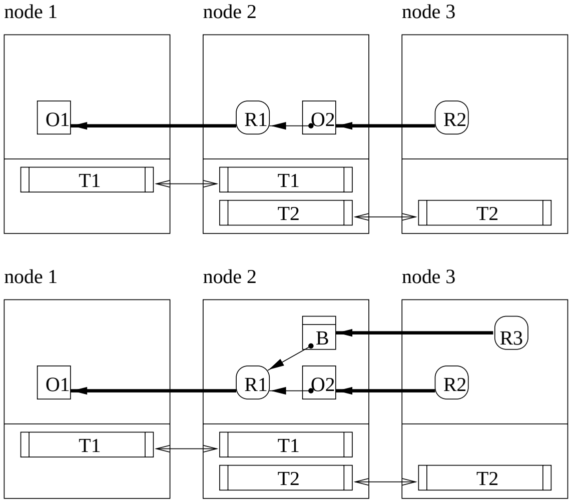

Figure 4. In the upper situation node 3 can only communicate with node 2 by means of technology T2. When a remote method invocation on node 3 returns a reference to object O1, a new reference R3 is created. However, R3 cannot refer to O1 directly because of the lacking technology object T1. Therefore, a communication bridge B is created in node 2 that forwards any access from R3 through R1 to O1. See figure 1 for the legend.

It seems sufficient for the sender to know how to access the bridge, since the bridge knows how to forward the message to the final target. However, it is more efficient if the sender stores both the address of the bridge and the address of the original target object. The reason becomes apparent when the sender passes on the reference to other nodes that might have a communication technology available to access the target directly.

For example, consider figure 4 again. Assume that node 3 passes on its reference to a new node 4 that has both technologies T1 and T2 available. If the reference R3 would only store the address of the bridge, every message from the new node to O1 would need to go through the slow bridge on node 2.

Only if R3 stores a direct address as well, the access path can be short-cut as soon as a common communication technology becomes available. Short-cutting that avoids previously created bridges is essential for good performance.

5.2 Bridge objects for DGC purposes

The obvious rule that ensures correct cooperation of several DGCs is that every object may only be claimed by a local collector if none of the DGCs signals the existence of a remote reference to that object.

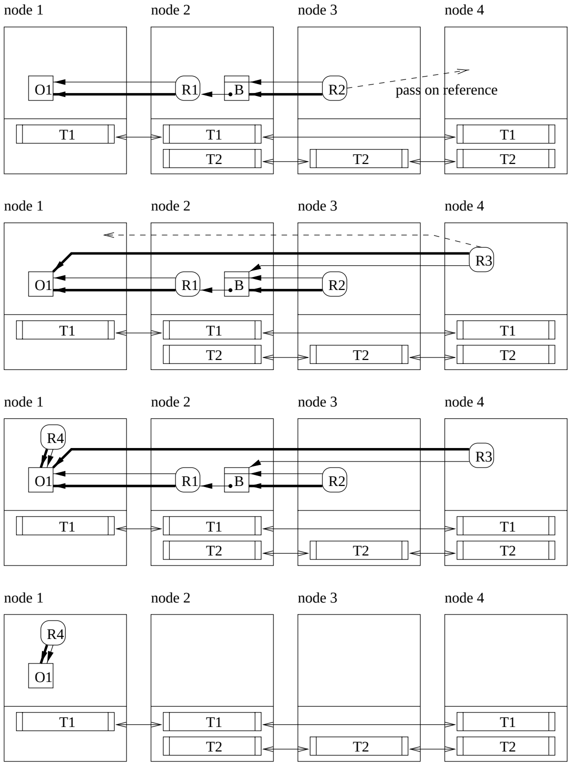

Figure 5. At first, node 3 has a reference to object O1 through bridge B. Node 3 passes on the reference to node 4. Since node 4 has technology T1 available the new reference R3 can reach O1 directly, but uses the bridge for collecting purposes (2nd row). Both addresses (O1 and B) are stored in R3. Node 4 then passes on the reference to node 1. The new reference R4 can use O1 directly for communication and collection. Since nodes 1 and 4 share technology T1, no new bridge has been built. When local collectors claim R1, R2, and R3, only R4 remains. Unfortunately, R4 still stores an address of B that no longer exists.

When bridge objects are added for communication purposes, there are two general options for the DGC. If only one DGC is used for the whole network, the bridge object can just be used to forward any messages to the target, i.e., bridge objects are more or less ignored by the DGC.

But since we would like to use different DGCs for different areas of the network, the DGCs need to cooperate at the boundaries of the technologies. Bridge objects seem ideal to implement that sort of cooperation, since they can be used as interfaces between different DGCs. The DGC of one technology area treats the bridge object as a new owner object to which remote references exist. To the other side and the other technology's DGC, the bridge object acts as a regular remote reference to the original owner object.

Unfortunately, short-cutting of communication paths is not a blessing for cooperating DGCs, as shown in figure 5. Short-cutting requires that both the address of the bridge and the direct address need to be stored in a reference. But this is insufficient from the DGC point of view. In addition, bridges must not be claimed by local collectors as long as references exist that store their addresses.

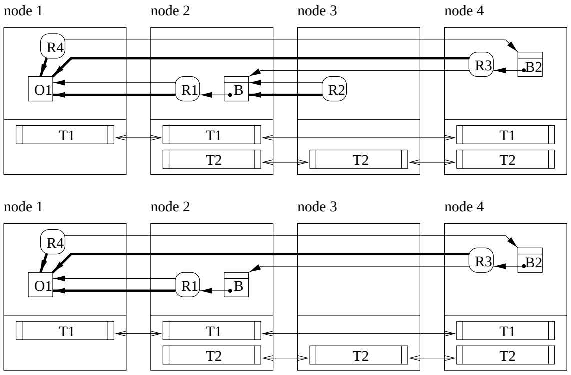

Figure 6. This replaces the last two rows of figure 5. R4 uses the new bridge B2 for collecting purposes. The local collectors can only claim R2. The remaining objects R1, B, and R3 will be kept as long as R4 might need them.

In the example, R4's DGC-reference needs to reference B, at least indirectly. Therefore, node 4 has to create a new bridge B2 when passing on a reference to node 1. This is shown in figure 6.

Thus, a new bridge is not only needed when there is no common communication technology for sender and receiver, but also when the latest bridge (on the path to the owner) is accessed by means of a technology that is different from the technology needed for direct access.

6 Conclusion

The max counter DGC presented in section 3 has been designed for reliable networks. It does not need any extra acknowledgement messages, even during collection. By storing the management information in a decentralized way, every node can pass on references to remote objects without consulting other nodes first. Hence, the max counter DGC does not add to the latency of a remote method invocation (at least if the network provides enough bandwidth for the enlarged message sizes.)

To use the max counter DGC in situations where a highly reliable area of the network is located within regular wide-area networks, we have shown how several DGCs can be made to cooperate. A rule has been derived for proper placement of bridge objects that allows short-cutting of communication paths and avoids the erroneous collection of previously created bridges.

Acknowledgements

I would like to thank all members of the JavaParty team. Special thanks to Christian Nester and Bernhard Haumacher for finding bugs in earlier approaches to our DGC. Jochen Reber did a great job in spelling out the proof that finally convinced us.

References

[1] D. I. Bevan. Distributed garbage collection using reference counting. In *Parallel Architectures and Languages Europe*, number 258 in Lecture Notes in Computer Science, pages 117–187, Eindhoven, The Netherlands, June 1987. Springer-Verlag Berlin, Heidelberg, New York.

[2] A. Birrel, D. Evers, G. Nelson, S. Owicki, and T. Wobber. Distributed garbage collection for network objects. Technical Report 116, Digital Equipment Corporation, Systems Research Center, December 1993.

[3] H. Corporaal, T. Veldman, and A. J. van de Goor. An efficient, reference weight based garbage collection method for distributed systems. In *Proc. of the PARBASE-90 Conf.*, pages 463–465, 1990.

[4] M. Jacob, M. Philippsen, and M. Karrenbach. Large-scale parallel geophysical algorithms in Java: A feasibility study. *Concurrency: Practice and Experience*, 10(11–13):1143–1154, September–November 1998.

[5] JavaParty. http://wwwipd.ira.uka.de/JavaParty/.

[6] C. Nester. Ein flexibles RMI Design für eine effiziente Cluster Computing Implementierung. Master's thesis, University of Karlsruhe, Department of Informatics, May 1999.

[7] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI. In *ACM 1999 Java Grande Conference*, pages 152–159, San Francisco, 1999.

[8] J. M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *Parallel Architectures and Languages Europe*, number 365 in Lecture Notes in Computer Science, pages 150–165, Eindhoven, The Netherlands, June 1991. Springer-Verlag Berlin, Heidelberg, New York.

[9] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

[10] J. Reber. Verteilte Speicherbereinigung für JavaParty. Studienarbeit, University of Karlsruhe, Department of Informatics, August 1999.

[11] M. Rudalics. Correctness of distributed garbage collection algorithms. Technical Report 90-40.0, Johannes Kepler Universität, Linz, Austria, 1990.

[12] Sun Microsystems Inc., Mountain View, CA. *Java Remote Method Invocation Specification*, October 1998. ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.

[13] G. K. Thiruvathukal, F. Breg, R. Boisvert, J. Darcy, G. C. Fox, D. Gannon, S. Hassanzadeh, J. Moreira, M. Philippsen, R. Pozo, and M. Snir (editors). Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98: International Conference on High Performance Computing and Communications*, Orlando, Florida, November 7–13, 1998. panel handout.

[14] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July–September 1998.

[15] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architecture. In *Parallel Architectures and Languages Europe*, number 258 in Lecture Notes in Computer Science, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer-Verlag Berlin, Heidelberg, New York.