Garbage Collecting the Internet: A Survey of Distributed Garbage Collection

SALEH E. ABDULLAHI AND GRAEM A. RINGWOOD

Queen Mary and Westfield College, University of London

Internet programming languages such as Java present new challenges to garbage-collection design. The spectrum of garbage-collection schema for linked structures distributed over a network are reviewed here. Distributed garbage collectors are classified first because they evolved from single-address-space collectors. This taxonomy is used as a framework to explore distribution issues: locality of action, communication overhead and indeterministic communication latency.

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming, Distributed Programming, Parallel Programming; D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**File Systems Management**]

General Terms: Languages, Management, Performance, Reliability

Additional Key Words and Phrases: Automatic storage reclamation, distributed object-oriented management, distributed file systems, distributed memories, memory management, network communication, distributed, object-oriented databases, reference counting

1. INTRODUCTION

Efficient automatic garbage collection is so useful and so difficult to make unobtrusive that it has been a field of active research for over three decades. The problem was considered "solved" in the late '80's with state-of-the-art generation scavengers employed by Smalltalk systems. Smalltalk was the major driving force in the development of garbage collectors in the 1980s. The 1990s saw a significant technology shift to distributed systems, which provide a further challenge to language and garbage-collection design.

Unfortunately, the term *distributed system* has been applied to so wide a range of computing systems—loosely coupled, closely coupled, tightly coupled, array processors, dataflow, neural nets, etc. [Booth 1981]—as to become almost without meaning. For this review, following Lamport [1978], a system is classified as distributed if the end-to-end message transmission delay is significantly greater than the time between consecutive events of a process.

Storage reclamation became a necessity when LISP pairs were introduced in the early 1960s [McCarthy 1981]. The lifetimes of such structures generally exceed the lifetime of modules that access them. Indeed, for Smalltalk the data structures are persistent. Moss [1990] defines a *persistent store* as one that outlives the execution of any process. The definition includes file systems and databases. Atkinson et al.

CONTENTS

[1983] insist that automatic garbage collection is a vital component of persistence.

Many factors contribute to the increasing use of distributed systems, but the most compelling is economics. Distributed file servers are commonplace and compute servers are now appearing. The problems of computer networking, heterogeneity, latency, and scale, are usually addressed by a hierarchy of communication protocols. The term *protocol* refers to a set of precisely defined conventions for communication between corresponding layers of the hierarchy on different sites. The definition of the layers and their protocols has been an area of intensive research and standardization for two decades. One particular seven-layer model has been adopted as a reference model for network software by the International Standards Organization, ISO. The model, the *Reference Model for Open Systems Interconnection*, is more commonly known as OSI. Unfortunately, the services offered by different OSI layers are duplicated and confused. The core concepts of reliability, flow control, and security can be addressed at all layers [Peterson and Davie 1996]. By contrast, these concepts are addressed in a single specific layer of the Internet protocols. This and the fact that code was bundled with BSD Unix and made essentially free to universities has made the Internet the *de facto* standard.

Prior to Lisp, Fortran was deliberately designed so that the size of the storage required for the program was known at compile time. The distributed counterpart of Fortran is Occam. For such languages, memory management is handled entirely by the compiler and no runtime support is necessary. These languages impose considerable restrictions on modularity and ease of expression. For example, recursive procedure calls and dynamic data structures are ignored. Programming languages such as Lisp, Prolog, and Smalltalk transparently offer the programmer dynamic data structures. Giving up the enormous productivity advantages of such languages is a high price to pay for distribution.

A new language that offers transparent dynamic data structures, Java, is enjoying unprecedented popularity. What is different about Java is that it is especially suited for programming Internet applications. The developers of Java, Sun Microsystems, conceived the language for programming consumer electronics, such as toasters and microwaves. This application favored an interpreted implementation like Lisp, Prolog, and Smalltalk. While Java was under development, the World Wide Web took the Internet by storm. The design decisions made for Java made it ideal for Internet applications. As a proof of concept, a web browser, HotJava included support for embedded Java applets in HTML. Like other interpreted languages such as Lisp, Prolog, and Smalltalk, Java provides automatic garbage collection. In Sun's early releases of Java, garbage collection was slow. Later releases are beginning to provide distributed garbage collection [Sun 1996].

This paper provides a review of distributed garbage-collection schemes that can be applied to autonomous systems connected by a network. Accepting the Internet as the *de facto* standard

protocol, the title of this paper paraphrases the title of Lang et al. [1992]. Empirical evidence [Abdullahi 1995] suggests this collector is the state of the art. Knuth [1973] reviewed (nondistributed) collectors that appeared before 1968, and Cohen [1981] brought the survey of papers up to 1981. The most recent review of (nondistributed) garbage collection is Wilson [1992] presented at the *International Workshop on Memory Management*. A preliminary version of the present review [Abdullahi et al. 1992] appeared in this same workshop. A second review of distributed garbage collectors [Plainfossé and Shapiro 1995] was presented at a subsequent International Workshop of Memory Management. Jones and Lins [1996] published the first textbook on garbage collection, but it is mainly concerned with nondistributed garbage collection; Jones [1996] maintains a webliography on garbage collection.

Research on distributed garbage collection can be better understood as trial-and-error adaptation of ideas developed for single-processor, single-address space collectors. For this reason, Section 2 surveys and classifies nondistributed garbage collectors. "Old hands" are warned that some attempt is made to rationalize the ontology of the subject. Section headings reflect the rationalized rather than historical nomenclature. Using this classification, Section 3 gradually uncovers the issues of distribution. Section 4 summarizes the problems and proposed solutions.

2. TAXONOMY OF SINGLE-ADDRESS-SPACE GARBAGE COLLECTORS

2.1 The Single-Address-Space Garbage-Collection Problem

Garbage collection, GC, is an inevitable consequence of programming languages that employ dynamic data structures. With dynamic data structures, the state of a computation at any instant can be considered as a many-rooted, directed graph called *the computation graph* (Figure 1). The roots are distinguished vertices that provide entry points to the graph. The internal vertices of the computation graph are realized as *cells*, contiguous segments of memory. A cell is effectively a base address from which offsets may be legitimately accessed. Due to concern for garbage collection in object-oriented languages, such as Smalltalk, cells (as everything in object-oriented languages) are called *objects*. If the cells are not of fixed size, they commonly have a terminator or a header field containing some indicator of the size of the cell. The indicator may be the number of bytes in the cell or a pointer to the last byte of the cell. Runtime-typed languages such as Prolog and Smalltalk often have a header field describing the data type of the cell. Edges of the graph are realized by store address fields within cells.

Cells referenced directly or indirectly from a root are said to be *reachable*, *accessible*, or *live*. As a computation progresses, addition and deletion of roots, vertices, and edges modify the graph. As a result, some portions of the graph become disconnected. Such cells are said to be *unreachable*, *inaccessible*, or *dead*. These disconnected subgraphs make no contribution to the result of the computation and are known as *garbage*. In Figure 1, a garbage cell is denoted by a filled circle and an accessible cell by an unfilled circle. Without reuse, the finite store available for allocating new vertices diminishes to zero. *Garbage collection* is the process by which the store occupied by garbage can be reused.

Such graph structures also occur in caches, file systems, and databases. The storage controlled by a file system appears as a directed graph of blocks. Some blocks are indices that contain references to other blocks. A file system maintains a file as long as its blocks can be traced from the root block of the disk. The idea of collecting processes was first addressed in the context of the Actor language [Agha 1986; Kafura et al. 1990]. Each process, or *actor*, is ref-

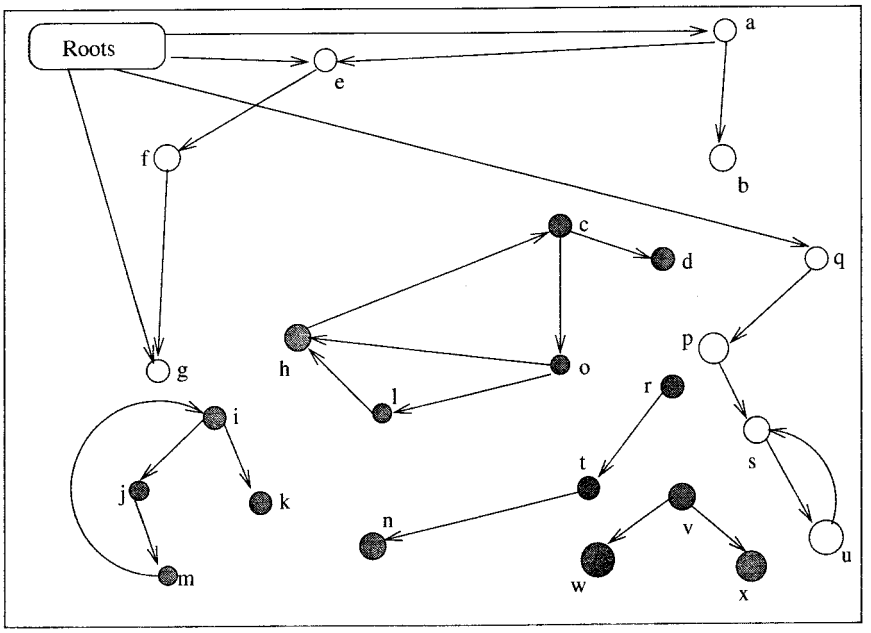

Figure 1. A representative, though small, state of a computation.

erenced by its mailbox. Root actors are those that interface directly with devices. One actor can communicate with another if its mailbox is known. An actor is garbage if it cannot be communicated with, indirectly, from a root actor.

Most imperative languages (e.g., Pascal, C, C++, etc.) place the responsibility for cell allocation and reclamation on the programmer. With such languages, the programmer must explicitly reclaim garbage cells by using *dispose*, *delete*, or *free* system calls. Explicit store management leads to two very common bugs: *incompleteness* (also called memory leaks), a failure to reclaim all extant garbage, and *unsoundness* (also called the dangling reference problem), the premature reclaiming of accessible cells. Memory leaks can gradually accumulate until the computation fails because no space is left to allocate a new cell needed to complete the computation or store a file. Even if the computation does not run out of space, a computation may make little progress because of thrashing. Following Denning [1968], a computation with page faults every few instructions is said to thrash. Memory leaks can cause extreme nonlocality of reference as live cells are dispersed over a large virtual address space.

Reclaiming cells prematurely by a careless use of delete or free can cause the other even more insidious bug, unsoundness. When memory is reclaimed too soon, the space occupied by a cell may be reused while the cell is still reachable from a root. The same store location may, therefore, be simultaneously interpreted in different ways. Updates to one interpretation will cause unpredictable effects on the other. Unsoundness can be difficult to detect and debug because computations tend to fail long after the event that caused the problem.

The ontology, soundness, and completeness derive from theorem proving. What is different here is that the transitive closure of the references is dynamic. Because the two problems are difficult for programmers to address, a wide variety of languages provide automatic allocation and reclamation as part of their runtime system. In such

systems, the allocation routines (e.g., CONS in Lisp) perform special actions to reclaim space as necessary, often when a memory request cannot be satisfied from the available free store. Calls to the *deallocator* (free or dispose) become unnecessary, as they are implicit in calls to the allocator. Dijkstra et al. [1978] introduce two useful abstractions to the study of garbage collection. The *mutator*, M, abstracts the process that performs the computation, including allocation of a new cell. The process that automatically reclaims garbage is called the *collector*, C. Three kinds of mutator operations that affect the garbage collector can be distinguished:

- creation of an edge to a new vertex
- creation of a new edge between existing vertices, and
- destruction of an edge

Historically, the major drawback of automatic garbage collection was that it significantly detracted from the performance of the mutator, both by introducing unpredictably long pauses in computation and by using large proportions of available processing cycles. Measurements of early Smalltalk-80 implementations [Krasner 1983] indicate 20% to 70% of runtime spent garbage collecting. Steele [1975] and Wadler [1976] reported collection overheads of between 10% to 40% for Lisp, with pause times between mutation of 4.5 seconds every 79 seconds [Foderaro 1981]. In the '80s, state-of-the-art collectors for Smalltalk-80 achieved less than 5% collector overhead, with typically less than 100-millisecond pause times [Ungar 1992]. Assuming two orders of magnitude increase in processor speed over the decade, this represents an order of magnitude reduction in pause time.

Very few papers have considered file management from the point of view of garbage collection [Rosenblum and Ousterhout 1992]. There are, however, many tools that use the garbage-collection techniques for optimizing and repairing hard disks, Unix's fsck for example. Many object-oriented databases, object servers and persistent stores such as O2 [Delobel et al. 1995] and Object-Store [Lamb et al. 1991] have garbage collectors. While there have been numerous papers on garbage collection of data structures, they tend to be language-specific, which is problematical because some languages allow optimizations that are not generally applicable. Language semantics can restrict the topology of the computation graph, which may be cyclic, acyclic, or a polytree tree. (A polytree is a singly connected graph.) In unoptimized functional languages, the graph is predominantly treelike [Clarke 1977]. This is not the case for the purer object-oriented languages such as Smalltalk, which make extensive use of cyclic data structures. The topology of the computation graph in turn (as will become clear) constrains the type of collector that can be effective.

Cohen [1981] refined the collector process, C [Dijkstra et al. 1978] to two subprocesses: identification, I, and reclamation, R. This can be extended into a taxonomy of collectors by distinguishing two classes of identification, *direct* and *indirect* (Figure 2). Direct identification of garbage (also called reference counting) identifies cells that have no references to them. Indirect identification identifies live cells—what remains of the total store must be unallocated store and garbage. While direct identification is local in nature, indirect identification is global—it requires tracing live cells from the roots. The latter is called the *tracing family* in Lang and Dupont [1987].

The form of reclamation depends on how free-store is managed. It can be managed as a *free-list* (equally well as a bitmap or buddy system) or a *heap* (linear allocation) [Wilson et al. 1995]. If managed as a free-list, contiguous garbage can be coalesced to provide larger cells. If managed as a heap, a single reference, the *top of heap*, indicates the division between allocated and unallocated store. Reclamation can be per-

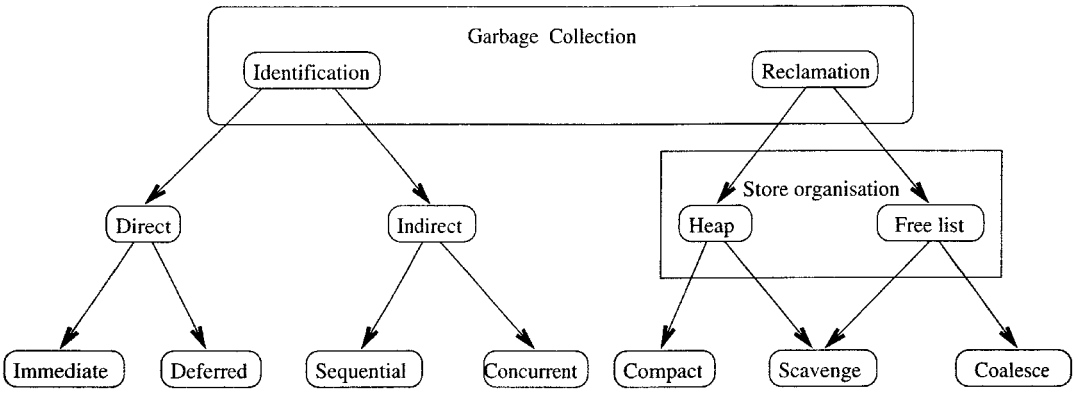

Figure 2. Garbage collector taxonomy.

formed either by compacting or scavenging.

Various collectors have been proposed that seek to optimize different criteria. Some aim to minimize the ratio of time spent collecting to the time spent mutating; some aim to minimize the time taken in any one invocation of the collector to provide predictable performance for *reactive* (interactive or real-time) systems; some aim to minimize the space overhead (the additional memory required to identify and reclaim garbage; and some are concerned with localization to prevent thrashing in virtual memory systems. The following sections further refine the taxonomy, outlining the advantages and disadvantages of different species.

2.2 Direct Identification of Garbage (Reference Counting)

Direct identification of garbage is effected by a *reference count*: a record of the number of references to a cell [Collins 1960; Weizenbaum 1963]. Each cell carries the space overhead of an extra header field to hold the count.

2.2.1 Immediate Identification and Reclamation

In the simplest form, garbage identification is coupled to mutation—after every mutation the count is adjusted. If as a result of mutation a cell count falls to zero, the cell is garbage and is reclaimed immediately [Collins 1960]. The collector reclaims cells recursively, decrementing the counts of referents and reclaiming as appropriate. This process is known, naturally enough, as *recursive freeing*. It is illustrated using a slightly elaborated informal notation introduced by Watson [1986]:

```
(M I) (M I)(M I). . .(R I) (R I) (R I). . .
(M I)(M I) (M I). . .(R I) (R I). . . .
```

The parentheses indicate atomic actions. The notation illustrates that recursive freeing can cause unbounded mutator delays.

A problem with immediate reference counting is that one component of the time spent in identification is proportional to the number of mutation operations [Steele 1975; Ungar 1984]. Significant time is also spent in recursive freeing: 5% on Berkeley Smalltalk and 1.9% on Dorado Smalltalk [Ungar 1984]. Because recursive freeing is unbounded, immediate reference counting can cause indeterministically long pauses in mutation, and so is unsuitable for reactive applications.

2.2.2 Deferred Reclamation

The problem of recursive freeing can be alleviated by deferring reclamation. Using doubly linked free-list store management [Weizenbaum 1963], a newly

deallocated cell is placed on the end of the free-list, but its referents are not immediately reclaimed. The referents are reclaimed when the cell advances to the head of the free-list. Only when the cell is reallocated are the counts of its immediate referents decremented and added to the free-list. In this regime, garbage collection is local, fine-grained and interleaved with mutation:

```
(M I) (M I) (R I) (M I) (M I)(R I). . .
```

Collectors that underestimate the extent of garbage in a single invocation of the collector are said to be *incomplete*. Such temporarily (or permanently) unidentified or unreclaimed garbage is known as *floating garbage*.

Deferred reclamation provides a smoother collection policy, one not so vulnerable to unbounded mutator delays. A scheme similar to Weizenbaum's but more suitable for arbitrary-size cells is that of Glaser and Thomson [1985]. It uses a *to-be-decremented* stack, TBD, instead of a doubly-linked list. In this scheme, references to cells are pushed onto the TBD stack if they have a count of one that is due to be decremented. When cells are allocated from the stack their count is already one, so the scheme also manages to elide clearing and setting the count.

2.2.3 Deferred Identification

Deutsch and Bobrow [Deutsch 1976] observe that frequently, over a series of reference-counting operations, the net change in a cell's reference count is small, if not nil. For example, when duplicating a cell reference as a stack parameter to a procedure call, the cell acquires a reference that is lost once the procedure call returns. If adjusting such volatile references can be deferred, many garbage-identification operations can be elided.

Baden [1983] proposes such a scheme for Smalltalk-80, which was implemented by Miranda [1987]. References to cells from roots, such as the stack, are not included in a cell's count. Instead, root-only referenced cells are recorded in a *Zero Count Table* (ZCT). If a reference to a new cell is pushed on the call stack (the typical way new cells join the computation graph), a reference is placed in the ZCT. When a nonroot reference-counting operation causes a cell's count to fall to zero, a reference is also placed in the ZCT because it might still be referenced from a root. When the ZCT fills up or when no free store is available for cell allocation, the collector preferentially reclaims cells referenced by the ZCT. Reference counts are first *stabilized* (made consistent) by scanning the roots and increasing the count of all referenced cells. The ZCT is emptied by scanning the table and any cell with a zero count is freed. Finally, the roots are scanned again and the counts of cells referred to from roots are decremented. During this process any cells whose counts return to zero are placed in the ZCT because they are now only referenced by roots.

With this technique, stack pushes and pops are made without identification operations. Baden's measurements of a Smalltalk-80 system suggest that this method eliminates 90% of the reference-count operations and reduces the total time spent on garbage collection by half [Baden 1983]. A potential disadvantage is that scanning the ZCT causes a pause in mutation. However, typical pause times are of a few milliseconds [Miranda 1987]; a further disadvantage is the extra storage required by the ZCT.

2.2.4 Space Overhead and Overflow

Another drawback of reference counting is the space overhead of the count field. It has been observed [Krasner 1983] that the majority of cells have a small count. To reduce the space overhead, the size of the count field of a cell is often chosen smaller than needed. Typically, systems allocate one byte to hold the count. To prevent *overflow* once a count becomes *saturated*, it is not al-

tered and no longer accurately reflects the number of references to the cell. To minimize the test for saturation, a signed byte is used to hold the count; a count is saturated if the byte is negative. This only allows a count to record accurately up to 127 references.

Clark's measurements of Lisp programs [Clark 1979] show that about 97% of list cells have a reference count of 1. This suggests an extreme form of saturation using a *single-bit* count [Friedman and Wise 1977; Chikayama and Kimura 1987]. A clear bit is used to indicate a single reference to a cell. When a second reference to the cell is created the bit is set. Once set, the bit cannot be cleared because, without tracing from the root, it cannot be determined if the cell has more than one reference. To reclaim cells that acquire more than one reference during their lifetime, it is necessary to use a second collector that uses indirect identification. Because of the predominance of single references, the indirect collector will be invoked considerably less often than if it were used alone.

2.2.5 Memory Leaks

An important aspect of direct identification is its dependence on local information (the count). As a result of mutation, a subgraph may become detached from the computation graph, yet the reference count of none of its cells is zero. This will happen if the mutator can generate cycles. Reference-counting schemes do exist that attempt to collect cycles of garbage, but they are complex [Friedman and Wise 1979], have significant computational overheads and lack generality. Bobrow's [1980] concern is functional languages while Brownbridge [1985] specializes in combinator graph-reduction machines.

Brownbridge [1985] uses two types of reference. *Strong* references form an acyclic graph of accessible cells, while *weak* references complete cycles (Figure 3). A weak reference is intended to reference a cell with a strong reference. A

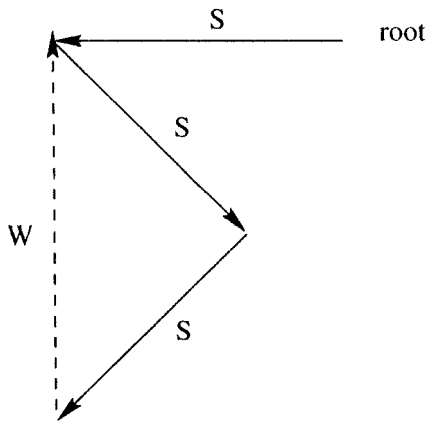

Figure 3. Weak and strong references.

cell holds two reference counts: one for strong references, SC, and the other, WC, for weak. When the deletion of a strong reference results in a zero SC and nonzero WC, there is a possible cycle of garbage. To determine if the cell is garbage, Brownbridge's scheme recursively traces the cells' referents. If a cell is located with zero SC, the weak reference is made strong. If a cell is located with SC greater than one, the reference is made weak and the search terminated. If the trace returns to the starting point without having located a cell with SC greater than one, the cycle is garbage and can be reclaimed.

Tracing can spread over arbitrarily large parts of the computation graph. The algorithm fails when there are intersecting cycles (e.g. Figure 4). With two mutually referencing cycles, each will regard the other as an external reference.

Hughes [1984] gives a scheme based on identifying *strongly connected components*, SCCs, of a graph. SCCs are those subgraphs for which there is a cycle at each vertex. Such SCCs have their own reference count. SCCs are merged to produce larger SCCs so that no cycles of SCCs are formed. During mutation, SCCs can be created, destroyed, split, or amalgamated. The major complication is splitting. A split requires tracing the graph looking for

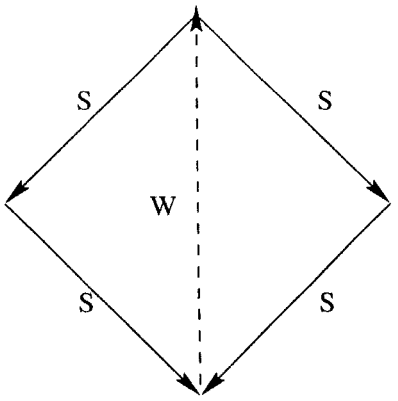

Figure 4. Mutually referencing cycles.

newly formed SCCs—the tracing is unbounded.

2.3 Indirect Identification of Garbage (Tracing)

The problem of cycles of garbage is usually overcome by a collector that identifies garbage indirectly. Recursively traversing the computation graph from the roots identify all cells that are alive. By default, the unvisited part of the store is unallocated or garbage. By such global means, cyclically connected subgraphs that become disconnected are indirectly identified and can be collected.

2.3.1 Mark and Scan

In their simplest form (stop-the-world), *mark-scan* collectors delay collection until the unallocated store is exhausted [McCarthy 1960]. Mutation is then temporarily suspended. Identification (marking) and reclamation (scanning) are treated as sequential phases. The identification phase traces the computation graph from the roots, *marking* all accessible cells. A single *mark-bit* is sufficient to indicate whether a cell is referenced by cells reachable from a root. The marking phase concludes when all accessible cells have been marked. The second phase, a *scan* of the entire store, reclaims the unmarked cells and clears the marked ones. Consequently, mark-scan has a much coarser grain than reference counting:

```
M M M. . .(I I I. . .) (R R R. . .) M M M. . . .
```

The parentheses here indicate that identification and reclamation run to completion.

The mark-bit is a single-bit reference count, but is different from the single-bit count of Section 2.2.4. The former indicates zero and one or more references; the latter indicates one and more than one reference. Immediate single-bit direct identification is also distinguished from mark-scan by the periods of time for which the mark bit is consistent. For mark-scan, the information is only accurate at the end of the mark phase. For immediate direct identification, it is made consistent after each mutation. Deferred direct identification reference counting is distinguished from mark-scan because it is incomplete.

If cells have differing sizes, allocation will fragment the free store. When an allocation request is made, the free-list may contain no free cells of the required size, but may contain cells larger than required. Typically, the allocator satisfies a request by splitting a larger cell into an allocated cell and a remaining free fragment. Over time, the free-list becomes composed of small fragments. Eventually a scenario can occur where no free cell is large enough to meet the allocation, yet the total free space is sufficient. The allocation may be met by coalescing contiguous fragments.

Compaction can be provided by heap store management [Cohen 1967]. For fixed size cells, compaction can be performed by scanning the heap twice (known as a *sweep*). In the first pass, two references are used, one pointing to the bottom of the heap, the other to the top. The reference to the top of the heap is used to scan down until it references a marked cell. The reference to the bottom of the heap scans up until it comes to an unmarked cell. At this stage, the contents of the cell indicated by the top reference is copied to the unmarked cell

(assuming the cells are the same size), the mark-bit is cleared and a *forwarding reference* to the new cell placed in its previous position. When the two references meet, all marked cells have been unmarked and compacted in the lower part of the heap. A second scan is needed to readjust references to moved cells. Indirect references from cells in the compacted area, by way of the cleared area, are made direct.

In some schemes compaction is controlled by the mutator. The allocator in BrouHaHa Smalltalk [Miranda 1987] checks whether the total size of free cells is insufficient, and if so, invokes a compactor. If compaction proves futile, the collector is invoked. Martin [1982] combines the marking phase with a rearrangement of the references so that cells can be moved more readily. Carlsson et al. [1990] present a variation suitable for cells of varying sizes. During the mark phase, the reference fields of the accessible cells (not the data) are copied to a table. After sorting the addresses, the reachable cells are compacted by "sliding" the cells to one end of the store.

Mark-scan has large pause times. The scan time is proportional to the size of the store. In virtual memory systems, the collector may access numerous pages on secondary store, an inherently slow process. As such, mark-scan is unsuitable for reactive applications. Even if the garbage collector goes into action infrequently, when it does no reaction is possible.

2.3.2 Concurrent Mark-Scan

A major advantage of deferred direct identification is that identification and reclamation have a fine grain size. This makes it suitable for interactive and real-time applications [Goldberg 1983]. Dijkstra et al. [1978] describe a variation of mark-scan in which the mutator and the collector operate concurrently, called *on-the-fly* garbage collection. The concepts of mutator and collector were coined in this context.

In the simple mark-scan scheme of Section 2.3.1, concurrency is not possible because of possible interference of identification with mutation. If a reference to a new cell is added after the identifier has passed over its referents, the new cell is not recognized as part of the computation graph and so collected. Dijkstra et al. achieve a decoupling of the mutator from the collector by introducing a third state for a cell. The three states referred to, perhaps inappropriately, as colors are white (unreachable); black (reachable); and gray (possibly reachable). They can be realized by two mark bits.

Two or more processes, one or more responsible for mutation and exactly one for collection, run concurrently:

```
M M(M I) M M. . .(I I I. . .) (R R R. . .)
(I I I. . .). . . .
```

The mutator aids the marker by setting a cell gray at the point of allocation. In the marking phase, the roots of the graph are initially marked gray. The identification process scans the heap graying all descendents of a gray cell and then blackening the cell. As previously, white cells are unreachable from the roots. In the scan phase, white cells are reclaimed and black cells are whitened.

The collector is incomplete because it may take two cycles to reclaim a dead cell. Dead gray cells are blackened in one invocation of the marker and then whitened in the next. With direct identification, an incomplete collector underestimates the amount garbage cells. With indirect identification an incomplete collector overestimates the live cells. Dijkstra et al.'s scheme allows concurrency of mutation and collection, but the phases of identification and reclamation are strictly serialized. This has the effect that a mutator may still have to wait until a collection finishes if there is no free store to make an allocation. Starvation of the mutator is avoided by Queinnec et al. [1985].

Despite the decoupling the extra color

gives, concurrency of the mutator and collector has to be carefully controlled to prevent unsoundness. Several purported implementations have contained subtle synchronization problems [Andrews 1991]. A sound solution requires the mutator and collector actions of testing and changing colors to be atomic. Without atomic actions, concurrency leads to lost updates, sometimes called the test-and-set problem.

Wadler [1976] has shown that a concurrent mark-scan collector uses a greater proportion of the computation time than the sequential scheme. This might be expected because of the overhead of atomic operations and because the collector runs even when there is no garbage to collect.

2.3.3 Scavenging Collectors

The generality and modularity of compacting mark-scan account for the attention it has received in the past three decades. The language implementations of the 1960s for which mark-scan collection was originally intended had small physical memories (by current standards). For small address spaces, the execution cost of scanning the entire store is negligible. With large modern systems, compacting mark-scan is inefficient because of its global nature. The marking phase inspects all accessible cells while the sweep phase traverses the whole store twice. Ungar [1984] reported that Fateman found mark-scan to take 25% to 40% of the mutator time of Franz-Lisp programs. Wadler [1976] reported that typical Lisp programs spend from 10% to 30% of their time collecting.

The cost of the scan phase of mark-scan is proportional to the total size of store. This phase can be eliminated if, rather than scanning, live cells are relocated as they are identified. The store is managed as two heaps, historically called *semi-spaces* [Fenichel and Yochelson 1969]. The mutator begins allocating in *from-space*. When the heap is exhausted, the collector scavenges. A *scavenge* is a simultaneous traversal and copy of the computation graph from from-space to the second heap, *to-space*:

```
M M M. . .(I R I R I R. . .)(R R R. . .) M M M. . . .
```

Multiple mutations are followed by combined identification and reclamation. In Fenichel and Yochelson [1969], when each cell is moved to to-space, a forwarding reference is left behind. (This can be compared with the forwarding reference of mark-scan collectors, Section 2.3.1). After a scavenge, a scan of to-space is needed to redirect references to from-space. From-space then becomes free and can be reused. The two semi-spaces are *flipped* and the mutator continues allocating in the new from-space. This combination of tree traversal and copying also has the advantage of improving locality, which is beneficial in virtual address spaces.

2.3.4 Incremental Scavenging

The Fenichel–Yochelson scheme appeared in the late 1960s, but only in the late 1970s had technology changed sufficiently that new algorithms for garbage collection were required. Processors became faster, memories became larger, and programs became significantly larger. There is a Parkinson's law in operation: programs expand to fill the memory available. As program data increased from tens of kilobytes to megabytes, the time required to collect garbage increased dramatically. By the late 1970s, pauses resulting from garbage collection could last tens of seconds or more. At this time, Baker [1978] proposed a modification of Cheney's [1970] compacting algorithm that avoided substantial collector interruptions.

In Baker's incremental scavenger, the mutator is given some responsibility for reclamation. After the from-space becomes full, the mutator allocates new cells in the to-space. Each time the mutator allocates a cell in to-space, a number of live cells are traced and copied from from-space to to-space. This means

that the two semi-spaces are simultaneously active. A consequence of this is that mutation and collection are interleaved:

```
(M I R) (M I R I R). . . .
```

By distributing mutation through collection, the conservative scavenging collector gives bounded collection when cell size is bounded. Baker examined the effect of varying the number of cells traced at each invocation on the memory requirements. He concluded that the maximum memory requirement of conservative scavenging is similar to the use of two mutators running concurrently. Scavenging schemes trade space for time because they require two heaps. Consequently, they have much higher space overhead than either mark-scan or reference-counting algorithms. A major reason for their success is that virtual memory appears cheap, so flagrant use of address space becomes acceptable. It is, of course, paid for by I/O costs.

Baker [1992] recently described a collector that is isomorphic to his original conservative copying algorithm [Baker 1978], but does not require relocation. Baker recognized that the "spaces" of a scavenging collector are just manifestations of sets of cells. Any other reification of sets would do just as well. All that is necessary for any cell is to identify which set (from-space or to-space) it belongs to. It need not be copied if its allegiance can be transferred from one set to another. Baker requires two reference fields and a *color* field for each cell. The reference fields link each cell into doubly-linked lists that implement sets. The color field indicates which set a cell belongs to. The colors of Baker can be compared with the colors of Dijkstra et al.'s [1978] concurrent mark-scan collector. The colors gray and black serve to distinguish alternate collections.

In Baker [1992], all free store is initially in *from-set*. An allocation reference serves to divide the list into the part that has been allocated and the remaining "free" part. Allocation is as efficient as heap store management because it only requires advancing the pointer by the size of the new cell. When the free space is exhausted, the collector traverses the reachable cells and "moves" them from the allocated from-set to *to-set* by unlinking the cell from from-set, toggling its color field, and linking it into to-set. When all the reachable cells have been traversed and reassigned from from-set to to-set, from-set is known to contain only garbage, and is therefore a list of free store.

Free-list store management is best suited to languages that use equal-size cells. If cells of different sizes are managed, the free-list must be searched to find a cell of appropriate size. This can lead to fragmentation, poor locality of reference, and thrashing. These are just the problems that copying solves.

2.3.5 Generation Scavenging

Lieberman and Hewitt [1983] observed that cells tend to die young and that long-lived cells are typically very long-lived. Having to copy long-lived cells for every invocation of the collector seems extravagant. Baker [1992] and Dijkstra et al.'s [1978] collectors are incomplete and effectively distinguish new cells from old cells with colors. Lieberman and Hewitt's collector segregates cells into generations, each with its own pair of semi-spaces. Each generation may be scavenged without disturbing older ones. Younger generations are scavenged more frequently. The youngest generation will be filled most rapidly, but on flipping very few cells survive. This drastically reduces the amount of copying. Generations can be created dynamically when the youngest generation fills up with cells that survive several flips.

Ungar [1984] presents a simpler, more efficient generation scavenger. This collector classifies cells as either *new* or *old*. Old cells live in a region of store called *Old-Space*, OS. Old cells that reference new ones are members of

the *Remembered-Set*, RS cells are added to RS as a side effect of the mutator. Cells that no longer refer to new cells are removed from RS when scavenging. All new cells must be reachable from RS, and so RS behaves as roots for the new cells. Any traversal of new cells only needs to start from RS.

Three heaps are used for the new cells: *New-Space*, NS, a large nursery heap where new cells are spawned; *Past-Survivor* space, PS, which holds new cells that have survived previous scavenges; and *Future-Survivor* space, FS, which remains empty while the mutator is in operation. A scavenge copies live cells from NS and PS to FS space and then flips PS and FS. Cells that have survived more than a prescribed number of flips are copied to OS, a process called *tenuring*. With Ungar's collector, the mutator is stopped during scavenging. This elides forwarding references and achieves some performance gains. While explicitly not concurrent, pause times are short because generations are small. By carefully tailoring the size of NS, FS, and PS, an implementation of Ungar's scheme for Smalltalk manages to keep scavenge times to a median of 150 milliseconds occurring every 16 seconds [Ungar 1984] on a Sun workstation.

Other generation-based collectors include: *opportunistic* collectors [Wilson and Moher 1989]; *ephemeral* collectors used in Symbolics machines [Moon 1984]; and the Tektronix Smalltalk collector [McCullough 1983]. All three commercial Smalltalk systems, Digital, Tektronix, and ParcPlace, adopted generation scavengers [Ungar and Jackson 1988]. The New Jersey SML compiler [Wilson 1992a] also includes a generation collector. Demers et al. [1990] have investigated a generation scheme combined with a mark-scan garbage collector for use with Scheme, Mesa, and C intermixed in one virtual memory. Before Demers et al. [1990], many believed that only scavenging collectors could be made generational.

Wilson et al. [1990] show that generation scavengers typically have poor locality of reference, but careful attention to memory hierarchy issues greatly improves performance. They attributed the small success recorded by several researchers in their attempts to improve locality to two flaws in traversal algorithms. They failed to group data structures in a manner reflecting their hierarchical organization. What is more important, they ignored the disastrous grouping effects caused by reaching cells by linear traversal of hash tables (i.e., in pseudo-random order).

A generation scavenger that adapts to the allocation patterns of applications was presented by Hudson and Diwan [1990]. This collector has a variable number of fixed-size (power of 2) generations. The generations are placed in store at contiguous addresses. The generation is apparent from the most significant bits of the address. Each generation has its own to-space, from-space, and RS (remembered-set). RS is fed indirectly through a buffer containing addresses of possible intergeneration references. The feeder may filter out duplicates, intrageneration references and nonreferences. When scavenging more cells than a generation can accommodate, a new generation is inserted. To retain the ordering, the younger generations are shuffled backwards during scavenging. Conservative collectors that copy cells when the mutator addresses them have also been looked at by White [1980] and Kolodner [Kolodner et al. 1989; 1991]. These reorder cells in the order they are likely to be accessed in the future, giving improved locality. However, the technique requires special hardware. Other reordering optimizations that don't require special hardware work by reordering pages within larger units of disk transfer [Wilson 1990].

Although generation collectors are the most complex single-processor collection schemes, they suffer poor performance if many cells live just long enough to be promoted before dying, the so-called *premature tenuring problem*. Ungar and

Jackson propose an adaptive tenuring scheme based on extensive measurements of real Smalltalk runs [Ungar 1988; 1992]. This scheme varies the tenuring threshold depending on dynamically measured cell lifetimes. It also proposes a refinement that has been included in the ParcPlace [1991] collector. With languages like Smalltalk, interactive response is at a premium and many large cells, mainly bit-maps and strings, don't contain references to other cells. To avoid copying these cells, they are segregated in a *large-cell* space and tenured to OS (old space) when opportune.

Multigenerational collectors have to cope with the *waterfall problem* [McCullough 1983]: collecting a particular generation requires collection of all younger generations. The result is that pause times will be longer for older generations. While generation collectors collect intrageneration cycles, they cannot collect intergeneration cycles of references that cross more than one generation. Some schemes do not attempt to scavenge old generations. In persistent stores, reclamation of such garbage is often left to *off-line* reorganization [Ungar 1984], where a full garbage collection is done after the system has been stopped. The ParcPlace [1991] Smalltalk-80 generation garbage collector is backed up by a mark-scan compactor that collects OS.

2.4 Hybrid Collectors

To tackle the memory leaks of cyclic structures, Martinez et al. [1990] combine simple reference counting with a local mark-scan. Besides the reference count, an extra field holds the *color* of the cell: green, red, or blue. The general idea is to perform a local mark-scan whenever a reference to a shared subgraph is deleted. That is, local mark-scan is initiated each time a reference is deleted to a cell with counter greater than one. Marking starts from the deleted reference, decrements the counter, and sets the color *red* (possible garbage). The subgraph is then rescanned; any subgraphs with external references (nonzero count) are remarked *green* (accessible) and their counts restored. All other cells are marked *blue* (garbage). At the end of the cycle, all blue cells are part of a dead cycle and may safely be reclaimed.

The algorithm has one major problem: the need to perform a local mark-scan every time a reference to a shared cell is deleted. This increases the complexity of the local mark-scan to $O(n)$, where n is the size of the shared subgraph. In unoptimized functional languages, most structures have a reference count of one [Clarke 1977], and the cost of the algorithm is exactly the same as the standard reference count. This is not the case for purer object-oriented languages like Smalltalk, which make extensive use of sharing and cyclic data structures, making the overhead of this scheme high.

Lins [1990] addresses the problem by introducing an extra state information in the form of a fourth color, black, and a *control queue*. This data allows the mark-scan to be done lazily. As in Weizenbaum [1963] and Glaser and Thomson [1985], subgraphs are not scanned immediately, but are queued in a special list, the *control queue*, and the root cells set black. When the allocator is unable to supply memory, the control queue is scanned to reclaim possible garbage cycles. Lins and Vasques [1991] found that, with appropriate management of the control queue, no unnecessary calls to mark-scan are made.

Lins [1992] applies the concept of the cell age from generation scavenging to the problem of cyclic reference counting. A second counter records the age of cells. A global *time* counter is initialized to zero and is incremented every time a cell is allocated from the free-list. Lins profits from the age information in two ways. First, as most cells die young, mark-scan is initiated from the youngest cell in the control queue. Second, the age information gives a check on the absence of cycles. A sufficient but not

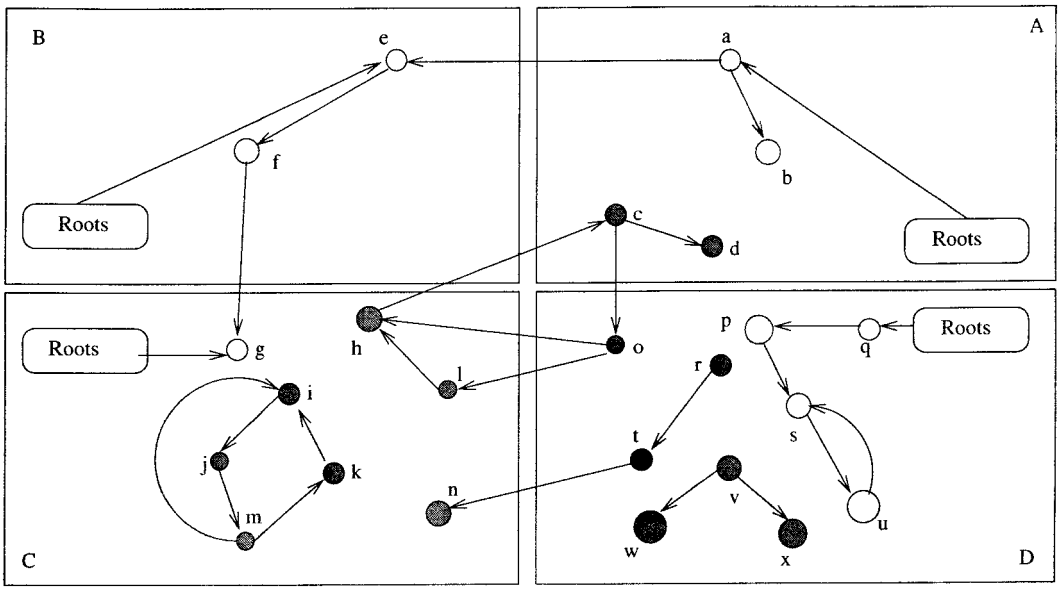

Figure 5. An illustrative, but small, distributed computation graph.

necessary condition for the absence of cycles is that younger cells do not reference older cells. During cycle detection (red marking phase), a check for the condition that *the parent cells are older than their offspring* is made. If at the end of the mark phase the condition is true for all traced cells, the graph is acyclic. As a result, cells can be put directly into the free-list or restored to their original status without having to be set blue.

3. DISTRIBUTED GARBAGE COLLECTORS

For the purposes of this paper, a distributed system means a collection of autonomous sites that share a communication facility for exchanging messages. Each site has its own store, at least one mutator, and at least one stack. The computation graph and roots are distributed over a number of sites (Figure 5). A similar structure is exhibited by distributed file systems [Garnett and Needham 1980], distributed object-oriented databases [Delobel et al. 1995] and web pages.

A reference to a cell in the same site is said to be *local*. A reference to a cell on another site is said to be *remote*. Four classes of garbage subgraphs are exemplified in Figure 5:

- intrasite acyclic garbage (e.g., v, w, x);
- intrasite cyclic garbage (e.g., i, j, k, m);
- intersite acyclic garbage (e.g., n, r, t);
- intersite cyclic garbage (e.g., c, d, h, l, o).

Processing power is necessarily localized in sites. Each site has direct access only to those cells that live in its local store. Access to a remote cell is achieved by sending a message to the site on which it lives to spawn a task to perform the required operation. Because there is no global address space, reference to a remote cell is necessarily indirect. A cell references a remote *import* record and the import record references a local cell. To simplify remote references, a cell can reference a local *export* record that in turn references the remote import record (Figure 6).

Indirection due to the absence of a homogeneous address space is compounded because message transmission between sites is unreliable: messages

Figure 6. A remote reference.

may be duplicated, delivered out of order or lost, and sites may temporarily be incommunicado. Establishing reliable message transmission between computers is a complex problem. This complexity is handled in communication systems by the principle of division of concerns. The communication system is layered, lower layers guaranteeing service properties to higher layers. Typical properties offered to the application layer are that

- messages are not lost,
- messages are not duplicated, and
- messages are delivered in mutual-causal order (FOFI[1]) between pairs of sites.

The Internet TCP/IP protocols have emerged as the *de facto* open system interconnection. The main tasks of the *Internet Protocol*, IP layer are the fragmentation of messages into packets and the routing of packets to destination machines. The size of a packet is determined dynamically by a number of factors that include network loading. IP makes a "best effort" to forward packets to the next destination, but forwarding is not guaranteed. If a router is overrun with packets, it discards them. If a router fails, other routers send packets along alternative paths. Thus, packets may be duplicated, arrive out of sequence, and take a relatively long time to arrive intact at their destination.

Above the IP layer, TCP eliminates duplicates and reassembles the packets in their correct order. In more detail, TCP, like Unix, is byte-oriented. A sequence number gives a position in the byte stream of data so far exchanged. A checksum is applied to each packet. A number of packets received intact (checksum agrees) can be acknowledged ith a byte position. A packet is retransmitted if it has not been acknowledged after a certain time, the *time-out*. The time-out is determined by network loading. Each message is received once and messages between any two sites are received in the order in which they were sent—mutual causal ordering.

To add to the problem of reliable message transmission, a remotely spawned task is not acted upon immediately. Once accepted, the task is added to the task queue and must wait its turn. This means that remote tasks may take a considerable time (relative to machine instruction execution) to be acted upon. *Latency* is the elapsed time between the issue of a remote task-request message and when it is executed. The latency is typically orders of magnitude greater than an instruction cycle, particularly for RISC processors. For efficiency, avoiding processor idling while waiting for a response is vital. In most evaluations of distributed garbage collection, communication overhead is the principal metric. Published measurements [Bennett 1987; Schelvis and Bledoeg 1988] indicate remote cell access to be slower than local by three to four orders of magnitude.

Early distributed garbage collectors were, naturally enough, based on single-address-space collectors. Develop-

[1] First Out First In.

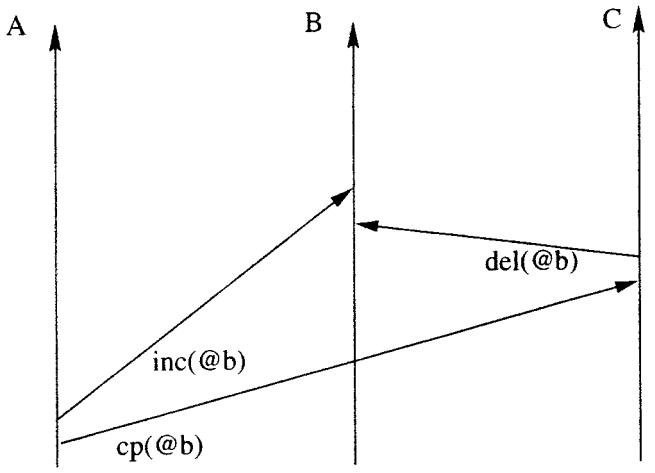

Figure 7. Site A increments the count.

ments in distributed garbage collection can well be understood as the trial-and-error adaptation of the ideas developed for single-processor, single-address-space collectors to the distributed environment. Garbage collection processes, mutation, identification, and reclamation, are necessarily decomposed according to site boundaries. The high cost of communication relative to local computation make efficient distributed garbage collection difficult enough. This problem, as will be illustrated, is compounded by the problem of indeterministic latency.

3.1 Direct Identification of Distributed Garbage

It was noted in Section 2.2 that, with direct identification, the processes of identification and mutation are localized. This initially seems ideally suited to distribution, as (M I) phases can run concurrently on different sites. Only small portions of the graph that have been affected by a mutation need be considered for reclamation. These can be reclaimed concurrently on different sites. A straightforward attempt to use reference counting in distributed environments exposes the problem of indeterministic latency. A succession of improvements elide the problem by transferring successively more state information from the cell to the reference. At the same time, these solutions reduce the communication overhead.

3.1.1 Distributed Reference Counting

One of the earliest distributed reference-counting collectors was described by Nori [1979]. To preserve the reference count invariant, it is essential that messages are not duplicated. Even if the communication system does filter duplicate messages, counts can become unsound if a decrement count task is acted on before a corresponding increment task, as illustrated in Figures 7 and 8. In these timing diagrams, the horizontal axes indicate spatial distribution and the vertical axes increasing time. A message between sites is represented by an arrow—the tail of the arrow below the arrowhead reflects the latency. Suppose site A duplicates a reference to a cell b on site B by sending a cp(@b) message to site C. (With a C-like notation, @n denotes the reference to an entity denoted by n.) If either A or C is responsible for incrementing the count, premature reception of a decrement count task could lead to a dangling reference.

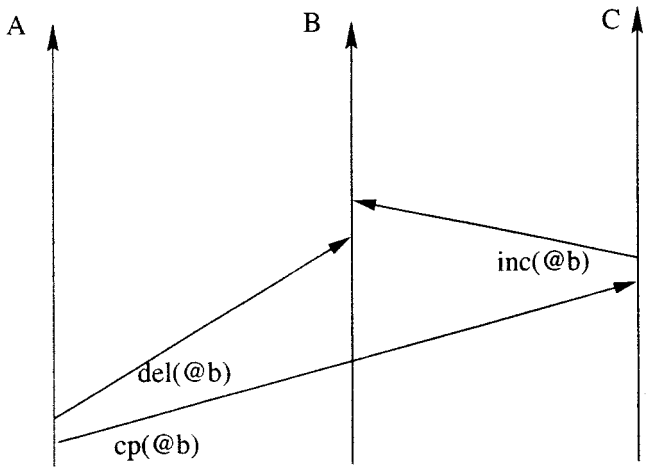

Figure 8. Site C increments the count.

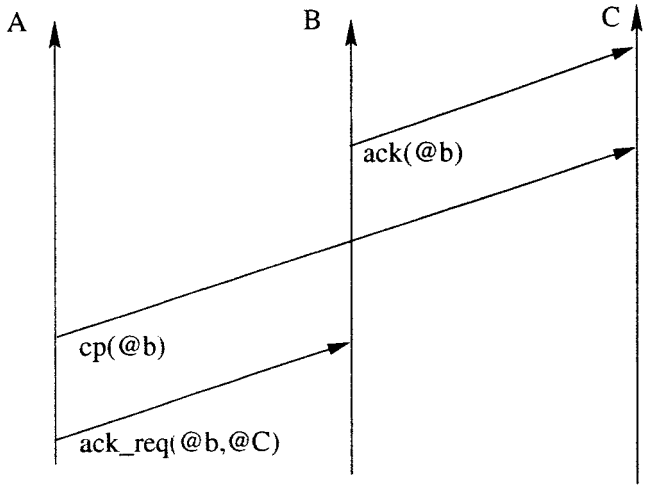

Figure 9. LM protocol.

To prevent these *race conditions*, Lermen and Maurer [1986] impose a protocol (Figure 9) on remote duplication and decrement tasks. When a site A that holds a reference to a cell b on site B wants to duplicate the reference on site C, it sends an *acknowledge-request* message, ack_req(@b, @C), to site B in addition to the copy message, cp(@b), to site C. The copy message provides C with a reference to b. The ack_req(@b, @C) message informs B about the creation of the new reference. When site B acts on the ack_req(@b, @C) task, it increments b's reference count and sends an *acknowledgment* ack(@b) message to site C. On arrival, this informs site C that cell b "knows" about the additional reference to it. Lermen and Maurer's [1986] protocol ensures that site C cannot send a *delete* message, del(@b), to site B before it accepts the ack(@b) message from B.

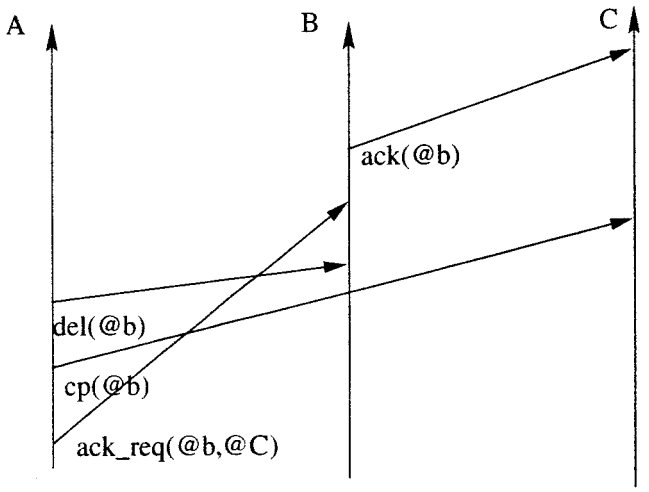

Figure 10. Non-FOFI order.

It is essential for soundness of the protocol that the messages be delivered in mutual causal order (FOFI) (Figure 10). The crossing of message arrows reflects the nonmutual causal ordering. Suppose site A sends a del(@b) message to site B soon after sending ack_req(@b, @C). If these messages do not arrive at B in the order they were sent from A, there can be premature reclamation of cell b.

3.1.2 Weighted Reference Count

An extension of reference counting that elides the problem of nonmutual causal order is WRC, *weighted reference counting*. There is some controversy as to its origin. The scheme was published at the same conference by Watson and Watson [1987] and Bevan [1987]. Watson and Watson attribute the algorithm to Weng [1979], but Thomas [1981] credits Arvind. The idea is to associate a *weight* with each reference. The count is only decremented, and so there can be no race conditions. The protocol guarantees preserving an invariant—the sum of weights of all the references to a cell is equal to the count of the cell.

To illustrate how weighted reference counting works, a cell is represented by a triple (Figure 11). (In general, a cell will have any number of reference fields, but one is sufficient to illustrate the mechanisms). When a cell is allocated (Figure 12), its count is set to the maximum the field can hold and the weight of the reference set equal to it.

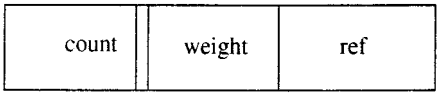

Figure 11. A cell with a single reference in WRC.

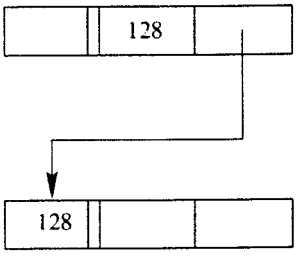

Figure 12. Creation of a new cell in WRC.

When a reference is duplicated (Figure 13), the weight of the reference is divided between itself and the copy—it is not necessary to access the cell. Only one message is required to duplicate a remote reference. The sum of the weights of references pointing to the cell remains unchanged.

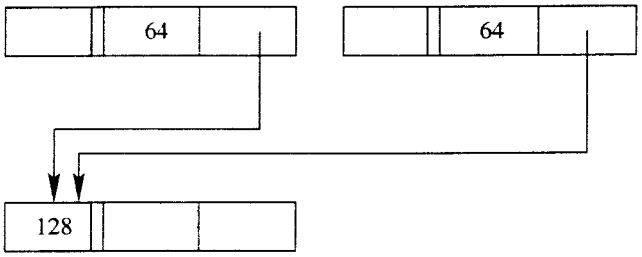

Figure 13. Duplication of a cell in WRC.

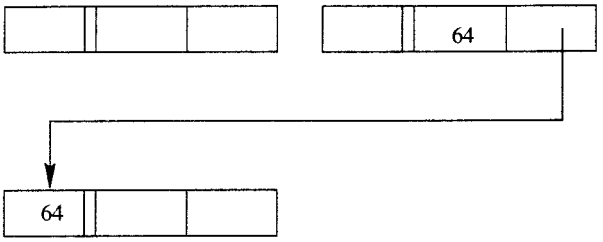

Figure 14. Deletion of a reference in WRC.

When a reference is destroyed (Figure 14), its weight must be decremented from the count to preserve the invariant. If this involves remote cells, a delete reference message, del(@cell, weight), is sent to the remote site hosting the cell. If a cell's count falls to zero, it is garbage and can be reclaimed.

Besides eliding race conditions, WRC reduces the communication overhead by eliminating the need for an increment message when duplicating a reference. This is achieved at the cost of space for storing a weight for each reference. If the weight is always a power of two, to allow for equal division, the $\log_2$ of the weight can be stored. This provides an important reduction in the space requirement. However, when a reference is deleted, the weight must be converted (by shifting) to effect subtraction, so increasing the overhead of identification.

A problem occurs, *underflow*, when a reference weight of 1 needs to be duplicated. A reference with a total weight W can have at most W references, each of weight 1. This could be overcome by adding a fixed number to the weight and the count of the reference. But this proposal is essentially the same as incrementing the count, and so suffers from the same race conditions as naive reference counting (Section 3.1).

A sound solution is the use of indirection illustrated in Figure 15. When a weight falls to one, an indirection allows further duplication. This has the disadvantage of requiring two messages to access a cell if a reference and its indirection live on different sites: one to the site hosting the indirection and one to the site hosting the cell. In a worst-case scenario, a long chain of indirections can be created. Once an indirection is created, it remains for ever. Rudalics [1990] calls this the *domino* problem. A reference consisting of a long chain of remote indirections may even loop back to a local cell a number of times.

3.1.3 Generation Reference Count

Generation reference counting, GRC [Goldberg 1989], provides another solution to the problem of duplicating a unit weight reference in WRC. This is achieved by replacing the weight by a

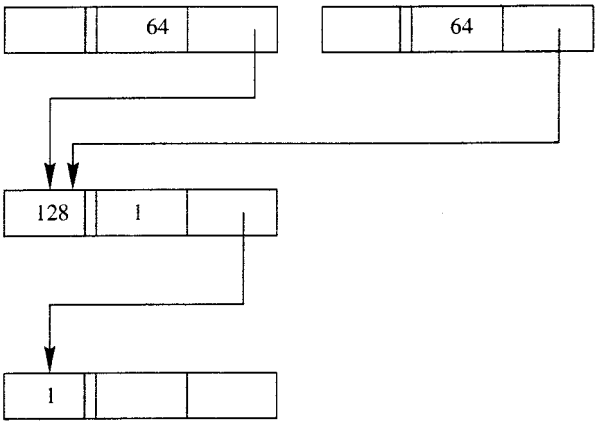

Figure 15. Indirection to duplicate a reference of weight 1.

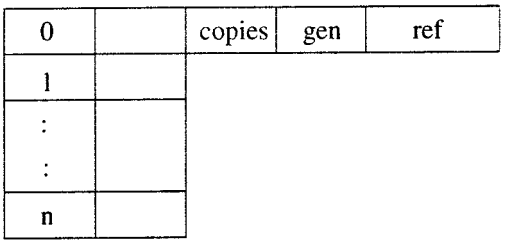

Figure 16. A single reference cell in GRC.

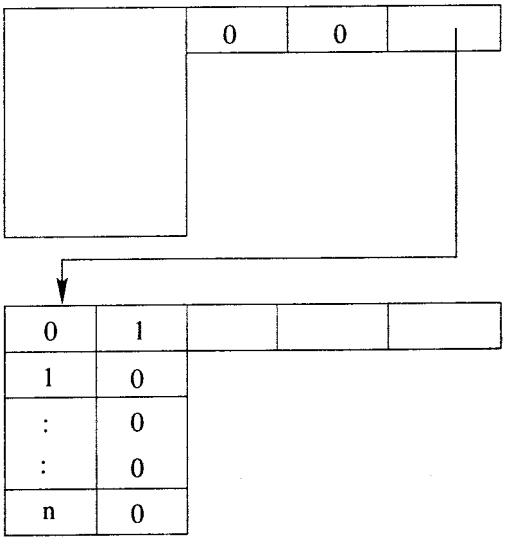

Figure 17. Allocating a new cell in GRC.

generation and a *copy count*. Each newly created reference is a zero generation reference. A copy of an ith-generation reference is an (i + 1)th-generation reference. The reference count is replaced by a table, called a *ledger*, which counts the references from each generation.

To illustrate, a cell is represented by a quadruplet (Figure 16). (As previously, a cell may have any number of references but one again is sufficient to illustrate the mechanisms.) The ith element of the ledger contains a count of ith-generation references.

When a new cell is created (Figure 17), the generation and count fields of the reference are cleared and the ledger is initialized. (When, as in Figure 17, the ledger contains no relevant information it is omitted.) When a reference is duplicated (Figure 18), a new first-generation reference is allocated.

When a reference is deleted (Figure 19a), a delete message del(@cell, n, copies) is sent to the host site. On receipt, the host decrements the copy count for the nth generation and increments the count for the $(n + 1)$st generation by the number of copies made. As a result, some elements of the ledger may hold negative values. This can occur when delete messages for $(n + 1)$th-generation references are acted upon before delete messages for the nth-generation references. For example, in Figure 19b, if the copied reference is deleted, rather

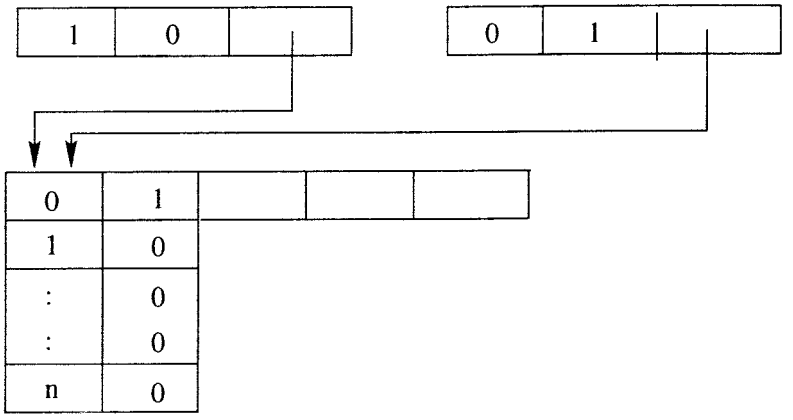

Figure 18. Duplicating a reference in GRC.

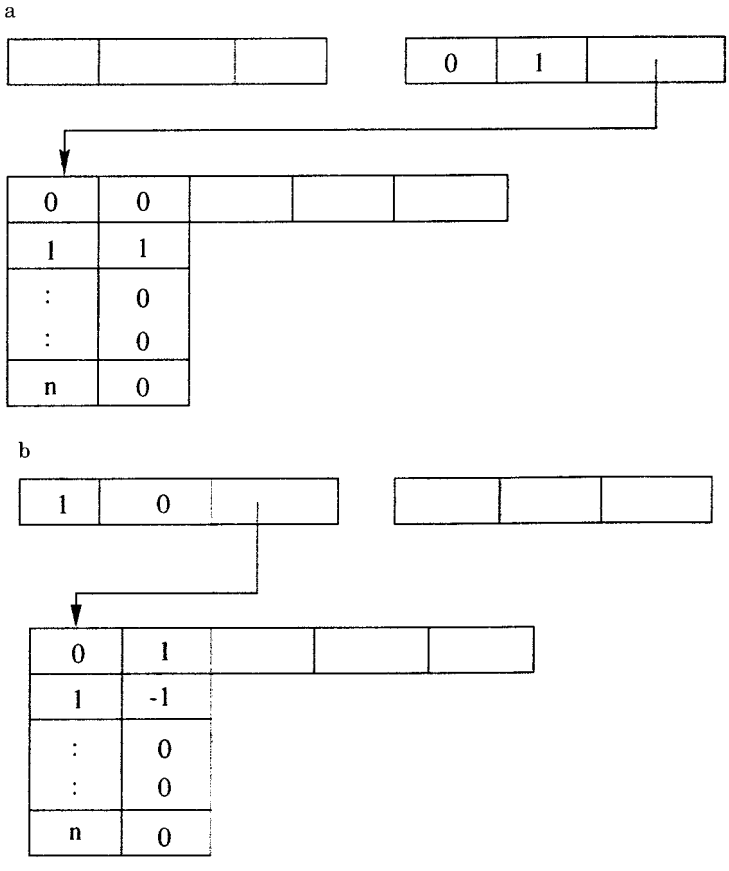

Figure 19. Reference deletion in GRC: (a) deleting a reference; (b) deleting a duplicated reference.

than the original, a ledger value will be negative. A ledger correctly indicates outstanding references to a cell for any order in which delete messages are received. A cell is only reclaimed if all the ledger entries are zero. This is only the

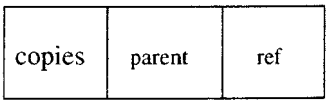

Figure 20. A reference in IRC.

case if a delete message for every reference has been received.

While GRC may have lower communication overhead than WRC, it has greater computational and space requirements. If no underflow indirection is needed, its communication overhead is the same as WRC, namely, one acknowledged message for each copy of a remote reference. Just as WRC is susceptible to *underflow*, GRC ledgers can *overflow*. Goldberg [1989] suggests using indirection to solve this problem. Unlike WRC, the indirection will always be on the same site as the reference, thus adding no extra communication overhead.

3.1.4 Indirect Reference Count

Indirect reference counting, IRC [Ichisugi and Yonezawa 1990; Rudalics 1990; Piquer 1991], provides a solution to the problem of underflow in GRC. IRC replaces the generation by a reference, *parent*, to the source of the copy. Each reference is a triplet (Figure 20).

The parent field is used to maintain an *inverted diffusion tree* of duplicated references (Figure 21). The depth of a reference in the tree is the generation. The number of vertices in the diffusion tree equals the total number of references to the root. The *indirect reference count* counts the number of children of each vertex in the diffusion tree and corresponds to the copy count of GRC. While WRC and GRC can be seen to divide state information between the cell and the reference, for IRC all state information is maintained by the reference.

When a new cell is created (Figure 22) the initial reference becomes the root of a diffusion tree. When a reference is duplicated, the copy count is incremented and the new reference linked into the diffusion tree (Figure 23). Remote duplication requires just one message.

If a copy is deleted, the parent copy count is decremented, resulting in (again) Figure 22. For a remote reference this requires one message. While references in the body of the diffusion tree can be excised, only cells that are leaves of the diffusion tree (zero copy count) can be reclaimed, Figure 24. Thus, like WRC, IRC can accumulate large amounts of floating garbage. However, an excised reference can be restored before its copy count reaches zero. This can be compared with deferred reclamation (Sect. 2.2.2).

3.1.5 Indirect Reference Listing (IRL)

Piquer [1991] suggests that the space overhead of IRC is acceptable if it is used only for remote references. Following Fowler [1986], Shapiro et al. [1990] replace the copy count of IRC by a list of references where duplicates have been diffused. A reference is interpreted as a shortcut to the root of the diffusion tree. The additional space overhead of indirect reference listing, IRL, is justified by simpler management of message loss, duplication, and latency. Plainfossé and Shapiro [1992] describe a prototype implementation for Lisp; Birrell et al. [1993] describe an implementation for remote objects in Modula3. Site failure is detected by regular pinging. The import records of sites that do not promptly acknowledge a ping are unsoundly deleted.

3.1.6 Trial Deletion

Vestal [1987] proposes trial deletion to remove cycles of garbage with indirect identification. The algorithm is seeded with some cell suspected of being part of a dead cycle. The method consists of hypothetical recursive deletion of the seed and its referents and checking if this brings all the counts in the subgraph to zero. A drawback of trial dele-

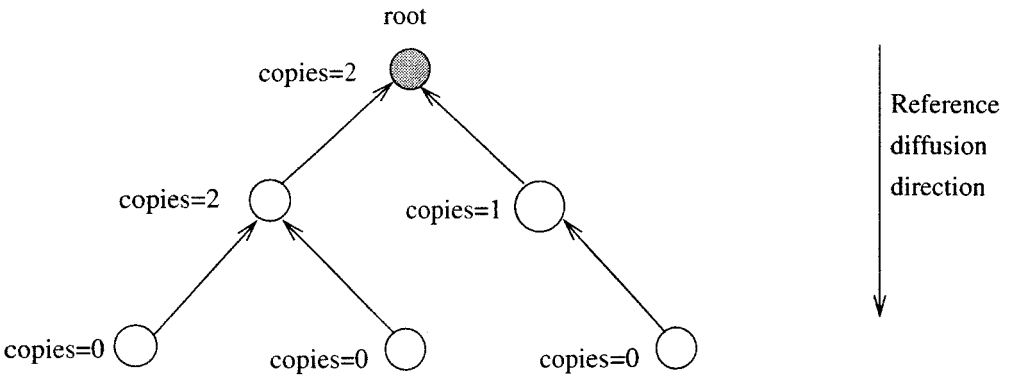

Figure 21. An inverted diffusion tree.

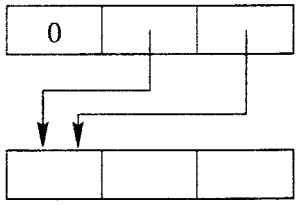

Figure 22. Allocating a new cell in IRC.

tion is that, like recursive freeing, it is unbounded. Furthermore, seeds are chosen heuristically, so a bad choice can lead to wasted effort. The scheme can be seen as a generalization of Brownbridge [1985] (Section 2.2.5), where the strong counter is used as a heuristic. Trial deletion has the same problem as Brownbridge's scheme, mutually referencing cycles.

3.2 Indirect Identification of Distributed Garbage

Mohammed-Ali [1984] describes a number of variations of distributed mark-scan collectors. The simplest, sequential mark-scan (referred to as [Mohammed-Ali 1984a] in Table I), is not a serious contender but provides a straw man to compare improved versions.

3.2.1 Distributed Mark-Scan

Mohammed-Ali's [1984a] sequential mark-scan requires mutation to be suspended during garbage collection on all sites. Across the sites, the processes have the following synchronous behavior:

```
site A:
 M M M. . . | I I I. . . |R R R. . . | M M M. . .
      :   : |    :      |   :       |   :
site Z:
 M M M. . . | I I I. . . | R R R. . .| M M M. . .
```

The vertical bars indicate global synchronization points.

Any site that has exhausted its free store can initiate garbage collection by sending a request to some master site. This master may be designated statically or determined dynamically. If dynamic, the initiating site can be the master but arbitration is necessary if more than one site simultaneously needs to collect garbage. The master sends a command to each site to suspend mutation. The master waits for each site to report all messages in transit have been received and acted upon. The master then directs each site to start the identification (marking) process. Mohammed-Ali remarks that although fast, parallel breadth-first marking has high and unpredictable space requirements that make it impractical. The alternative, sequential depth-first marking, requires much less space. The master waits until each site reports all messages in transit have been received and acted upon and local marking is complete. The master then directs each site to perform a local reclamation (scan). When all sites report

messages in transit and reclamation is complete, the master directs each to resume mutation.

Similar schemes have been implemented in Berkeley Smalltalk [Schelvis and Bledoeg 1988] and the Emerald object system [Black et al. 1987; Jul et al. 1988]. The problem with such schemes is that without global termination there can be interference between mutation, identification, and reclamation on different sites. Synchronization is achieved by the master waiting for all sites to report phase completion. A major problem is that a slow site cannot be distinguished from failed site. Mohammed-Ali [1984] observes that while only one site needs to collect garbage, the others are compelled to do so. Furthermore, forcing sites to synchronize requires all but one site to be idle waiting for the last to complete (usually the one that initiates the collection).

3.2.2 Distributed Concurrent Mark-Scan

Dijkstra et al.'s [1978] concurrent mark-scan, which allows mutation to continue while collecting garbage, seems better suited to multiple mutators. One of the first distributed adaptations was the marking-tree collector [Hudak and Keller 1982]. In this variation, there is assumed to be a single root of the whole distributed computation graph. (This is the case for graph reduction of functional languages.) Identification and mutation take place concurrently across sites:

```
site A   (mutator): M M M. . .
site A (collector): I I I. . . | R R R. . . | I I I. . .
                    :   :          :          :
site Z   (mutator): M M M. . .
site Z (collector): I I I. . . | R R R. . . | I I I. . .
```

Each recursive mark step in Dijkstra et al.'s scheme is replaced by a mark task. Each site maintains two task queues: one for mutation operations and one for collection operations. Termination of the mark phase is detected by each mark task of a leaf node spawning a task that is propagated upward in the mark tree. Tricolor marking, as in [Dijkstra et al. 1978], is used to record the identification state of a cell but the interpretation of the colors is subtly different. A *white* cell is one to which identification has not yet propagated. Initially, all cells are white and after marking is complete, white cells identify garbage. A *gray* cell is one to which marking has propagated and from which a mark task has been spawned for each of its referents. A *black* cell is of one of two types: a newly allocated cell or a previously gray cell for which all of its spawned marking tasks have terminated.

Mutator tasks and identifier tasks compete to modify cells. Each task has to lock all cells it intends to modify to prevent lost updates. At the end of the marking (identification) phase, white cells are garbage and all tasks referencing white cells are garbage. The scan (reclamation) phase first terminates all redundant tasks and then collects all white cells. No locks are necessary in the reclamation phase because there is no contention with the mutator. The algorithm is concurrent, but the phases of identification and reclamation must be globally synchronized across sites.

Similar mark-scan collectors are described by Augusteijn [1987], Vestal [1987], and Derbyshire [1990]. Augusteijn [1987] describes the collector for the object-oriented language POOL-T. Communication between objects is made using a rendezvous protocol with a sender suspending until it receives a reply. A central synchronization object is introduced to establish and maintain global invariants.

As in the nondistributed version of concurrent mark-scan, the collector operates even when there is no garbage to

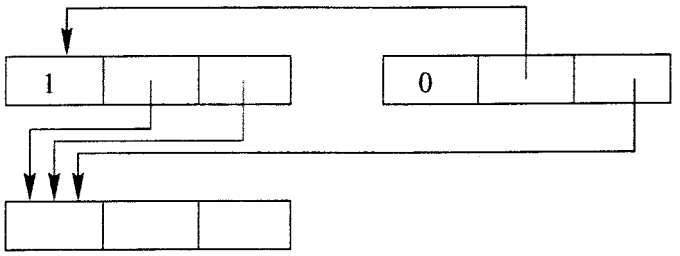

Figure 23. Duplicating a reference in IRC.

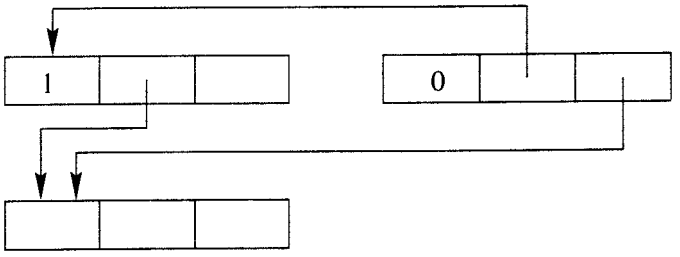

Figure 24. Deleting a reference in IRC.

collect. Propagating gray marks causes a combinatorial avalanche of marking tasks. Because collectors generally do not batch remote tasks, this imposes high message traffic. If batched, space needed for storing these requests cannot be determined in advance.

3.2.3 Central Coordination of Local Collection

Mohammed-Ali [1984b] proposes that local garbage collection might free enough space for a site to continue without requiring a global collection. Adapting the area concept of Bishop [1977] developed for large (virtual) address spaces, each site is provided with an *Import Record Table*, IRT, which holds all import records. The IRT is used as additional roots for local garbage collection. Grouping the export records in a table, the *Export Record Table*, ERT (Figure 25) restores the symmetry.

Liskov and Ladin [1986] use the client-server model to extend local mark-scan with centralized identification of parts of the graph between import and export records. Each local collector informs a server about the paths it knows of. Local collectors query the centralized service for the current IRT. Dead intersite cycles are detected by the centralized service from the paths advised by the local collectors. The centralized service builds a graph of intersite references and detects dead cycles with a standard collector. While logically centralized, Liskov and Ladin's [1986] scheme is physically replicated to achieve high availability. A client communicates with a single replica; replicas stay up-to-date by exchanging background "gossip" messages.

By means of a counterexample, Rudalics [1990] demonstrates that the Liskov and Ladin [1986] scheme is unsound. A scenario can occur when a cell, such as b in Figure 26, has more than one reference to it. If the local marker on site C traverses cell d before a, cell b will only be traversed once. At the end of collection, site C only informs the server of the path between c and d and not the one between a and c. The central server unsoundly concludes that d and c are garbage. Rudalics proposes two computationally expensive solutions to overcome the problem.

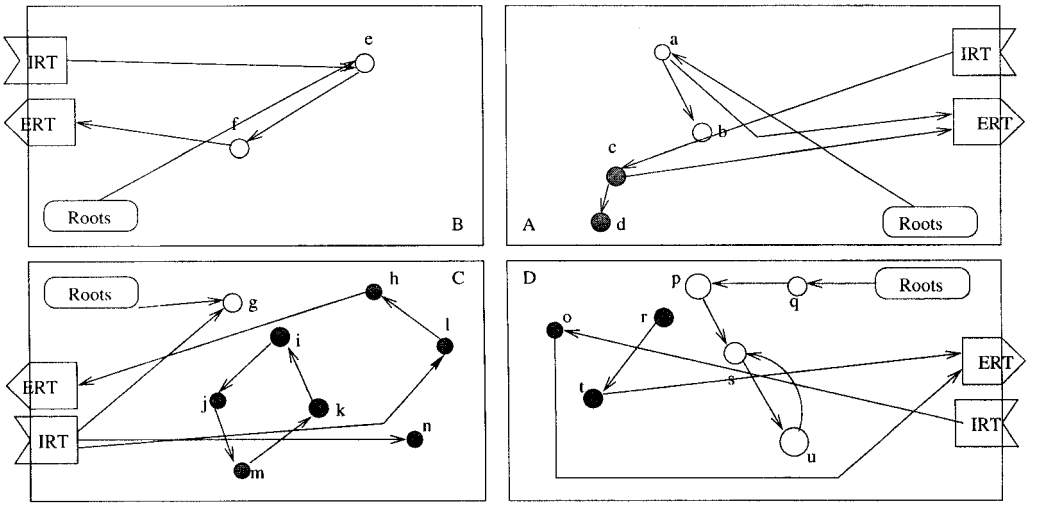

Figure 25. Distributed GC by local collection.

3.2.4 Cell Migration

As with generation scavengers, Section 2.3.5, local collection does not remove garbage subgraphs that cross site boundaries. While generation scavengers give a temporal segregation of cells, distributed systems have a spatial segregation of cells. Following Bishop [1977], El-Habbash et al. [1990] propose migrating cells so that intersite cycles can be reclaimed by local collection (Figure 27).

El-Habbash et al. introduce a *PrivateTable*, PT, to provide complete location-independent addressing. Cells are partitioned into *locality clusters*, each with its own IRT, ERT, and PT. A cluster is a logical partition of cells in contrast to a physical partition, a site. Ideally, a cluster has many more intracluster references than intercluster references. The division of cells into locality clusters can be compared with generation scavengers (Section 2.3.5), a division of cells into temporal clusters. Remotely referenced cells in a locality cluster are given unique *public identifiers*, PIDs. Cells that are only referenced locally are not known outside the cluster and are given *local identifiers*, LIDs. The LIDs comprise entries in PT. A major problem with this scheme is generating unique PIDs, particularly in a very large network.

Clusters are the unit of management for El-Habbash et al. [1990]. The objective of management is to increase the locality of reference of a cluster. Garbage collection is a by-product of increasing locality. To increase locality, cells may migrate from cluster to cluster via *archive* clusters. Subgraphs that are only reachable from IRT are transferred to an archive cluster. When an archived cell is accessed from another cluster, that cell and its subgraph are moved to the referencing cluster. Cells that are not accessed remain in the archive. Starting from the roots of a cluster and traversing the subgraphs rooted in them, any cells encountered remain in the cluster. Cells that are not reachable from the roots are moved to an archive. The cells that are not reachable from any remote cells (roots or nonroots) in the cluster are garbage.

The El-Habbash et al. [1990] collector is intended for use in persistent environments such as Smalltalk. A similar scheme for persistent store is described by Moss [1990] for the Mneme project. Moss equates a persistent store with a

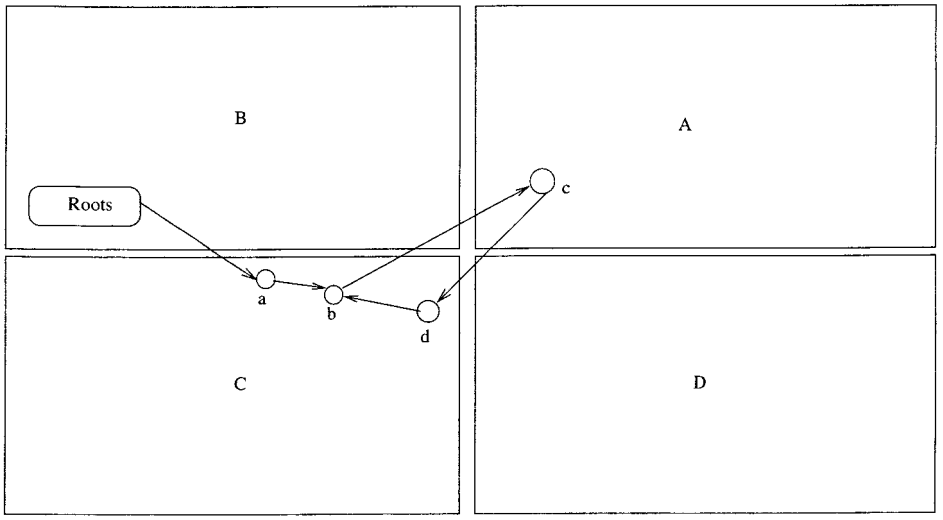

Figure 26. Rudalic's counterexample.

database, but cell retention is based on *reachability* (in garbage collection) as opposed to *explicit deletion* (in the database sense).

One problem with cell migration as a means of collecting intercluster cycles is thrashing. Migration can lead to a scenario (Figure 27a) where *a* is migrated to D, *d* to C and *c* to A. El-Habbash et al. [1990] propose a total ordering on clusters (such as name ordering) to avoid thrashing. A cell can only migrate to an inferior cluster. A more serious problem is that archival garbage collection is controlled by setting time limits on access. With slow sites this will lead to unsoundness.

3.2.5 Pipelined Local Collections

As with generation scavenging, copying large cells is expensive. Mohammed-Ali [1984c] suggests that garbage that crosses site boundaries can be collected if at the end of a local collection a site informs other sites of the export records it holds. A message containing a reference may be in transit when a local collection is invoked. This can lead to a cell not being identified as live. Mohammed-Ali proposes each site be provided with a temporary *Transport Table*, TT, which records in-transit references. These are moved to IRT or ERT when they are acknowledged.

Rudalics [1986] describes a distributed collector adapted from Baker's [1978] incremental scavenger. Each site has two semi-spaces used for garbage-collecting local cells. The upper part of each semi-space is used for export records. The import records are linked in either of three lists. The first two act as semi-spaces for external references, while the third corresponds to Mohammed-Ali's TT. As with single-address-space generation scavengers, neither Rudalics nor Mohammed Ali's scheme is able to identify cycles of garbage that span more than one site.

Hughes [1985] describes a way of pipelining local collections that can detect intersite cycles of garbage. This is achieved by propagating timestamps in place of marks. Import and export records are initialized with a global clock [Lamport 1978]. Necessary conditions for a global clock are that the underlying message-passing system guarantees that messages are not lost, duplicated, and arrive in mutual causal order (FOFI). An export record reachable from a local root is marked with

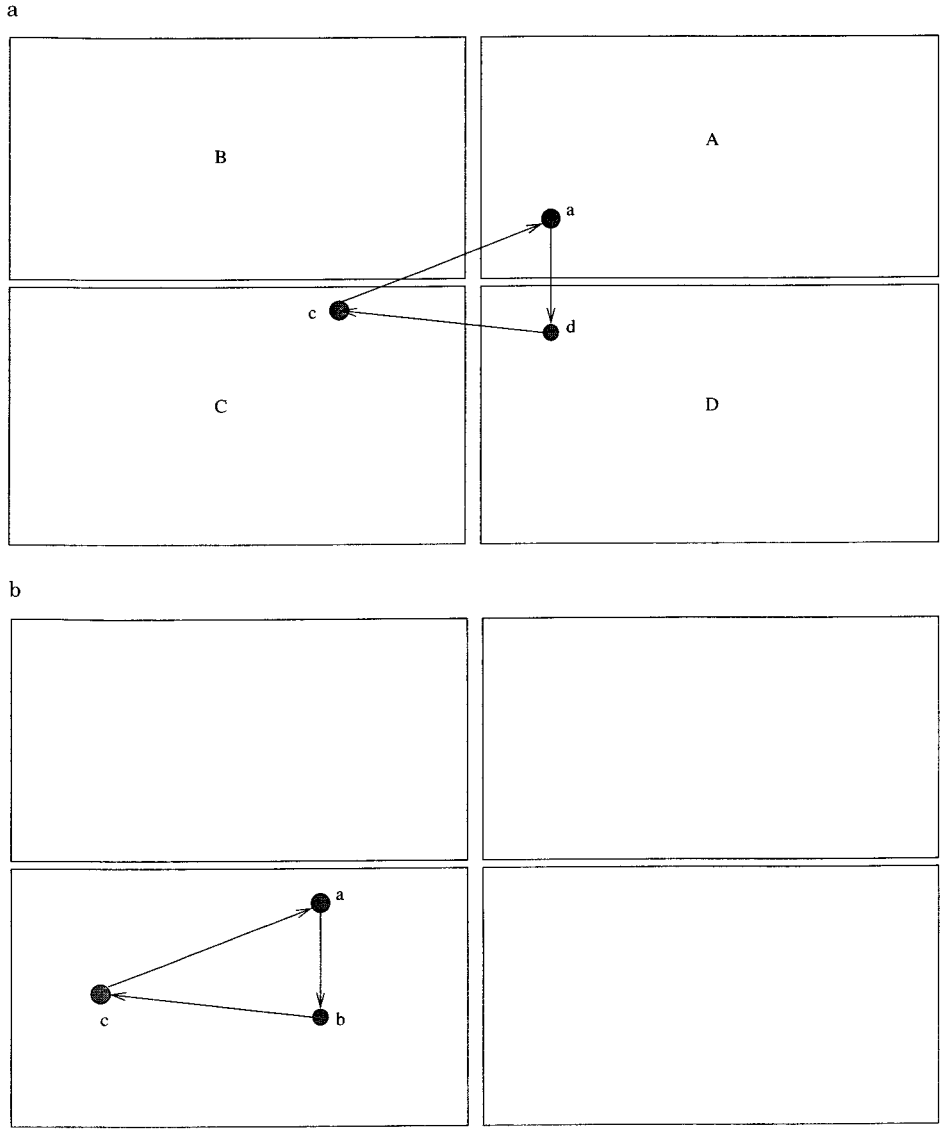

Figure 27. Cell migration: (a) intersite cycle of garbage; (b) migration of two cells.

the time at which the local marking phase started. An export record traced from an import record adopts the timestamp of the traced import record.

At the end of a local collection, export record timestamps are sent to corresponding import records. If the timestamp of the export record is greater than the import record, the import record timestamp is updated. When receipt of all such messages has been acknowledged, the local clock is incremented to the greatest propagated timestamp. In this way, the timestamp of a dead import or export record remains constant while live ones increase.

Import records that carry a timestamp less than some threshold are reclaimed. The threshold is the least local timestamp. Hughes determines the threshold using Rana's [1983] termination algorithm. A problem is that a slow

site unwilling to initiate a local collection will leave the threshold at the initial value. This is the case even when a slow site does not hold any remote references.

3.3 Distributed Hybrid Collectors

While WRC, GRC, and IRC elide race conditions (and at the same time reduce the communication overhead), they suffer the same problem as their single-address-space progenitors: memory leaks due to cycles of garbage. Worse still, the cycles may be intersite, such as o-c-h in Figure 5.

Lins and Jones [1991] give an adaptation of the cyclic reference counting schemes of Martinez et al. [1990] and Lins [1990] to the distributed environment. The algorithm combines WRC (Section 3.1.2) with Lins' [1990] local mark-scan (Section 2.4). The algorithm has the same problem as its progenitor: the need to perform a local mark-scan every time a reference to a shared subgraph is deleted. Successive attempts to address these problems are presented by Jones and Lins [1992, 1993]. As admitted by the authors, the scheme has four deficiencies. The first is that reclamation of garbage cycles may be delayed indefinitely. Second, the scheme has higher storage overheads than WRC. Third, the three phases of garbage collection require termination detection. Last, unlike Hudak and Keller [1982], the scheme cannot detect nor remove tasks that become redundant due to garbage collection. Dehne and Lins [1994] attempt to address these problems. The scheme allows sites to perform local mark-scan without the need to synchronize the phases either on a single site or across sites. This requires six colors.

Piquer [1991] suggests that the space overhead of IRC (Section 3.1.4) is acceptable if it is only used for remote references. Intersite cycles can be collected by cell migration and local direct identification collectors. Inverted diffusion trees (Section 3.1.4) can easily support cell migration with an overhead of only one decrement message between source and destination sites. The migration of a cell requires a change of the root in the diffusion tree (Figure 28). This operation is trivial, as the old root is known: the new root is extracted from the tree and the old root added as a child of the new root. The extraction costs one "decrement" message and the addition is done locally at the respective sites (the new and old roots). Like El Habbash [1992], a total ordering on sites will avoid thrashing of migration. Cell migration can, however, lead to unsoundness if references to the old location of a cell are in transit while a cell emigrates.

Shapiro et al. [1992a] describe an RPC (remote procedure call) implementation of a hybrid collector that uses IRL (Section 3.1.5) for remote references and local tracing collectors. The garbage collector is tightly coupled with an object management system. The cell finder RPC handles cell deletion and site crashes. When given an indirect (parent) reference, the procedure locates the cell referred to. In this way, the reference field is completed lazily. Other RPCs include reference-sending, cell migration, cycle-detection, and abnormal termination.

As in Hughes [1985] (Section 3.2.5), messages in Shapiro et al. [1992a] are time-stamped by a local monotonic (increasing) clock. Each IRT entry is stamped with the clock value of the last corresponding message sent. Unlike IRC (Section 3.1.4), remote references to the same cell each have separate import records. Each site maintains a vector of highest time-stamped messages received from other sites. Unlike Hughes [1990], clocks on different sites need not be synchronized; a total count of transmitted (mutator and control) messages is sufficient for the purpose. To detect duplicated or lost messages, a list of export records is sent to the site referencing them.

When a mutator exports a reference to another site, it is first added to the

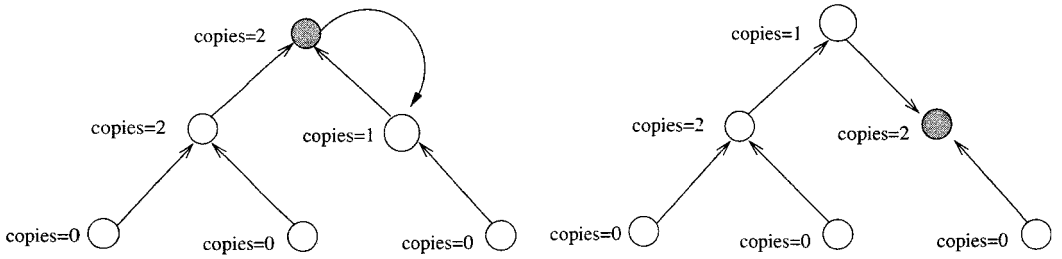

Figure 28. Cell migration in an inverted diffusion tree.

local IRT. Both the IRT and the ERT are incomplete (overestimates). Local garbage collection proceeds from both local roots and the IRT. Shapiro et al. use two colors in local marking. A cell accessible from the local root is marked *green*. A cell accessible only from the IRT is marked *red*. The collector removes garbage entries in the ERT sending update information to the IRT entries in appropriate sites. This, in turn, allows previously referenced IRT entries to be collected. Unlike the distributed concurrent mark-scan collectors (Section 3.2.2) the interface between the global collector and other components (i.e., the mutator and the cell finder) is limited to just the IRT and ERT. Updates to IRT and ERT can occur in parallel with other activities.

In a prototype distributed Smalltalk-80 system, Bennett [1987] describes a scheme that pipelines local deferred-reference-counting collectors through global-reference-counting and mark-scan collectors. The global collectors rely on the local collection to enumerate the export records, called *proxy cells*. Bennett's fast global reference counter relies on cells in alternate collection cycles being distinguishable. Each IRT entry has a flag that identifies import records created since the start of a collection. This is similar to the gray color of Dijkstra et al. [1978]. During a local collection, each site enumerates its export records and for each sends a message that increases the *external* reference count in its corresponding IRT entry. After this *marking* phase, live remotely referenced cells have a nonzero external reference count. Each site then scans its IRT and removes those cells with a zero external reference count that were in existence before the start of the cycle (i.e., not gray). Any referents not referenced locally are reclaimed by the site's local collector.

Bennett's fast collector cannot detect intersite cycles. The second, slower collector is a mark-scan algorithm. The marking phase proceeds from those cells in the IRT that also have local references (determined by the local reference count). References are followed to export records and messages are sent to the remote sites to continue the trace remotely. At the end of the phase, intersite cycles have not been marked and can be removed from the IRT.

Lang et al. [1992] describes a scheme that pipelines local tracing collectors through a global reference-counting collector. When an ERT entry is reclaimed, a decrement message is sent to the site hosting the corresponding import record. If the decrement action brings its counter to zero, the IRT entry is reclaimed. This is the only mechanism for reclaiming IRT entries. It is sound, since sites that are down do not send decrement messages.

Lang et al.'s [1992] sites are organized into *groups* that cooperate to remove garbage cycles that span their members. The groups can be hierarchical with the largest containing all the sites. Collection begins with group establishment. The composition of a group

can be determined statically or dynamically, but is independent of collection. A site would contemplate group collection only if local collection does not free enough space for the mutator to continue. When a site fails to cooperate, the group is reorganized to exclude it and collection continues without losing work already done. Messages with acknowledgments and time-outs are used to detect noncooperating sites. Multiple overlapping group collections can be simultaneously active if each group associates a unique identifier to a collection.

A group cooperates to collect their ERTs by direct identification. Local garbage collection is used to transmit marks from IRTs to ERTs. For each group collection, IRT and ERT entries have a mark that is local to the group. IRT entries may be marked *soft* or *hard*. The ERT entries may be marked *hard*, *soft*, or *none*. Effectively, an IRT entry is marked *hard* if it is needed outside the group or is accessible from a root of a site in the group. It is marked *soft* if it is referenced only from another member of the group.

Local garbage collection has two marking phases. In the first phase, the initial marks of IRT entries are determined from the reference count and references from members of the group (after Christopher [1984]). All marks on ERT entries are reset to *none*. Marking proceeds from both local roots and hard IRT entries. Any ERT entry reached by this tracing is marked *hard*. In the second phase, tracing starts from the *soft* IRT entries. Any ERT entry reached is marked *soft* if it is not already marked *hard*.

After a local garbage collection, the ERT entries that are marked *none* are garbage. They can be reclaimed while sending decrement messages to the IRT entries they reference. ERT entries marked *hard* (and the IRT they reference) are reachable either from a *hard* IRT entry or from a local root. When an ERT entry is known to be *hard*, its mark has to be propagated to the IRT entry it references (if it is in the group). When a new remote reference is created, the associated IRT entry is marked emphhard (as in the distributed version of concurrent mark-scan (Section 3.2.2)) since it is necessarily accessible from a root.

After n such marking cycles, where n is the number of sites a cycle spans, all *hard* IRT entries are directly or indirectly accessible from a root or from a site outside the group. IRT entries marked *soft* are inaccessible, and can thus be safely reclaimed. Such *soft* IRT entries are set to reference *nil* rather than a local cell. The unreachable offspring of these IRT entries will be reclaimed by the next local GC. Similarly, the ERT entries that were kept alive exclusively by these entries will be reclaimed by the next local GC. The reclamation of such an ERT entry causes the sending of a decrement message to the IRT entry it references. In the case of dead cycles, dead IRT entries in the cycle eventually receive decrement messages from all the dead ERT entries that reference them. Hence their reference counts decrease to zero and they are eventually reclaimed by the reference-counting mechanism. This protocol is conservative as it achieves a deferred reclamation instead of a synchronized deletion of dead cells.

A problem with the Lang et al. collector is the propagation of ERT entries to IRT entries after a local collection. It is only safe to assume that soft ERT entries are garbage after all hard marking messages have been received—thus it requires a group termination-detection algorithm. If the group consists of all the sites, the problem is similar to Hughes [1985].

4. CONCLUSIONS

The problems of distributed garbage collection appear to be the same as those of single-computer collection: completeness and soundness. Soundness cannot be compromised. According to Juul and Jul [1992], the broad spectrum of collec-

tors is explained by the tradeoff between completeness, the ability to collect all garbage, and expediency, the ability to satisfy mutator allocation requests unobtrusively. The problems of concurrency have been met in concurrent mark-scan collectors (Section 2.3.2). The problems of localization have been met in scavenging collectors (Section 2.3.3) and large address space collectors [Bishop 1977]. But the lack of synchrony of distributed systems poses questions of what completeness and soundness mean. Garbage collection, in fact, is a microcosm that exhibits all the problems of distribution [Vestal 1987].

4.1 Overview

The early distributed garbage collectors were, naturally enough, based on single-address-space collectors. Single-address-space collectors were surveyed and classified in Section 2; this classification is used to explore the issues of distribution in Section 3.

A straightforward attempt to use reference counting in distributed environments (Section 3.1.1) exposes the problem of indeterministic latency. A succession of improvements (Sections 3.1.2, 3.1.3, 3.1.4) elide the problem by transferring successively more state information to the reference. At the same time, these solutions reduce the communication overhead.

All indirect identification schemes fail to solve the main difficulty with reference counting, memory leaks due to cyclic structures. The concurrent mark-scan collector (Section 2.3.2) initially seems appropriate for distribution, but suffers from avalanches of marking messages. A critique of global stop-the-world synchronization mark-scan suggests pipelining local collections to collect garbage subgraphs that cross site boundaries (Section 3.2.5).

As demonstrated by Rudalics [1990], Liskov and Ladin's [1986] centralized client-server collector permits unsound scenario (Section 3.2.3). The error in Liskov and Ladin was corrected by Laden and Liskov [1992] using a global clock after Hughes [1985]. The termination protocol is not required because the central service determines the threshold. Noncooperating sites still suppress the threshold value. As noted by Shapiro et al. [1994], an unsound scenario was implicitly and wrongly assumed not to occur in Shapiro et al. [1992a]. Shapiro et al. [1992b] corrects the error. The origin of these anomalies and the race conditions of reference counting is indeterministic message latency.

Because in a distributed system latency is indeterministic, time is relativistic [Babaoglu and Marzullo 1993]. Local task scheduling provides only a partial order on events. The partial order can be extended to a total causal order using local vector clocks or a global clock. However, the total order is not unique. As there is no unique total order, so there is no unique meaning of completeness and soundness. Hughes [1985] (Section 3.2.5) uses a global clock for termination detection. Tel and Mattern [1993] show that IRC (Section 3.1.4) is equivalent to Dijkstra and Scholten's [1989] termination-detection algorithm. A global clock requires that the transport layer provide mutual causal order [Babaoglu and Marzullo 1993]. Unlike Hughes [1985], Shapiro et al. [1992b] do not assume that the transport layer provides a reliable message-passing system. Instead, garbage collection and reliable message-passing are integrated using vector clocks.

The current implementation of Java uses a distributed collection based on Birrell et al.'s [1993] collector for Modula-3, Section 3.1.5. Unlike Shapiro et al. [1992a], Java's remote method invocation is built directly on sockets. Sockets are an API (application programming interface) to TCP. The architecture adds two further layers on top of TCP [Sun 1996] (Figure 29). The migration layer provides facilities for cell (object) migration. As Smalltalk was a major driving force in the development of single-address-space garbage collectors,

Table 1. Distributed Direct Identification Schemes

Author(s) and year of publication	Pause	Space Overhead	Comm Overhead	Synchronization Overhead	Local Collection	Removal of intersite cycles	Notes
Lermen and Maurer [1986]	Low (recursive freeing)	Ref count	3*(rem refs copied)	Ref deletion	MI phase	No	
Thomas [1981]; Watson and Watson [1987]; Bevan [1987]	Low (recursive freeing)	Count on cell & weight on reference	rem refs + rem indirections	None	MI phase	No	Underflow. Indirections on different sites.
Goldberg [1989]	Low (recursive freeing)	Ledger on cell, generation & copy count on reference	rem refs	None	MI phase	No	Overflow. Indirections on same site.
Piquer [1991]	Low (recursive freeing)	Copy count and parent for each ref.	rem refs	None	MI phase	No	No indirections.

Figure 29. Migration and remote reference layers.

it is likely that Java will play a similar role for distributed garbage collection.

The problem of slow and noncooperating sites is partially solved by Lang et al., Section 3.3, who exclude slow sites from a group collection without any loss of work. This grouping strategy is similar to those used by single-address-space-generation scavengers.

4.2 Collector Comparison

Numerous schemes have been proposed for distributed garbage collection, but little has been done in the way of systematic comparison. Zorn [1989] categorizes the evaluation of garbage collectors into language-specific implementations, analytic studies and simulation. Previous evaluation of single-site garbage collectors boasts representatives of all three classes. Ungar [1984; 1987] describes several single-address-space implementations of generation-scavenging garbage collectors (Section 2.3.5 for Smalltalk). His performance metrics include CPU overhead, pause length, peak main memory use, and backing store access. Ungar reported the average CPU overhead as 1.5% and average pause times of 150ms every 16 seconds. These measurements were on Berkeley

Table 2. Distributed Indirect Identification Schemes

Author(s) and year of publication	Pause	Space Overhead	Comm Overhead	Synchronization Overhead	Local Collection	Removal of intersite cycles	Notes
Mohammed-Ali [1984a]; Schelvis & Bledoeg [1988]	High (stop mutation)	Mark	High	2*termination detection. Centralized.	No (network wide marking)	Yes	Basis for comparison.
Hudak & Keller [1982]	Low (concurrent mutator)	Mark	High	Termination detection + local locks. Centralized.	No (network wide marking)	Yes	Mutation and collection in parallel.
Ladin & Liskov [1992]	Medium (collecting local space)	Intersite graph	High	Termination detection. Centralized but replicated.	Yes (site)	Yes (centralized)	Client/server (replicated).
Mohammed-Ali [1984b]	Medium	Mark	Low (remote refs)	None	Yes (site)	No	No collection of garbage which crosses sites.
El-Habbash, Harris & Horn [1990]	Medium	Mark and tables	Low	None	Yes (cluster)	Yes (migration)	Object-oriented distributed persistent environment.
Mohammed-Ali [1984c]	Medium	Mark and TT table	Low (remote refs)	Ack messages in transit	Yes (site)	No	Pipelines local collections
Hughes [1985]	Medium	timestamps	Low	Termination detection	Yes (site)	Yes	Pipelines local collections.

Table 3. Distributed Hybrid Schemes

Author(s) and year of publication	Pause	Space Overhead	Comm Overhead	Synchronization Overhead	Local Collection	Removal of intersite cycles	Notes
Dehne and Lins [1994]	Low (recursive freeing)	Ref count, color, control heap & 2 queues	3*(rem refs copied) + 2*(suspect cyclic rem refs)	None (doubtful)	MI phase	Yes (lazy mark-scan)	Addresses the issue of synchronization
Shapiro, Dickman & Plainfossé [1992b]	Medium (site collection)	Tables	Low	None	Yes (site)	Yes (migration)	Low-level distributed object support system.
Bennett [1987]	Medium (tracing RCT)	Flagged entry record (RCT)	rem refs	None	Yes (site)	Not really	Uses a second, mark-scan, collector to remove cycles.
Lang, Queinnec & Piquer [1992]	Medium (site collection)	Tables	Medium (marking entry items)	group termination detection.	Yes (site)	Yes	A site not responding in a timely manner is excluded from a collection without loss of work.

Smalltalk, an implementation of the Smalltalk-80 system for Sun workstations.

Evaluation of distributed garbage collection is much more difficult, and only language-specific comparisons have been reported so far. A number of authors have evaluated collectors for distributed versions of Smalltalk. Bennett [1987] (Section 3.3) adds a global garbage collector to multiple Smalltalk-80 incarnations with their own local collector. His performance metric was communication overhead. Bennett reported that on average remote messages are slower than local messages by a factor of 1000. The measured system consisted of two Sun-2 diskless workstations connected by a 10 megabit/second Ethernet. Schelvis and Bledoeg [1988] also evaluated collectors for distributed Smalltalk implemented on a network of Sun workstations running Berkeley UNIX. These included a conservative generation-scavenging collector. Schelvis and Bledoeg conclude that remote send is approximately 500 times slower than local send. In addition, the cost of remote computation is reported at about 45% slower than local computation.

Plainfossé and Shapiro [1992] report an implementation of Shapiro et al. [1990], SGP, for Lisp. Measurements for the number of messages and the CPU overhead are given. They compared SGP with the indirect reference count collector (IRC, Section 3.1.4). The CPU overhead shows that SGP is on average 20% slower than with no garbage collection and 10% slower than IRC. This was contrary to the expectation that distributed garbage collection is communication-bound. The number of control messages sent in the SGP protocol is 70–80% lower than the IRC protocol. Plainfossé and Shapiro attribute the difference to the "buffering" strategy of the SGP protocol. These measurements were taken on a Parsytec board composed of transputers (T800) with one megabyte of memory each, hosted by a Sun.

Distributed garbage collection is com-

plex and a realistic empirical comparison of distributed collectors has been lacking. The Ph.D. thesis of one of the authors [Abdullahi 1995] addresses this problem. It appears that the features of distributed garbage collectors are delicately balanced. Improvements in one aspect are made to the detriment of another. These results will be reported elsewhere [Abdullahi and Ringwood 1996]. In lieu of quantitative information, Tables I, II, and III give a qualitative comparison of representative distributed collectors described in this review. Where qualification is appropriate and known, as in pause, space and communication overhead, a rank of low, medium, and high is given. These are qualitative and relative terms. An order of magnitude or further explanation, where available, is given; otherwise the source of such overhead is given.

ACKNOWLEDGMENT

The authors are grateful to the referees for pointing out omissions and misconceptions in the draft. Referees' requests for clarification prompted some of the rationalization of the ontology. Both authors contend that serious omissions and misconceptions that still remain are entirely the fault of the other author.

BIBLIOGRAPHY AND REFERENCES

ABDULLAHI, S. E. 1992. Managing computer memory: Dynamic allocation and deallocation strategies. In *Proceedings of the 2nd Conference on Information Technology and its Applications* (Leicester UK, Dec. 19–20), 25–40.

—. 1994. Recycling garbage. In *Proceedings of the 3rd Conference on Information Technology and its Applications* (Leicester, UK, April 2–3), 192–197.

—. 1995. Empirical studies of distributed garbage collection. Ph.D. thesis, Dec. 1995. Univ. of London.

ABDULLAHI, S. E. AND EDEMENANG, E. J. A. 1993. A comparative study of dynamic memory management techniques. *Advances in Model. Anal. 15,* 2, 17–31.

ABDULLAHI, S. E. AND RINGWOOD, G. A. 1996. Empirical studies of distributed garbage collection parts I, II, and III. TR, Dept. of Computer Science, QMW College, Univ. of London.

ABDULLAHI, S. E., MIRANDA, E. E., AND RINGWOOD, G. A. 1992. Collection schemes for distributed garbage. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France). LNCS 637, Springer-Verlag, 43–81.

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, Cambridge, MA.

ALMES, G., BORNING, A., AND MESSINGER, E. 1983. Implementing a Smalltalk-80 system on the Intel 432: A feasibility study. In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, 175–187.

AMSALEG, L., GRUBER, O., AND FRANKLIN, M. 1995. Efficient incremental garbage collection for workstation-server database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases* (Zurich, Switzerland).

ANDREWS, G. R. 1991. *Concurrent Programming—Principles and Practice.* Benjamin/Cummings.

APPLEBY, K., CARLSSON, M., HARIDI, S., AND SAHLIN, D. 1983. Garbage collection for Prolog based on WAM. *Commun. ACM 31,* 6, 719–741.

ARNBORG, S. 1974. Optimal memory management in a system with garbage collection. *BIT 14,* 375–381.

ARVIND, V. K. AND IANNUCCI, R. A. 1987. Two fundamental issues in multiprogramming. In *Proceedings of the Conference on Parallel Processing in Science and Engineering* (Bonn-Bad Godesberg), 61–68.

ATKINSON, M. P., BAILEY, P. J., CHISHOLM, K. J., COCKSHOTT, P. W., AND MORRISON, R. 1983. An approach to persistent programming. *Computer J. 26,* 4, 360–365.

AUGUSTEIJN, L. 1987. Garbage collection in a distributed environment. In *PARLE'87—Parallel Architectures and Languages Europe.* LNCS 259, Springer-Verlag, 75–93.

BABAOGLU, O. AND MARZULLO, K. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems,* Addison-Wesley.

BADEN, S. B. 1983. Low-overhead storage reclamation in the Smalltalk-80 virtual machine. In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, 331–342.

BAECKER, H. D. 1972. Garbage collection for virtual memory computer systems. *Commun. ACM 15,* 11, 981–986.

BAKER, H. G. 1978. List processing in real-time on a serial computer. *Commun. ACM 21,* 4, 280–294.

—. 1992. The treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Not. 27,* 3 (March), 66–70.

BAL, H. 1990. *Programming Distributed Systems,* Prentice Hall, Englewood Cliffs, NJ.

BALLARD, S. AND SHIRRON, S. 1983. The design

and implementation of VAX/Smalltalk-80. In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, 127–150.

BARACH, D. R., TAENZER, D. H., AND WELLS, R. E. 1982. A technique for finding storage allocation errors in C-language programs. *ACM SIGPLAN Not. 17,* 5 (March), 16–23.

BARTBARA, L. AND RIRKA, L. 1986. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing* (Aug.), ACM, 29–39.

BARTLETT, J. F. 1990. A generational, compacting garbage collector for C++. Position paper, ECOOP/OOPSLA '90 Workshop on Garbage Collection.

BATES, R. L., DYER, D., AND KOOMEN, J. A. G. M. 1982. Implementation of Interlisp on the VAX. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Pittsburgh, PA, Aug. 15–18, 81–87.

BEKKERS, Y. AND COHEN, J. 1992. General discussions. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France). LNCS 637, Springer-Verlag.

BEKKERS, Y., RIDOUX, O., AND UNGARO, L. 1992. Dynamic memory management for sequential logic programming languages. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France). LNCS 637, Springer-Verlag, 82–102.

BEN-ARI, M. 1984. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst. 6,* 333–344.

BENNETT, J. K. 1987. The design and implementation of distributed Smalltalk. *OOPSLA '87. ACM SIGPLAN Not. 22,* 12, 318–330.

BENGTSSON, M. AND MAGNUSSON, B. 1990. Real-time compacting garbage collection. Position paper. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection.*

BEVAN, D. I. 1987. Distributed garbage collection using reference counting. In *PARLE '87—Parallel Architectures and Languages Europe.* LNCS 259, Springer-Verlag, 176–187.

BLACK, A., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. 1987. Distribution and abstract types in Emerald. *ACM Trans. Softw. Eng. 13,* 1, 65–76.

BIRRELL, A., EVERS, D., NELSON, G., OWICKI, S., AND WOBBER, E. 1993. Distributed garbage collection for network objects. TR 116, Digital Equipment Corp. Research Center.

BISHOP, B. 1977. Computer systems with very large address space and garbage collection, Ph.D. thesis, MIT, Cambridge, MA.

BOBROW, D. G. 1980. Managing reentrant structures using reference counts. *ACM Trans. Program. Lang. Syst. 2,* 3, 269–273.

BOEHM, J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exper. 18,* 9, 807–820.

BROOKS, R. A., GABRIEL, R. P., AND STEELE, G. L. 1982. S-1 common Lisp implementation. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Pittsburgh, PA, Aug. 15–18), 108–113.

BROWNBRIDGE, D. R. 1985. Cyclic reference counting for combinator machines. In *Functional Programming Languages and Computer Architecture.* LNCS 201, Springer-Verlag, 273–288.

CARLSSON, S., MATTSSON, C., AND BENGTSSON, M. 1990. A fast expected-time compacting garbage collection algorithm. Position paper. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection.*

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of SELF: A dynamically-typed object-oriented language based on prototypes. *OOPSLA '89, ACM SIGPLAN Not. 24,* 10, 49–70.

CHAMBERS, F. B., DUCE, D. A., AND JONES, G. P., EDS. 1984. *Distributed Computing,* Academic Press. London.

CHENEY, C. J. 1970. A non-recursive list compacting algorithm. *Commun. ACM 13,* 11, 677–678.

CHIKAYAMA, T. AND KIMURA, Y. 1987. Multiple reference management. In *Flat GHC, ICLP,* MIT Press, 276–293.

CLARK, D. W. AND GREEN, C. C. 1977. An empirical study of list structure in Lisp. *Commun. ACM 20,* 2, 78–86.

CLARK, D. W. AND GREEN, C. C. 1977. A note on shared list structure in Lisp. *Inf. Process. Lett. 7,* 6, 312–314.

COHEN, J. 1981. Garbage collection of linked data structures. *ACM Comput. Surv. 13,* 3, 341–367.

COHEN, J. AND NICOLAU, A. 1983. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst. 5,* 4, 532–553.

COHEN, J. AND TRILLING, L. 1967. Remarks on garbage collection using a two level storage. *BIT 7,* 1, 22–30.

COLLINS, G. E. 1960. A method for overlapping and erasure of lists. *Commun. ACM 3,* 12, 655–657.

COULOURIS, G. F., DOLLIMORE, J., AND KINDBERG, T. 1994. *Distributed Systems: Concepts and Design* 2nd ed., Addison-Wesley.

COURTS, R. 1988. Improving locality of reference in a garbage-collecting memory management system. *Commun. ACM 31,* 9, 1128–1138.

CRICHLOW, J. M. 1988. *An Introduction to Distributed and Parallel Computing.* Prentice Hall, Englewood Cliffs, NJ.

DAVIES, D. J. M. 1984. Memory occupancy patterns in garbage collection systems. *Commun. ACM 27,* 8, 819–825.

DAWSON, J. L. 1982. Improved effectiveness from a real-time Lisp garbage collector. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Pittsburgh, PA, Aug. 15–18), 159–167.

DEHNEN, F. AND LINS, R. D. 1994. Distributed cyclic reference counting. In *Parallel and Distributed Computing (Theory and Practice).* LNCS 805, Springer-Verlag, 95–100.

DELLAR, C. N. R. 1980. Removing backing store administration from the CAP operating system. *Oper. Syst. Rev. 14,* 4, 9–41.

DELOBEL, C., LECLUSE, C., AND RICHARD, P. 1995. *Databases: from Relational to Object-Oriented Systems.* ITP.

DENNING, P. J. 1968. Thrashing: its causes and prevention. In *Proceedings of the AFIPS National Computer Conference,* 915–922.

DERBYSHIRE, M. H. 1990. Mark-scan garbage collection on a distributed architecture. *Lisp and Symbolic Comput. 3,* 2, 135–170.

DETLEFS, D. L. 1990a. Concurrent garbage collection for C++. Tech. Rep. CMU-CS-90-119, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA.

—. 1990b. Concurrent, atomic garbage collection. Position paper. *ECOOP/OOPSLA '90 Workshop on Garbage Collection.*

—. 1991. Concurrent, atomic garbage collection. Ph.D. thesis, Tech. Rep. CMU-CS-90-177, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA.

DEMERS, A., WEISER, M., HAYES, B., BOEHM, H., BOBROW, D., AND SHENKER, S. 1990. Combining generational and conservative garbage collection: framework and implementations. In *Proceedings of the ACM Symposium on Principles of Programming Languages,* 261–269.

DETREVILLE, J. 1990. Experience with garbage collection for Modula-2+ in the Topaz environment. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

DEUTSCH, L. P. 1983. The Dorado Smalltalk-80 implementation: Hardware architecture's impact on software architecture. In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, MA, 113–125.

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient, incremental, automatic garbage collector. *Commun. ACM 19,* 9, 522–526.

DICKMAN, P. 1991. Distributed object management in a non-small graph of autonomous networks with few failures. Ph.D. thesis, University of Cambridge.

DIJKSTRA, E. W. AND SCHOLTEN, C. S. 1989. Termination detection for diffusing computations. *Inf. Process. Lett., 11.*

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., AND STEFFENS, E. F. M. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21,* 11, 966–975.

DOLEY, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchronism needed for distributed consensus. *J. ACM 34,* 1, 77–97.

EDELSON, D. AND POHL, I. 1990. The case for garbage collector in C++. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

EL-HABBASH, A., HORN, C., AND HARRIS, M. 1990. Garbage collection in an object oriented, distributed, persistent environment. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

FALCONE, J. R. AND STINGER, J. R. 1983. The Smalltalk-80 implementation at Hewlett-Packard. In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, MA, 79–112.

FENICHEL, R. R. AND YOCHELSON, J. C. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM 12,* 11, 611–612.

FERREIRA, P. 1990. Storage reclamation. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

FERREIRA, P. AND SHAPIRO, M. 1994. Garbage collection and DSM consistency. In *Proceedings of the First Symposium on Operating Systems Design and Implementation,* 229–241.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32,* 2, 374–382.

FISHER, D. A. 1974. Bounded workspace garbage collection in an address-order preserving list processing environment. *Inf. Process. Lett. 3,* 1, 29–32.

FODERARO, J. K. AND FATEMAN, R. J. 1981. Characterization of VAX Macsyma. In *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation,* 14–19.

FOWLER, R. J. 1986. The complexity of using forwarding addresses for decentralized object finding. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing,* 108–120.

FRIEDMAN, D. P. AND WISE, D. S. 1976. Garbage collecting a heap which includes a scatter table. *Inf. Process. Lett. 5,* 6, 161–164.

FRIEDMAN, D. P. AND WISE, D. S. 1977. The one-bit reference count. *BIT 17,* 351–359.

FRIEDMAN, D. P. AND WISE, D. S. 1979. Reference counting can manage the circular environments of mutual recursion. *Inf. Process. Lett. 8,* 1, 41–45.

FUCHS, M. 1995. Garbage collection on an open network. In *Memory Management, Proceed-*

ings of the IWMM95, LNCS 986, H. G. Baker, Ed., Springer-Verlag, New York, 251–265.

GABRIEL, R. P. AND MANSINTER, L. M. 1982. Performance of Lisp systems. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Pittsburgh, PA, August 15–18), 123–142.

GARNETT, N. H. AND NEEDHAM, R. M. 1980. An asynchronous garbage collector for the Cambridge file server. *Oper. Syst. Rev. 14,* 4, 36–40.

GELERNTER, H., HANSEN, J. R., AND GERBERRICH, C. L. 1960. A FORTRAN-compiled list processing language. *J. ACM 7,* 2, 87–101.

GLASER, H. W. AND THOMPSON, P. 1985. Lazy garbage collection. *Softw. Pract. Exper. 17,* 1, 1–4.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, Reading, MA, 674–681.

GOLDBERG, B. 1989. Generational reference counting: A reduced communication distributed storage reclamation scheme. In *Programming Languages Design and Implementation, ACM SIGPLAN Not. 24,* 313–321.

HANSEN, W. J. 1969. Compact list representation: Definition, garbage collection, and system implementation. *Commun. ACM 12,* 9, 499.

HAYES, B. 1990a. Open systems require conservative garbage collectors. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

—. 1990b. Using key object opportunism to collect old objects. *OOPSLA '91, ACM SIGPLAN Not. 26,* 11, 33–46.

HICKEY, T. AND COHEN, J. 1984. Performance analysis of on-the-fly garbage collection. *Commun. ACM 27,* 11, 341–367.

HOARE, C. A. R. 1974. Optimization of store size for garbage collection. *Inf. Process. Lett. 2,* 6, 165–166.

HUDAK, P. 1982. Object and task reclamation in distributed applicative processing systems. Ph.D. Thesis, University of Utah.

—. 1986. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (MIT), 351–363.

HUDAK, P. AND KELLER, R. M. 1982. Garbage collection and task deletion in distributed applicative processing systems. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Pittsburgh, PA, August), 168–178.

HUDSON, R. AND DIWAN, A. 1990. Adaptive garbage collection for Modula-3 and Smalltalk. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

HUGHES, J. 1984. Reference counting with circular structures in virtual memory, applicative systems. Tech. Rep., Programming Research Group, Oxford University.

—. 1985. A distributed garbage collection algorithm. In *Functional Programming Languages and Computer Architecture,* LNCS 201, Springer-Verlag, New York, 256–272.

INMOS LIMITED 1984. *Occam Programming Manual*, Prentice-Hall, Englewood Cliffs, NJ.

JOHNSON, D. 1991. The case for a real barrier. *ACM SIGPLAN Not. 26,* 4, 279–281.

JONES, R. E. 1996. http://www.ukc.ac.uk/computer_science/Html/Jones/gc.html.

JONES, R. E. AND LINS, R. D. 1992. Cyclic weighted reference counting without delay. Tech. Rep. TR 28-92, UKC Computing Lab, University of Kent at Canterbury.

JONES, R. E. AND LINS, R. D. 1993. Cyclic weighted reference counting without delay. In *Proceedings of PARLE'93—Parallel Architectures and Languages Europe,* LNCS 694, Springer-Verlag, New York, 712–715.

JONES, R. E. AND LINS, R. D. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, New York.

JONKERS, H. B. M. 1979. A fast garbage compaction algorithm. *Inf. Process. Lett. 9,* 1, 26–30.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6,* 1, 109–133.

JUUL, N. C. 1990. Report. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection in Object-Oriented Systems.*

JUUL, N. C. AND JUL, E. 1992. Comprehensive and robust garbage collection in a distributed system. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France), LNCS 637, Springer-Verlag, New York, 103–115.

KAFURA, D., WASHABAUGH, D., AND NELSON, J. 1990. Garbage collection of actors. In *ECOOP/OOPSLA '90 Proceedings of Workshop on Garbage Collection,* 126–134.

KAIN, R. Y. 1969. Block structures, indirect addressing and garbage collection. *Commun. ACM 12,* 7, 395–398.

KNOWLTON, K. C. 1965. A fast storage allocator. *Commun. ACM 8,* 10, 623–625.

KNUTH, D. E. 1973. *The Art of Computer Programming; Vol 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA.

KOLODNER, E. 1991. Atomic incremental garbage collection and recovery for large stable heap, implementing persistent object bases: Principles and practice. In *Proceedings of the Fourth International Workshop on Persistent Object Systems,* Morgan-Kaufmann, San Mateo, CA.

KOLODNER, E., LISKOV, B., AND WEIHL, W. 1989. Atomic garbage collection: Managing a stable heap. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data,* 15–25.

KRASNER, G., ED. 1983. *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, MA.

LADIN, R. AND LISKOV, B. 1992. Garbage collection of a distributed heap. In *Proceedings of the International Conference on Distributed Computing Systems,* 708–715.

LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. 1991. The Object Store database system. *Commun. ACM 34,* 10, 50–63.

LAMPORT, L. 1978. Time, clocks and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

LANG, B. AND DUPONT, F. 1987. Incremental incrementally compacting garbage collection. In *SIGPLAN '87—Symposium on Interpreters and Interpretive Techniques,* 253–263.

LANG, B., QUEINNEC, C., AND PIQUER, J. 1992. Garbage collecting the world. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92).*

LERMEN, C. W. AND MAURER, D. 1986. A protocol for distributed reference counting. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (MIT), 343–350.

LI, K. 1988. Real-time concurrent collection in user mode. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection.*

LI, K., APPEL, A. W., AND ELLIS, J. R. 1988. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* 11–20.

LIEBERMAN, H. AND HEWITT, C. 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM 26,* 6, 419–429.

LINDSTROM, G. 1974. Copying list structures using bounded workspace. *Commun. ACM 17,* 4, 198–202.

LINS, R. D. 1992a. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett. 44,* 215–220.

—. 1992b. Generational cyclic reference counting. Tech. Rep. TR 22-92 UKC Computing Lab, University of Kent at Canterbury.

LINS, R. D. AND VASQUES, M. A. 1991. A comparative study of algorithms for cyclic reference counting. TR 92 UKC Computing Lab, University of Kent at Canterbury, August.

LINS, R. D. AND JONES, R. E. 1991. Cyclic weighted reference counting. TR 95, UKC Computing Lab. Tech. Rep., University of Kent at Canterbury, December.

LISKOV, B. AND LADIN, R. 1986. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Symposium on the Principles of Distributed Computing,* ACM, New York, 29–39.

MARTIN, J. J. 1982. An efficient garbage compaction algorithm. *Commun. ACM 25,* 8, 571–581.

MARTINER, A. D., WACHENCHAUZER, R., AND LINS, R. D. 1990. Cyclic reference counting with local mark-scan. *Inf. Process. Lett. 34,* 31–35.

MARTINEZ, A. D., WACHENCHAUER, R., AND LINS, R. D. 1990. Cyclic reference counting with local mark-scan. *Inf. Process. Lett. 34,* 31–35.

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine: Part I. *Commun. ACM 3,* 4, 184–195.

—. 1981. History of Lisp. In *History of Programming Languages,* R. L. Wexelblat, Ed., Academic Press, 173–183.

MCCULLOUGH, P. L. 1983. Implementing the Smalltalk-80 system: The Tektronix experience. In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, MA, 59–78.

MEYERS, R. AND CASSERES, D. 1983. An MC68000-based Smalltalk-80 system. In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, MA, 175–187.

MIRANDA, E. 1987. BrouHaHa—a portable Smalltalk interpreter. *OOPSLA '87, ACM SIGPLAN Not. 22,* 12, 354–365.

MOHAMMED-ALI, K. A. 1984a,b,c. Object-oriented storage management and garbage collection in distributed processing systems. Academic Dissertation, Royal Institute of Technology, Dept. of Computer Systems, Stockholm, Sweden.

MOON, D. 1984. Garbage collection in a large Lisp system. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming,* 235–246.

MORRIS, F. L. 1978. A time- and space-efficient garbage compaction algorithm. *Commun. ACM 21,* 8, 662–665.

—. 1979. On a comparison of garbage collection techniques. *Commun. ACM 22,* 10, 571.

MOSS, J. E. B. 1990. Garbage collecting persistent object stores. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

NEWELL, A. AND TONGE, F. M. 1960. An introduction to IPL-V. *Commun. ACM 3,* 205–211.

NEWMAN, I. A. AND WOODWARD, M. C. 1982. Alternative approaches to multiprocessor garbage collection. In *Proceedings of the 1982 International Conference on Parallel Processing* (Ohio State University, Columbus, OH, August), 205–210.

NEWMAN, I. A., STALLARD, R. P., AND WOODWARD,

M. C. 1982. Performance of parallel garbage collection algorithms. *Comput. Stud. 166* (Sept.).

NILSEN, K. 1988. Garbage collection of strings and linked data structures in real time. *Softw. Pract. Exper. 18,* 7, 613–640.

NILSEN, K. AND SCHMIDT, W. J. 1990. Hardware support for garbage collection of linked objects and arrays in real time. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

NORI, A. K. 1979. A storage reclamation scheme for applicative multiprocessor systems. Masters Thesis, Department of Computer Science, University of Utah.

NORTH, S. C. AND REPPY, J. H. 1987. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture,* LNCS 274, Springer-Verlag, New York, 113–133.

OZAWA, T., HOSOI, A., AND HATTORI, A. 1990. Generation type garbage collection for parallel logic languages. *NACLP,* MIT Press, Cambridge, MA, 291–305.

PARC-PLACE 1991. *Objectworks/Smalltalk Release 4 User's Guide.* Memory Management, 229–237.

PETERSON, L. L. AND DAVIE, B. S. 1996. *Computer Networks: A Systems Approach.* Morgan-Kaufmann, San Mateo, CA.

PEYTON-JONES, S. L. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall, Englewood Cliffs, NJ.

PIQUER, J. M. 1991. Indirect reference counting: A distributed garbage collection algorithm. In *Proceedings of PARLE '91—Parallel Architectures and Languages Europe,* LNCS 505, Springer-Verlag, New York, 150–165.

—. 1995. Indirect mark and sweep. In *Memory Management, Proceedings of IWMM95,* H. G. Baker, Ed., LNCS 986, Springer-Verlag, New York, 268–282.

PLAINFOSSÉ, D. 1994. Distributed garbage collection and reference management in the Soul object support system. Ph.D. Thesis, Université Paris.

PLAINFOSSÉ, D. AND SHAPIRO, M. 1992. Experience with fault-tolerant garbage collection in a distributed Lisp system. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France), LNCS 637, Springer-Verlag, New York, 116–133.

PLAINFOSSÉ, D. AND SHAPIRO, M. 1995. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (Kinross, UK), LNCS 986, Springer-Verlag, New York, 211–249.

QUEINNEC, C., BEAUDOING, B., AND QUEILLE, J. 1989. Mark DURING sweep rather than mark THEN sweep. In *Proceedings of PARLE '89—Parallel Architectures and Languages Europe,* LNCS 365, Springer-Verlag, New York.

RANA, S. P. 1983. A distributed solution to the distributed termination problem. *Inf. Process. Lett. 17,* 43–46.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10,* 1, 26–52.

RUDALICS, M. 1986. Distributed copying garbage collection. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* MIT, Cambridge, MA, 364–372.

—. 1990. Correctness of distributed garbage collection algorithms. TR 90-40.0, Johannes Kepler Universität, Linz.

SCHELVIS, M. 1989. Incremental distribution of timestamp packets: A new approach to distributed garbage collection. *OOPSLA '89, ACM SIGPLAN Not. 24,* 10, 37–48.

SCHELVIS, M. AND BLEDOEG, E. 1988. The implementation of a distributed Smalltalk. In *ECOOP Proceedings* (August), LNCS 322, Springer-Verlag, New York, 212–232.

SCHORR, H. AND WAITE, W. M. 1967. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM 10,* 8, 501–506.

SHAPIRO, M., DICKMAN, P., AND PLAINFOSSÉ, D. 1992a. Robust, distributed references and acyclic garbage collection. In *ACM Symposium on Principles of Distributed Computing* (Vancouver).

SHAPIRO, M., DICKMAN, P., AND PLAINFOSSÉ, D. 1992b. SSP chains: Robust, distributed references and acyclic garbage collection. Tech. Rep. TR INRIA.

SHAPIRO, M., PLAINFOSSÉ, D., AND GRUBER, O. 1990. A garbage detection protocol for a realistic distributed object-support system. Tech. Rep. TR INRIA 1320.

SHAPIRO, M., PLAINFOSSÉ, D., FERREIRA, P., AND AMSALEG, L. 1994. Some key issues in the design of distributed garbage collection and references. Tech. Rep. TR INRIA.

SHARMA, R. AND SOFFA, M. L. 1991. Parallel generational garbage collection. *OOPSLA 91, ACM SIGPLAN Not. 26,* 11, 16–32.

SHAW, R. A. 1987. Empirical analysis of a Lisp system. Ph.D. Thesis, Stanford University, Stanford, CA. February, 1988.

STANDISH, T. A. 1980. *Data Structures Techniques.* Addison-Wesley, Reading, MA.

STEELE, G. L. 1975. Multiprocessing compactifying garbage collection. *Commun. ACM 18,* 9, 495–508.

STEENKISTE, P. 1987. Lisp on a reduced-instruction-set processor: Characterization and optimization. Ph.D. Dissertation. TR CSL-TR-87-324, Stanford University, March.

SUN 1997. [http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html.

TANENBAUM, A. S. 1989. *Computer Networks.* Prentice-Hall, Englewood Cliffs, NJ.

TEL, G. AND MATTERN, F. 1993. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Trans. Program. Lang. Syst. 15,* 1, 1–35.

TERESHIMA, M. AND GOTO, E. 1978. Genetic order and compactifying garbage collector. *Inf. Process. Lett. 7,* 1, 27–32.

TICK, E. 1988. *Memory Performance of Prolog Architectures.* Kluwer, Norwell, MA.

TUCK, B. 1990. *OSI and Library Services.* Library and Information Briefings, University of Westminster.

UNGAR, D. M. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (April), 157–167.

—. 1987. *The Design and Evaluation of a High Performance Smalltalk System.* An ACM Distinguished Dissertation 1986, MIT Press, Cambridge, MA.

UNGAR, D. M. AND JACKSON, F. 1988. Tenuring policies for generation-based storage reclamation. *OOPSLA '88 ACM SIGPLAN Not. 23,* 11, 1–17.

UNGAR, D. M. AND JACKSON, F. 1992. An adaptive tenuring policy for generation scavengers. *ACM Trans. Program. Lang. Syst. 14,* 1, 1–27.

UNGAR, D. M. AND PATTERSON, D. A. 1983. Berkeley Smalltalk: Who knows where the time goes? In *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, MA, 189–206.

VESTAL, S. C. 1987. Garbage collection: An exercise in distributed, fault-tolerant programming. Ph.D. Thesis, Department of Computer Science, University of Washington, Seattle, WA, January.

WADLER, P. L. 1976. Analysis of an algorithm for real-time garbage collection. *Commun. ACM 19,* 9, 491–500.

WATSON, I. 1986. An analysis of garbage collection for distributed systems. TR Department of Computer Science, University of Manchester.

WATSON, P. AND WATSON, I. 1987. An efficient garbage collection scheme for parallel computer architecture. In *Proceedings of PARLE '87—Parallel Architectures and Languages Europe,* LNCS 259, Springer, 432–443.

WEGBREIT, B. 1972. A generalized compacting garbage collector. *Comput. J. 15,* 3, 204–208.

WEIZENBAUM, J. 1962. Knotted list structures. *Commun. ACM 5,* 3, 161–165.

—. 1963. Symmetric list processor. *Commun. ACM 6,* 9, 524–544.

WENG, J. 1979. An abstract implementation for a generalized data-flow language. TR MIT/LCS/228, MIT Laboratory for Computer Science.

WHITE, J. L. 1980. Address/memory management for a gigantic Lisp environment or GC considered harmful. In *Record of the 1980 Lisp Conference,* 119–127.

WILSON, P. R. 1990. Some issues and strategies in heap management and memory hierarchies. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

—. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France), LNCS 637, Springer-Verlag, New York, 1–42.

WILSON, P. R. AND MOHER, T. G. 1989. Design of the opportunistic garbage collector. *OOPSLA '89, ACM SIGPLAN Not. 24,* 10, 23–35.

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. 1995. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management* (Kinross, UK), LNCS 986, Springer-Verlag, New York, 1–116.

WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1990. Caching considerations for generational garbage collection: A case for large and set-associative caches. TR UIC-EECS-90-5, December.

WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1991. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto), 177–191.

WOLCZKO, M. AND WILLIAMS, I. 1990. Garbage collection in high-performance system. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

WOODWARD, M. C. 1981. Multiprocessor garbage collection—a new solution. *Comput. Stud. 115.*

ZAVE, D. A. 1975. A fast compacting garbage collector. *Inf. Process. Lett. 3,* 167–169.

ZORN, B. 1989. Comparative performance evaluation of garbage collection algorithms. Ph.D. Thesis, EECS Department, University of California, Berkeley.

—. 1990. Designing systems for evaluation: A case study of garbage collection. In *Proceedings of the ECOOP/OOPSLA '90 Workshop on Garbage Collection,* position paper.

—. 1992a. The measured cost of conservative garbage collection. TR CU-CS-573-92. Department of Computer Science, University of Colorado, Boulder, April.

—. 1992b. Evaluating models of memory allocation. TR CU-CS-603-92, Department of Computer Science, University of Colorado, Boulder, July.

ZORN, B. AND GRUNWALD, D. 1992. Empirical measurements of six allocation-intensive C programs. *ACM SIGPLAN Not. 27,* 12 (Dec.), 71–80.

ZORN, B. AND HILFINGER, P. 1988. A memory allocation profiler for C and Lisp programs. In *Proceedings of Summer 1988 USENIX Conference* (San Francisco, June).

Received March 1996; revised November 1997; accepted November 1997