Where have all the pointers gone?: A story of distributed pointer tracking

Richard L. Hudson,[¥] Ron Morrison,[†] J. Eliot B. Moss,[¥] & David S. Munro[†]

[¥]Department of Computer Science, University of Massachusetts,
Amherst, MA 01003, U.S.A.
Email: {hudson, moss}@cs.umass.edu

[†]School of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SS, Scotland
Email: {ron, dave}@dcs.st-and.ac.uk

Abstract

We present new algorithms for tracking pointers in distributed object systems where each node in the system has its own local storage and may communicate with other nodes only by passing messages. The problem is difficult because of asynchrony, implying lack of knowledge of global state, and lack of globally atomic operators on that state. The pointer tracking algoritms may be used as part of a garbage collector to identifiy when there are no pointers to an object from another node, in object migration protocols where one object is substituted by another object possibly located on a different node, and in persistent systems to identify persistent data. A discussion of the correctness of the algoritms is given.

Keywords: distributed systems, pointer tracking, persistence

1 Introduction

The ability to identify which objects hold references to other objects is an essential property of all systems based on reachability such as: garbage collection systems [Wilson92]; object migration systems [PS95]; and persistent object systems [AM95]. We term the mechanism for achieving this *pointer tracking.* In garbage collection it is important to find out when an object is no longer referenced; in object migration systems the objects holding references must be informed of the new location of the object that is substituted for the old one; and in persistent object systems the transitive closure of the references from some root is computed to identify live data in order to preserve the integrity of the store. Pointer tracking serves all of these applications.

Here we are concerned with pointer tracking in a distributed system where each node in the system has its own local storage and may communicate with other nodes only by passing messages. The problem is difficult because of asynchrony, implying lack of knowledge of global state, and lack of globally atomic operators on that state.

The pointer tracking algorithms assume the following support:

- Each node in the system has its own local storage and may communicate with other nodes only by passing messages.
- Ordered delivery of messages is guaranteed without omission, corruption, or duplication. Causal delivery is not assumed.
- Nodes appear to operate correctly, without crashes or Byzantine behaviour.
- No bounds are placed on the relative rates of computation of the nodes.
- Events and actions at a given node are totally ordered, yielding a partial ordering of events in the system as a whole.

As presented, the pointer tracking algorithms are well suited to distributed memory multiprocessors and to applications that do not require fault-tolerance. To widen their applicability, fault-tolerance may be provided by lower levels of the system. While it is possible to build some of this support in, we regard these facilities as being provided by lower level protocols, upon which the pointer tracking algorithms can be built, in order

to separate policy and mechanism and to keep the levels of abstraction relatively understandable.

1.1 The Computational Model

Our distributed computational model is made up of computation, objects, and pointers.

Computation consists of one logical *process*, perhaps with concurrent threads, per (logical) node with the processes communicating via *messages*. Computation proceeds by creating and mutating *objects*. A physical node may support more than one logical node.

A (physical) object resides on a single node, which we term the object's *home*, and may contain any number of pointers (references to other objects) as well as non-pointer data.

Each node has zero or more *root pointers* to objects, which we can view as being part of the node's process's state. Pointers may propagate to other nodes, and be stored there, by being included in messages.

We will assume that references somehow encode the home node and the location of the referent object there and that object identifiers are unique.

2 Pointer Tracking

We are concerned with mechanisms that track external references to objects (from objects outside the node to objects inside the node) where the mechanisms are both safe and complete. To avoid global synchronisation we cannot demand that such knowledge be entirely up-to-date at any one node. Safety requires that a referenced object is never thought to be unreferenced, and completeness requires that a node eventually discovers when there are no longer external references to a local object.

The pointer tracking algorithm is designed to meet the above criteria by ensuring that the home node H, of an object o, is informed of any relevant manipulation of a pointer to o by another node. Since the knowledge of the system cannot be required to be up-to-date at any one node, we must choose objectives for the pointer tracking algorithms to support applications. These objectives are to ensure that H has sufficient information:

- to allow it to know eventually that there are no pointers to o from other nodes,[1] and
- to allow object o to be substituted by object o' not necessarily on H.

First we describe the algorithms for pointer tracking during normal systems operation, deferring discussion of the correctness of the pointer tracking until after its description. We follow this by a description of object substitution.

2.1 Events Related to Pointer Tracking

Our pointer tracking mechanism consists of handling four events. The home node, H, is informed of these events via asynchronous messages. The events are detailed in Table 1.

[1] The protocol is designed specifically to avoid any need for causal messaging [Fidge96].

Event	Description
<s, o, B>	This (send) event indicates that node B has been sent a pointer to o. This event is said to *happen at* the sender, hence it is the sender's responsibility to inform H.
<r, o, B>	This (receive) event indicates that node B has received a pointer to o. The *r* event has two roles: one as a + event (see below) at the receiver and the other to balance the *s* event that caused the pointer to be sent. This event happens at B.
<+, o, A>	This event indicates that node A has created a new pointer to object o. This event happens at A.
<–, o, A>	This event indicates that node A has deleted a pointer to object o. This event happens at A.

Table 1: Pointer Tracking Events

Figure 1 illustrates a possible scenario of events in which a pointer is sent from one node to another. Messages sent are drawn as broken arrows and object pointers signified by unbroken arrows. Node A sends a copy of the pointer to node B and eventually informs H of the *s* event. Node B receives the pointer, and informs H of the *r* event. Node B copies the pointer,[2] eventually informing H of the + event (and eventually a – event to balance the + when it deletes this copy), a process which may be repeated many times. Finally, B deletes the original copy of the pointer and eventually informs H of the – event.

There is no specific requirement for rapid delivery to H of information about events at A and B. This is intentional, since such information can normally be piggy-backed on other communication, thereby reducing the overhead of the scheme. However, we require that H is informed of events that happen at any node in the order in which they happen at that node.[3] Messages 2, 3, and 4 in Figure 1 arrive at H in that order. There is, however, no constraint on the arrival of message 1, relative to messages 2, 3, and 4.

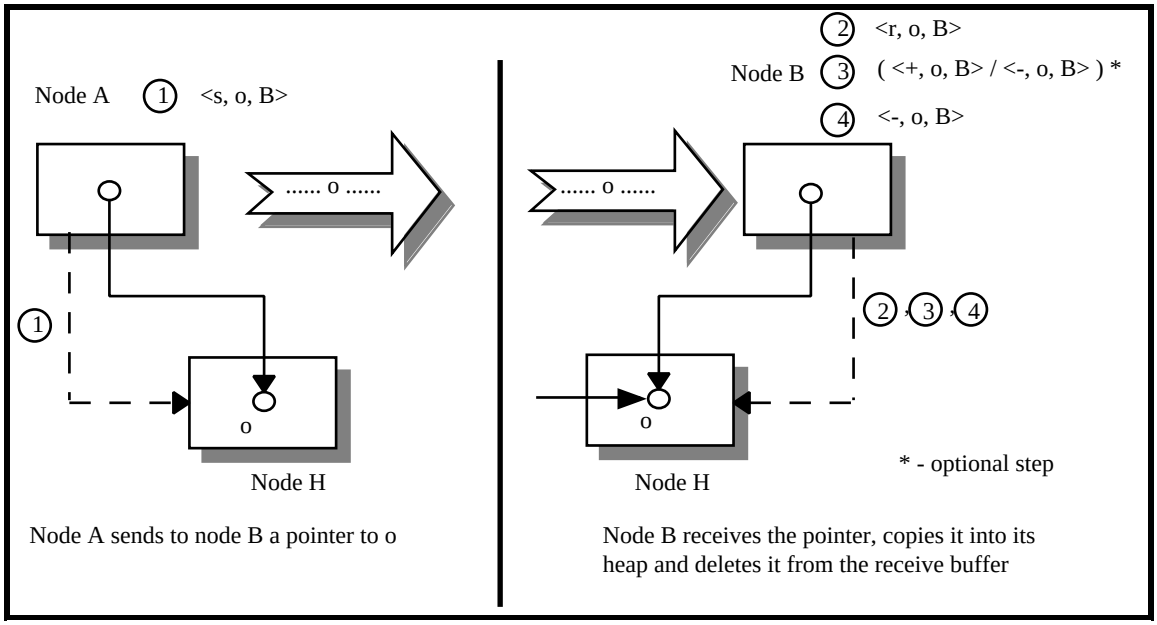

Figure 1: Node A Sends to Node B a Pointer to o

---

[2] Note that this is optional, and hence the * in the diagram.

[3] This is simplified by our assumption of in-order message delivery.

In Table 1, either A or B can be the home node, H. We assume that H is informed immediately of events that happen at H.

2.2 Constraints on Ordering of Events

To describe the correct operation of the system there are a number of constraints on the order in which the four kinds of events can occur. For this we introduce a precise definition of the predicate *any(o, Y, E)*, which indicates whether node Y has any pointers to object o in the situation described by the set of events E.

Definition: Given a set of events E, where each event is of the form described in Table 1, any(o, Y, E) for an object o and node Y holds iff E contains an event <r, o, Y> but not a pairing event <–, o, Y>, or E contains an event <+, o, Y> but not a pairing event <–, o, Y>.[4]

The legal event sets for the system are defined recursively, in terms of the events that may legally be added to a given event set E, as defined in Table 2:

Rule	Intuition
The empty event set is legal.	Initial state.
H adds to E a <+, o, H> event to E when object o is created at H.	The home node can create objects.
If any(o, A, E) is true then A can add an <s, o, B> event to E.	If A has a pointer to o then it can send it to any node.
If event <s, o, B> is in E but event <r, o, B> is not, then the *r* event may be added to E.	If a pointer has been sent but not yet received then the *r* event can occur.
If event <r, o, B> is in E but a pairing event <–, o, B> is not, then the – event may be added to E.	If a pointer has been received but not finally deleted then it may be so deleted. Note that before the final – event any(o, B, E) is true.
If any(o, A, E) then a <+, o, A> event to E may be added to E.	If A has a pointer to o then it may create a copy of the pointer.
If <+, o, A> is in E but a pairing <-, o, A> is not, then the – event may be added to E.	If A has a copy of a pointer that it has not yet deleted then it may delete it. Note that before the – event any(o, A, E) is true.

Table 2: Legal Event Sets

2.3 Piggy-backing and compressing messages

Since rapid delivery is not essential, messages informing home nodes of pointer events can be held and piggy-backed on other communications. Further, sequences of events can be compressed. For example, an *r* event, then a + event, then a – event can be processed to compress the *r* and – events together. We will not pursue the issue further here, since it does not relate to correctness or completeness of our algorithms, though the performance improvements may be important in practice.

3 Are there any pointers out there?

Our first objective was to determine that no other node has pointers to a given object. To establish this we require a further definition.

Definition: *absence(o, E)* is true iff any(o, X, E) is false for all nodes X other than H, the home node of o, and each event <s, o, B> can be paired with a receive event <r, o, B> in E.

---

[4] That is, #<r, o, Y> + #<+, o, Y> > #<–, o, Y>

Claim: Let E be the set of events known at H involving o; we call this H's view. If absence(o, E) then no node other than H has a pointer to o.

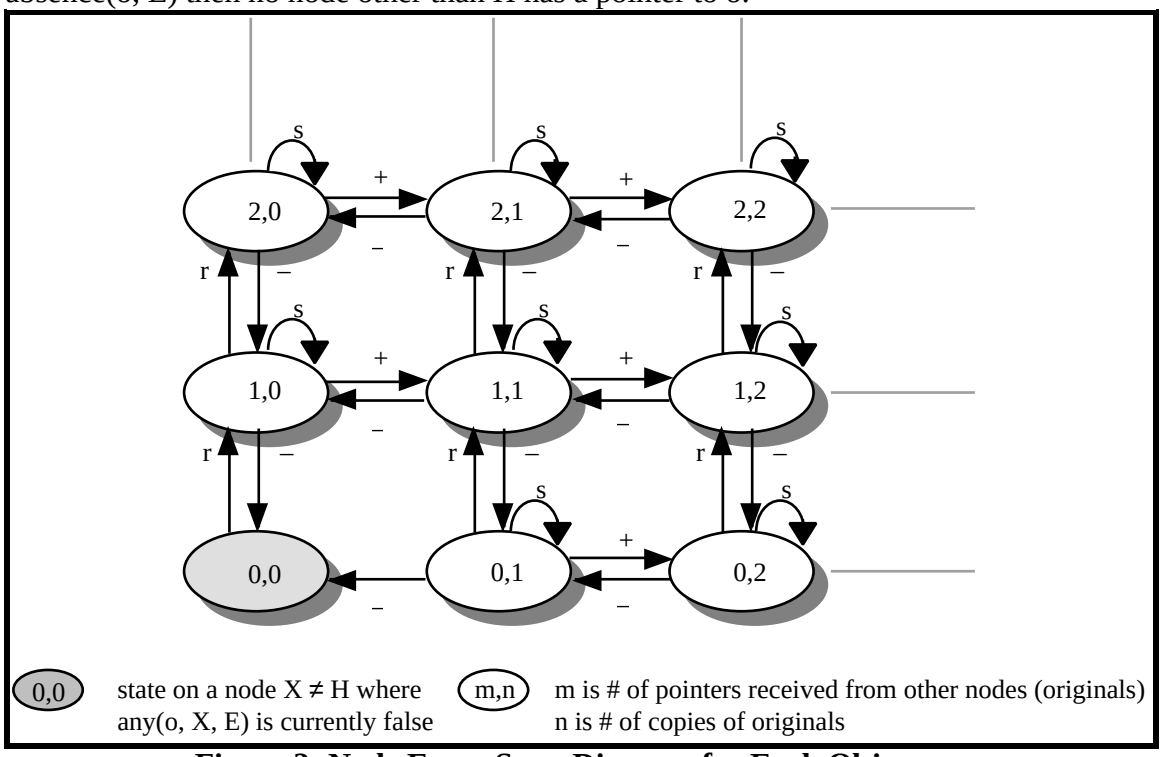

Figure 2: Node Event State Diagram for Each Object

Proof: The intuition is that if in H's view no other node has a pointer and there are no pointers in transit, then H's view is correct and is up to date.

Before tackling the proof itself, it is helpful to examine the state diagram presented in Figure 2, which shows the possible states of a node X ≠ H with respect to the number of copies it may have of a given pointer. We distinguish between the number of pointers that X has received (originals) (*m*) and the number of copies of those originals (*n*). The *s*, *r*, +, and – arrows signify send, receive, and +/– events occurring at X. In the initial (and final) state (*m* = 0, *n* = 0) the only event that can occur on this node is a receive event.

Firstly, observe that the legal event rules constrain adding events that happen at a given node X primarily in terms of what has already happened at X. The sole exception is *r* events, which require a matching *s* event, which generally occurs at a different node.

Secondly, observe that since message delivery is ordered, H's view of what has happened at a node X is a prefix of what has actually happened at X.

Consider any node X other than H. Now assume that absence(o, E) is true, implying that in H's view, X's state machine is in state (0, 0). Thus any(o, X, E) is false. Further, any(o, X, E|X) is also false, where E|X means the set of events in E that happened at X.[5] E|X is a prefix of what has actually happened at X, but since any(o, X, E|X) is false, the only legal event that could be added to E|X is an *r* event. Thus the only legal next event at any node in H's view is an *r* event.

Since all *s* events in H's view are matched with corresponding *r* events, the only possible source of a new *s* event is H. We observe that H's view of itself is necessarily up to date, so H can be the only node possessing a pointer to o. Put another way, since no other node can legally add an *s* event, no other node possesses a pointer to o.

---

[5] This is easily seen from the definition of any(o, X, E) since it refers only to events that happen at X.

Our conclusion relies only on accepting the definition of any(o, X, E) as corresponding to our informal notion of X possessing any pointers to o in the situation described by E, the ordered and reliable delivery of messages between any pair of nodes, and the definition of legal event sets.

To implement the above, two counts for each object and node are required at H: an *inTransit*(o, →B) count to record the number of pointers to object o sent out to node B but not yet received, and a *pointersTo*(o, B) count to record whether node B has any pointers to object o. The effect on these counts of messages arriving at H is summarised in Table 3.

Event	**Effect at H**	
	inTransit (o, →B)	**pointersTo** (o, B)
<s, o, B>	+1	
<r, o, B>	–1	+1
<+, o, B>		+1
<–, o, B>		–1

Table 3: Counts at H

The counts can take on any integer value including negative numbers. However, where it is only required to record that a node has a pointer to an object or not then the pointersTo count can be restricted to 0 or 1 provided that node B only generates + and – events when the number of pointers changes from 0 to 1 and visa versa.

4 Object Substitution Protocol

The pointer tracking algorithms allow movement of (logical) objects within nodes and allow applications to move objects across nodes, so they include algorithms to *substitute* one physical object for another and to update the affected references. When an object is copied, all references to the old copy must be updated to refer to the new copy.

Any object substitution protocol must

- work while references are being updated (safety), and
- find and update all references eventually (completeness).

The above must be achieved in the presence of asynchrony.

The specific goal of object substitution is to replace object o, home node H, with object o', home node H' (where H' may or may not be H), and to have all pointers to o in the entire system eventually replaced with pointers to o'.

To support object substitution, home nodes, H, maintain for each moved object o, KnownNodes(o), the set of nodes that H knows have had pointers to o since the decision was made to substitute o' for o. Likewise, all nodes maintain *object relocation tables* with entries of the form o o', meaning that o has been substituted by o'.

The algorithm is described through the messages required to support it and how nodes should respond to those messages:

- When o is substituted by o', H adds o o' to its relocation table, and initialises KnownNodes(o) to contain those nodes X for which any(o, X, E) is true in H's current view E. Then H sends a message [m, o, o'] (m is for *move*) to each node in KnownNodes(o). Note that the *m* message should be considered an *s* event for o', but not for o, in the pointer tracking algorithm.
- When a node X receives a message [m, o, o'], it adds o o' to its object relocation table. The *m* message should be treated as an *r* event of o' but not o, and the relocation table entry should count as an occurrence of o' but not of o.

- Once X has inserted a relocation table entry o o', it should (at its leisure, but before deleting the table entry) replace any pointers to o with pointers to o'. Such replacement is considered a <+, o', X> event and a <–, o, X> event.
- If X receives a pointer to o while it has o o' in its relocation table, it should replace o with o', causing events <r, o, X>, <+, o', X>, and <–, o, X>.
- If H has o o' in its relocation table and is informed of an event of the form <s, o, X>, <r, o, X>, <+, o, X>, or <–, o, X> then it should check whether X is in KnownNodes(o); if it is not, then it should be added and an *m* message sent to X.
- Since H deletes the contents of o during the substitution, a <–, …> event is induced at H for each pointer in o. Likewise a <+, …> event is induced at H' for each pointer in o'. If H ≠ H' then appropriate *s* and *r* events are also induced. As soon as the contents of o have been copied either directly into o' or into a message to H' then the space occupied by o may be reclaimed. This works since the object identifier o is unique and will not be reused.
- If H ≠ H', then the creation and management of the copy requires more steps. H sends a message to H' indicating that it would like to migrate o and requesting a pointer to the copy o' that H' will allocate. This message includes the contents of o, and is considered an *s* event at H for each pointer in the contents of o, but this communication to H' should not be considered an *s* event of o to H'. H' sends back a response to H with the new pointer, which is considered an *s* event of o'. H proceeds as described above. This protocol does not allow H' to reject the request.
- If H ≠ H', and H receives a message to manipulate o, then in addition to the protocol steps above, H forwards the request to H'.

4.1 Cleaning up the Tables

Upon detection of absence(o, E) using the pointer tracking algorithms and completion of substitution of o' for o at H, H sends [e, o, o'] (e is for *end move*) to each node X in KnownNodes(o), deletes KnownNodes(o), and removes o o' from its relocation table. This last step is a <–, o', H> event. An [e, o, o'] message is not an *s* event for either o or o'.

When node X receives [e, o, o'], it removes o o' from its relocation table which is a <–, o', X> event.

Note that the substitution protocol is entirely asynchronous and never delays computation.

4.2 Safety

We now argue that the pointer tracking algorithms never regard a reachable object as unreachable. Let us first consider the atomicity of object substitution. If o' is substituted for o with both o' and o at node H, H can ensure that the substitution is atomic locally. Other nodes can only pass around pointers to o, and we previously argued that pointers to o will be replaced with pointers to o', provided the number of nodes involved is bounded. Further, the substitution algorithms work by making the substitution atomically at each affected node, as the information reaches that node. Any later messages containing pointers to o are updated before the mutator can see them.

If o and o' are on different home nodes H and H', H' takes over responsibility for the migrated object as soon as the information arrives at H', and H gives up manipulating o as soon as it sends it to H'. There is a period of time during which H does not know the new identifier o' for o at H', and will have to refer application requests concerning o to H' under the identifier o, but H' will use its relocation table to rewrite the incoming pointer to o', so everything works out without indefinite waits.

Our point is that object reachability related to object substitution is not a problem. Observe also that since object relocation tables are considered to contain pointers to the

new objects (o' in the example), the new object will not be unreachable until we have cleaned up all pointers to the old one, or the new object is itself substituted.

4.3 Completeness

The clean up condition above establishes that all pointers to o have been deleted, either by applications directly, or by having o' substituted. Thus the table clean up actions are safe. The tricky part is understanding why termination will eventually be achieved. In fact, if the number of nodes is not bounded, then pointers to o may continue to propagate ahead of the *m* messages, so termination is not guaranteed without bounding system size or somehow constraining propagation behaviour. This appears to be unavoidable.

If we bound the number of nodes to which o is propagated, it can be passed around among those nodes forever. So without the KnownNodes set and relocation tables kept at the remote nodes substitution might never terminate. That is why we introduced them. The KnownNodes set prevents sending an *m* message to the same remote node twice and also allows us to send the *e* messages so that remote nodes can clean up their relocation tables.

5 Related Work

Hughes [Hughes85] uses time stamps based on global time to trace live objects. Each trace initiated on a node uses the time stamp to mark objects. Each outgoing pointer uses the time stamp whenever it propagates the trace to other nodes. The algorithm requires a globally synchronised clock, and message delivery time must be bounded. Given these requirements, Hughes shows that any object with a time stamp older than a certain time is unreachable. The termination algorithm used by Hughes is not scalable and reclamation of distributed garbage can be blocked until the slowest node in the system performs a local garbage collection.

Liskov and Ladin [LL86] propose using a centralised server to calculate global accessibility of objects. The idea is that each node informs the centralised (but possibly replicated) server of any pointers into and out of the node. The local collector is responsible for determining the connectivity between the incoming and outgoing references. Rudalics [Rudalics90] points out an error in the original algorithm that is corrected by Ladin and Liskov [LL92] using an adaptation of Hughes's time stamp algorithm. Their solution also uses the centralised server clock to simplify Hughes's termination algorithm.

Lang, Queninnec, and Piquer [LQP92] propose a technique where spaces (or nodes) are grouped. Any unreachable objects completely within a group are identified using a tracing algorithm. The groups can be hierarchically ordered so that increasingly large groups are traced. Ultimately, the entire system needs to be traced in order to identify unreachable objcets not located entirely within a previously associated group. This is therefore not scalable and requires a considerable amount of co-ordination between the nodes. Maheshwari and Liskov [ML97] claim that the algorithm will not terminate correctly if the object graph is mutated concurrently with tracing.

Ferreira and Shapiro [FS96] propose a system that allows replication of segments at multiple sites. Each segment maintains a list of incoming and outgoing pointers and is traced using these pointers as roots. Segments that appear at the same site are collected together so cyclic structures that span segments can be identified only if they are gathered at a single site. The co-ordination of segments is not a problem since replication is assumed.

Maheshwari and Liskov [ML97] describe a partitioned garbage collector that piggy-backs global marking with the marking of partitioned data. Their scheme is guaranteed to terminate correctly, and while not as yet distributed, is optimised for efficient tracking of a partition's incoming and outgoing pointers.

6 Conclusions

We have presented new algorithms for tracking pointers in distributed object systems. The pointer tracking algoritms may be used as part of a garbage collector to identifiy

when there are no points to an object from another node, in object migration protocols where one object is substituted by another object possibly located on a different node, and in persistent systems to identify persistent data.

The pointer tracking algorithm is designed to ensure that the home node, H, of object o has sufficient information:

- to allow it to know eventually that there are no pointers to o from other nodes, and
- to allow object o to be substituted by object o'.

Correctness and safety arguments are given in the paper. Interesting work remaining to be done includes implementation and practical evaluation, algorithmic performance analysis, and extensions to tolerate node and communications failures. We intend to address this in future work.

7 Acknowledgements

The pointer tracking algorithms are used in the DMOS collector [HMMM97]. We thank Peter Bailey, Huw Evans and Al Dearle for their constructive comments regarding this paper. The work was supported by NSF grants IRI-9632284 and INT-9600216 and by EPSRC Grant GR/J67611.

8 References

[AM95] Atkinson, M.P. & Morrison, R. “Orthogonally Persistent Object Systems”. VLDB Journal 4, 3 (1995) pp 319-401.

[Fidge96] C. J. Fidge. Fundamentals of distributed system observation. IEEE Software, 13(6):77-83, November 1996.

[FS96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, IEEE Press, 1996.

[Hughes85] R. John M. Hughes. A distributed garbage collection algorithm. In *Proceedings of the 1985 Conference on Functional Programming and Computer Architecture*, number 201, *Lecture Notes in Computer Science*, pp. 256-272, Springer-Verlag, 1985.

[HMMM97] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, David S. Munro. Garbage Collecting the World: One Car at a Time. OOPSLA 1997. Also available as a University of St Andrews, Dept. of Computer Science Technical Report (http://www-fide.dcs.st-and.ac.uk/Publications/1997.html#dmos).

[LL86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Fifth ACM Symposium on the Principles of Distributed Computing*, pp. 29-39, 1986.

[LL92] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Proceedings of the International Conference on Distributed Computing Systems*, IEEE Press, 1992.

[LQP92] Bernard Lang, Christian Queinniec, and Jose Piquer. Garbage collecting the world. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 39-50, ACM Press, 1992.

[ML97] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In Proceedings of ACM SIGMOD ’97, Phoenix, Arizona, 1997.

[PS95] David Plainfossé and Marc Shapiro, A Survey of Distributed Garbage Collection Techniques. In *Proceedings of International Workshop on Memory Management*, Kinross, Scotland, pp. 211-249, September 1995.

[Rudalics90] Martin Rudalics. Correctness of distributed garbage collection algorithms. Technical Report 90-40.0, Johannes Kepler Universitat, Linz Austria, 1990.

[Wilson92] Paul R. Wilson. Uniprocessor garbage collection techniques. In [BC92].