# Dynamic Join and Departure in Shared Object Middlewares

Yuanyuan Zhou and Kai Li
Computer Science Department
Princeton University, Princeton, NJ 08544, U.S.A.
{yzhou, li}@cs.princeton.edu
phone: (609)258-1795
FAX: (609)258-1771

## Abstract

Collaborative applications are increasingly being deployed to support interactions among widely distributed users. Replication of state that enables interaction among distributed users can be used to provide acceptable repsonse time in the presence of high communication latencies. Recent research has suggested using shared objects as the middleware for developing collaborative applications to minimize communication traffic. One of the key issues in using shared objects middleware to support collaborative applications is to allow nodes to dynamically join and departure. This paper presents four methods to manage distributed object directories in the presence of dynamic join and departure operations in shared objects middleware: full redistribution, master, indexed hashing and on-demand indexed hashing. We have compared these four schemes with four criteria: even distribution, memory overhead, number of messages and algorithm complexity. Among these four schemes, the two indexed hashing schemes on average are better than the other two schemes.

## 1 Introduction

The advent of the World Wide Web functionality by using Java, Plugins, ActiveX, etc., has enabled a range of collaborative applications such as shared whiteboards, calendars and editors, to be deployed to support interactions among widely distributed users. The interactive nature of such applications requires that the effect of a user's operation is seen by all the participants in a timely fashion. The challenge arises in how to provide fast interactive response time in a wide-area distributed environment, where high latencies are common.

Distributed users in collaborative applications interacts via shared state, a set of shared objects. Recent research [15, 17] has suggested that replication of shared objects has the potential to reduce response time by performing operations on the local copies of shared objects. In addition, replication can also reduce communication traffic by batching and propagating local updates to other users on demand. However, it introduces the problem of replica consistency. A natural solution to this problem is to extend conventional distributed shared virtual memory or shared objects systems (DSM) to support collaborative applications on the wide area network.

Shared virtual memory [14] or shared object approaches [1, 2] can provide a shared single address space on a network of computers without physically shared memory. These systems maintain memory coherence transparently to applications. Although shared memory is an attractive programming model, it has not been used for distributed collaborative applications because of lacking the ability to allow nodes to dynamically join and departure.

Dynamic join and departure are indispensable operations for distributed collaborative applications. Since collaborative applications such as multiple-party conferencing, editing and gaming require the ability to allow participants to dynamically join or withdraw, the shared objects middleware needs to provide such support.

The key issue in implementing dynamic join and departure operations is to deal with the directory information. The state-of-the-art conventional DSM systems all use distributed directories to maintain memory consistency. Upon a user departure, the directory information on the leaving participant needs to be transferred to remaining participants to avoid losing information. When a user joins, it needs to have a way to access the directory information correctly and efficiently. A good strategy to implement dynamic join and departure should keep the directory information evenly distributed to avoid communication bottlenecks.

This paper presents four methods to manage distributed object directories in the presence of dynamic join and departure operations with shared object middlewares: full redistribution, master, indexed hashing and on-demand indexed hashing. The full redistribution scheme can maintain even distribution, but the join and departure operations are expensive and requires many messages. The master approach can reduce the complexity and the number of messages for join and departure operations, but it requires more memory and does not have the even distribution property. The two indexed hashing schemes work better than the other two schemes when all factors are considered. Compared with the basic indexed hashing scheme, the on-demand scheme can reduces the number of messages for join operations with the cost of the uneven distribution and possible 3-hop messages for locate operations.

## 2 Background

Shared object middlewares maintain memory coherence mostly at object granularity [14]. They use directories to keep track of the information about each object in the shared object space. An object can be marked shared-read or exclusive-write. To implement the sequential consistency [13], for example, each object can have multiple copies if it is marked shared-read whereas it can have only one exclusive copy if it is marked exclusive-write. Read and write accesses trigger state transitions and object movements to maintain the replica consistency.

The core data structure in shared object middlewares is the directory. It contains an entry for each object to maintain memory consistency among distributed replicas. For sequential consistency, each entry consists of the following fields:

- state: indicates whether the object is null, shared-read, or exclusive-write.
- owner or home: indicates which location has (or possibly has) the most up-to-date copy of the object.
- copyset: keeps track which locations have a shared-read copy.

For relaxed consistency models, each entry will also contain information such as vector timestamps, write notices and so on. A coherence operation (such as a read or write access ) will read and/or modify the entry associated with the shared object. Consider the sequential consistency protocol as an example, when reading an object in the null state, a participant will use the directory entry to find out where to get a copy and add itself into the copyset. Before writing to an object in the shared-read state, a participant will access the directory entry to invalidate all the read copies of the object and change the state of the object to write-exclusive state. If the entry is not in the local memory, it will read or modify the entry in the remote location using messages.

The directory can be centralized, fully replicated or distributed. Although the centralized approach, which keeps a single copy of the directory on one location, is simple, it requires lots of messages going to the central location, causing a bottleneck. The fully replicated approach requires more memory resources and broadcast hardware to be efficient. Therefore, it is not practical for collaborative applications on a wide area network. The preferred approach is to distribute entries among multiple locations, because it can balance network traffic and remote services, and reduce memory overhead. A previous paper provides detailed comparisons of various approaches in conventional DSM systems [14].

A common way to locate a distributed directory entry for a shared object is to use a simple hashing function:

$$locate(b) = b \bmod n.$$

where $b$ is the object number (or address) and $n$ is the number of participants in the system. This approach assumes that each participant in the system has a unique id between 0 and $n-1$. This approach takes constant time to find out the location holding the entry for the given shared object, requiring at most one message. It requires no extra memory space for the directory because each entry only has one copy. It can evenly distribute all directory entries among participants in the absence of dynamic join and departure. Most of conventional DSM systems use this approach.

## 3 The Problem

The question addressed in this paper is how to manage distributed directories in the presence of dynamic join and departure with shared object middlewares. When a user departs, all entries located in this user's machine need to move to other participants. When a user joins, it needs to offload some entries from the old participants in order to avoid the situation where all entries are merged into a small set of participants after a series of joins and departures.

Obviously, a straightforward approach with the simple hashing scheme described in Section 2 does not work for dynamic join and departure operations without modifications. For example, initially we have participant 0,1,2, and 3 . If participant 2 departs, not only all entries on participant 2's machine need to be transferred to the others, but some entries on the remaining participants also need to be redistributed because the hash function $locate(b) = b \bmod 4$ has been changed to $locate(b) = b \bmod 3$. The entry for the object 7, which was initially at participant 3 according to the old hash function, now hashes to participant 1 according to the new hash function. The same situation occurs when a user joins. Therefore, we need to consider solutions to take care of three operations: locate, join and departure.

To evaluate a method, we consider four factors:

- **Even distribution** Whether the directory entries are evenly distributed implies how well the network traffic and the workload of remote services are balanced. In practice, a conventional DSM system with a centralized directory performs substantially worse than a distributed approach.

- **Number of messages**. Shared object middlewares use message passing to transfer data and control information among distributed users. Since wide-area distributed environment has high latencies, it is important to reduce the number of messages when performing operations.

- **Memory overhead**. System memory overhead has significant impact on the scalability of the middleware. If the system data structure occupies too much memory, it will limit applications' problem sizes, and it might also degrade the overall performance by polluting the cache and/or the physical memory.

- **Complexity of the algorithm**. In a shared object middleware, the locate operation is a frequent operation. It is used each time a coherence operation is invoked; it is very important to keep the access time as little as possible. Although join and departure operations are not as frequent as locate operations, it should be fast because it determines the responsiveness of an application.

We ignore the amount of data sent for each operation in this paper. The main reason is that how much data to be transferred varies with different consistency protocols and different applications. The amount of data transferred dominates the amount of directory entries.

## 4 Solutions

This section presents four solutions to manage distributed directories in the presence of dynamic join and departure operations with shared object middlewares: full redistribution, master , indexed hashing, and on-demand indexed hashing.

For convenience, we use the following notations:

- $m$: the number of directory entries.
- $n$: the number of participants.

## 4.1 Full Redistribution Scheme

The *full redistribution* scheme fixes the problem of the old hashing algorithm for dynamic join and departure operations by reshuffling the entries according to the new hash function after a participant joins or departs. It first rehashes all the entries using the new hash function. If the new location for an entry is different from the old location, the entry will be moved to the new location using messages. In a real implementation, all entries from the same old location to the same new location can be packed and sent together with one large message. Figure 1 shows the directory distribution before and after participant 2 joins. Entry 3 is initially at participant 1 before the join operation and it is moved to participant 0 after participant 2 joins in. Departure operations work in a similar way.

This simple modification can keep the directory entries distributed evenly among all participants in the presence of joins and departures. Each locate operation still takes constant time and requires at most one message. The main drawback of this scheme is the intensive overhead for dynamic join and departure operations because a full reshuffling

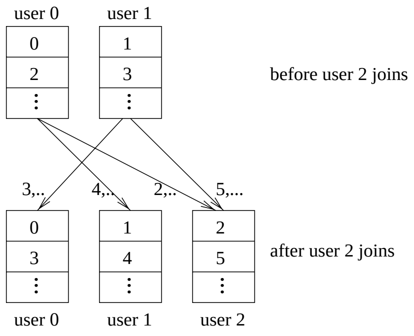

Figure 1: A example of a join operation in full redistribution scheme(arrows indicate messages, numbers on arrows indicate the entries sent by messages)

is expensive and requires a lot of messages. The analytical complexity and number of messages are summarized in the following simple theorem.

**Theorem 1** *The full redistribution scheme has the complexity of* $\Theta(m)$ *and requires* $\Theta(n^2)$ *total number of messages for join and departure operations.*

**Proof** The complexity part is obviously true since all the entries in the directory have to be rehashed. One can prove that the scheme requires $\Theta(n^2)$ total number of messages by showing that at least one entry needs to be transfered between any two participants. Suppose $n_1$ and $n_2$ are any two arbitrary participants and $n_1 \geq n_2$, then object $(n_1 - n_2)n + n_1$ is initially rehashed at participant $n_1$. After the $(n+1)$th participant joins in, the new hash function should hash this object at participant $n_2$ since $(n_1 - n_2)n + n_1 = (n_1 - n_2)(n+1) + n_2$.

Dynamic join and departure operations are very expensive with the full redistribution scheme. In most of the applications, the number of entries can be very large (can be up to $2^{20}$). The $\Theta(n^2)$ total number of messages can also become a problem when we have a lot of participants(for example, $n = 64$), which is usually the case in collaborative applications.

## 4.2 Master Scheme

The *master scheme* combines the ideas of the centralized and distributed directory approaches. In this scheme, a participant is assigned to be the master. The master has a master table containing the locations of all the entries. When a participant joins, no action is taken to maintain the directory. When a participant departs, only the entries on the departing participant need to be sent to new locations and all other entries will remain at old location. When a participant accesses an entry, it sends a message to the new location using the new hash function, and if the new location does not have the entry, it requests the master to fetch it from the old location, and the master changes the corresponding entry in the master table to point to the new location. Future accesses to the same entry are handled by the new location.

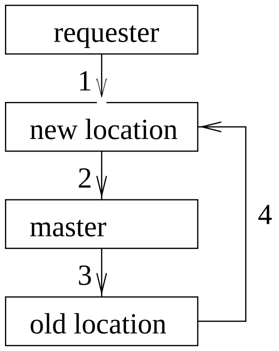

Figure 2: Four-hops procedure in the master scheme

Figure 2 shows the four hops procedure to access an entry which is still in the old location. When the master departs, a new master will be assigned, and the old master sends the master table to the new master.

In the master scheme, the overhead of dynamic join and departure operations is relatively small. Join operations have complexity of $O(1)$ and require no message overhead. The complexity for departure operations is $O(1)$ in the best case and $O(m)$ in the worse case, where $m$ is the number of entries. The best case occurs when a participant joins and departs immediately. Uneven distribution results in the worst case. For example, initially, participant 1 is the only participant, so participant 1 holds all the entries. Then other participants join the computation. And if no entries are moved from participant 1 to new locations because they are never accessed, participant 1 needs to shift all the entries to others when it departs.

This scheme has four disadvantages:

- Directory entries distribute unevenly among participants. Some locations may contain most of the entries while others only have a few. As we have observed in many systems, uneven distribution can significantly degrade performance.

- Locate operation requires a four-hop message when the entry is not at the new location, though future accesses need at most 1 message.

- There can be a bottleneck at the master. It may limit the scalability of the system.

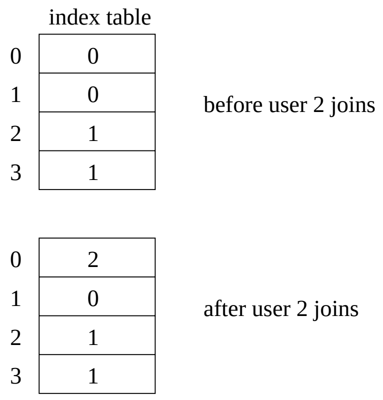

Figure 3: The index table before and after user 2 joins

- the master table may occupy too much memory, and it is proportional to the number of directory entries.

## 4.3 Indexed Hashing Scheme

### Main Idea

The main idea of the *indexed hashing scheme* is to use a hash function independent of the number of participants while still ensuring even distribution of the directory. To locate the entry for a given shared object, the object number is hashed into an index $i$ in a small table called *indexed table* using a universal hash function $h$. The $i$th element in the indexed table points to the location with the directory entry. In other words, for a given object $b$, participant $indexTable[h(b)]$ holds the directory entry. For convenience, we use $N$ to denote the maximal number of participants the middleware supports. The index table has $N$ entries. Figure 3 shows an indexed table $t$ with $N = 4$ and 2 participants.

When a user joins, some entries in the indexed table are changed to point to the new participant. When a user departs, all the entries pointing to the departing user are changed to point to other participants. Figure 3 shows an example of how the indexed table is changed after user 2 joins.

### Data Structures

The main data structures include an index table, a participant list and two pointers. Every participant has a copy of these data structures. All copies are kept identical by running the exact same algorithms on all participants every time a user joins or departs.

- **Index Table**
  All entries pointing in the index table to the same location are linked together. Each entry has two fields:
  - next: a pointer to the next index table entry mapped to the same participant;
  - location: a pointer to the location in the participant list.
- **Participant List**.
  All participant structures are linked together as a ring. Each participant structure has the following fields:
  - id: the sequence number assigned to the participant;
  - load: the workload of the participant, or the number of entries in the index table mapped to this location;
  - first: a pointer to the first entry in the index table mapped to this location;
  - next: a pointer to the next participant.
- **L Pointer** *lp*.
  *lp* points to the first participant with lighter workloads in the participant list.
- **H Pointer** *hp*.
  *hp* points to the first participant with heavier workloads in the participant list.

### Initial State

Initially, user 0 is the only participant, and both *lp*, *hp* and every entry on the index table point to user 0.

### Dynamic Join Algorithm

When a participant receives a new participant's join notification, it runs the join algorithm to change the index table and decides which entries should go to the new user. The join procedure goes through the elements in the participant list starting from the one pointed by *hp* until the workloads are balanced. Each iteration changes one entry in the indexed table to point to the new location. After the algorithm finishes, a link list called *moving list* contains all the entries which need to send to the new location from the current location. The pseudo-code for the algorithm is as follows:

```
1. repeat until the work load of participant $a$ reaches the minimum workload ($lp \to load$)
    (a) remove the first entry $e$ from the entry list of the participant pointed by $hp$;
    (b) add $e$ to $a$'s entry list;
    (c) $a \to load++$;
    (d) $hp \to load--$;
    (e) if $hp$ points to the current location running this algorithm, move $e$ to the sending list;
    (f) $hp = hp \to next$;
2. move all directory entries which are hashed to indexes contained in the moving list to new location $a$;
3. add $a$ into the participant list right before $hp$.
```

### Dynamic Departure Algorithm

When a user *a* departs, all remaining participants modify their index tables by performing the following departure procedure. The algorithm starts from the participant pointed by *lp* and distributes all the entries that point to the departing location to other participants in a round-robin fashion. The algorithm works as follows:

```
1. remove $a$ from the participant, if $hp == m$, $hp = hp \to next$; and if $lp == m$, $lp = lp \to next$;
2. repeat until the work load of participant $a$ equals 0
    (a) remove the first entry $e$ from participant $a$'s entry list;
    (b) add $e$ to the entry list of the participant pointed by $lp$;
```

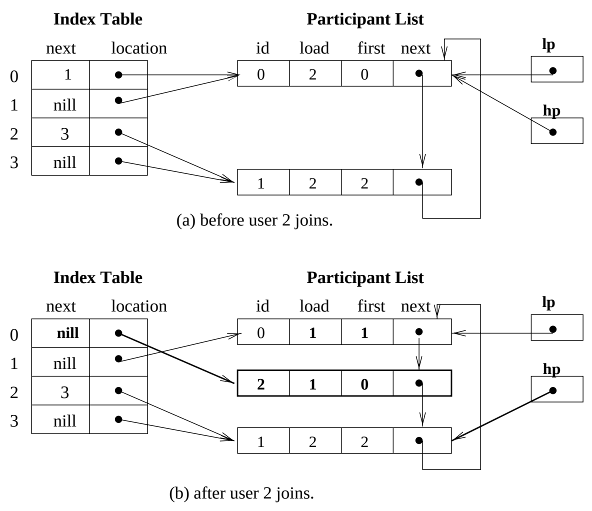

Figure 4: An example for the dynamic join algorithm in the indexed hashing scheme(darker numbers and arrows are changed values).

(c) $a \rightarrow load - -$;

(d) $lp \rightarrow load + +$;

(e) $lp = lp \rightarrow next$;

3. if I am departing, send directory entries to other remaining participants, otherwise wait for entries from user $a$

Figure 4 shows an example of how the indexed table and the participant list change after a user joins.

## Properties

The indexed hashing scheme takes constant time to locate an entry. The memory overhead for this scheme is small since we only need memory space to hold the small indexed table and the participant list. For $N = 1,024$, the maximal memory space needed for this scheme is less than 16K bytes.

One of the main advantage for this scheme is it can evenly distribute directories among participants. The following theorem states the property.

**Theorem 2** *In the indexed hashing scheme, the workload of any participant is either* $\lfloor \frac{N}{n} \rfloor$ *or* $\lceil \frac{N}{n} \rceil$*, where* $N$ *is the maximal number of users the system supports.*

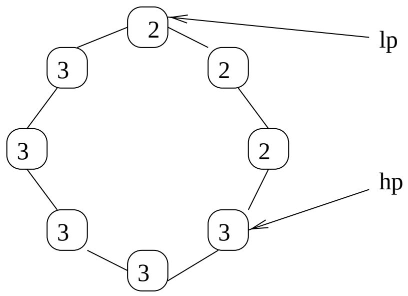

Figure 5: An example of the participant list

To simplify the problem, we only look at workloads in the participant list. Figure 5 shows the participant list for 8 participants. Each circle represents a user, The numbers inside the circle is the workload. $hp$ and $lp$ point to users in the participant list.

To prove theorem 2, it suffices to prove the following lemma.

**Lemma 1** *All the users in the path from* $lp$ *to the predecessor of* $hp$ *have workloads of* $\lfloor \frac{N}{n} \rfloor$*, while the others have workloads of* $\lceil \frac{N}{n} \rceil$*.*

The rigorous proof of the lemma can be done by induction. The idea of the proof, however, is not difficult. We can prove it by showing that

1. $hp$ always points to the first user with heavier workloads;

2. $lp$ always points to the first user with lighter workloads;

3. if $hp \neq lp$, then $hp \rightarrow load = lp \rightarrow load + 1$;

4. $hp$ and $lp$ break the participant list into 2 lists: $l_1 = [hp, \ldots, lp)$ and $l_2 = [lp, \ldots, hp)$. Any users in $l_1$ has the same workload as $hp$, and any users in $l_2$ has the same workload as $lp$.

All the four properties are obviously true with the initial state. They also hold after every iteration of the join and departure algorithms. In the join algorithm, each iteration moves the first entry mapped to user $hp$ to the new participant, and $hp$ then moves to the next user in the list. Now the user pointed by the old $hp$ has the same workload as $lp$, and the user pointed by the new $hp$

will be the first one with heavier workloads. And the workloads for all other users are unchanged. Therefore, the four properties are still true. Similarly, in the departure algorithm, each iteration moves one workload from the leaving user to the user pointed by *lp*, and *lp* slides down to the next one in the list. So the user pointed by the old *lp* has the same workload as *hp*, and the user pointed by the new *lp* will be the first one with lighter workloads.

After we proved the even distribution property of the indexed hashing scheme, we can easily figure out the complexity for join and departure operations. When a user joins, every iteration of the algorithm moves one unit of workload from an old location to the new location. And the algorithm finishes when the workload at the new participant reaches the same workload as *lp*, which is $\frac{N}{n}$. Therefore, only $\frac{N}{n}$ iterations are executed. Since every participants sends message only to the new participant, the total number of messages is $O(n)$. Similarly, when a user departs, each iteration moves one workload from the leaving user to other participants. Since the leaving location has $[\frac{N}{n}]$ workloads, the algorithms finish after $[\frac{N}{n}]$ iterations. In summary, we have the following corollary.

**Corollary 1** *The cost of a join or departure operation with n participants is $\Theta(\frac{N}{n})$. The total number of network messages introduced by a join or departure operation is $O(n)$.*

### 4.4 On-Demand Indexed Hashing Scheme

On-demand indexed hashing scheme is a modification to the basic indexed hashing scheme. It has the same departure algorithm as the basic scheme. The only difference from the basic scheme is that the on-demand scheme postpones the redistribution of the entries until the first accesses. After the join algorithm finishes, all entries which need to be moved to the new location from the current location are stored in the moving list. When the new participant receives a message requesting a directory entry which is still in the old location, 3-hop messages are paid to fetch from the old location all directory entries hashed to the same entry in the index table. To achieve this, a special data structure is needed in order to keep track of the old locations for the unmoved entries.

The complexity for the dynamic join and departure of this approach is the same as the basic approach. But a join operation needs not send any message. The tradeoff is that it may require 3-hops messages to locate the entry. Similar to the master scheme, it does not maintain even distribution among participants. The memory overhead is reasonable since the table keeping old locations are distributed among all locations.

### 4.5 Summary

Now, we briefly summarize this section by comparing all four methods to manage object directories in the presence of dynamic join and departure operations with shared object middlewares. Table 1 illustrates the difference of the four solution using the four criteria described in Section 3.

As shown in Table 1, the full redistribution scheme has the highest complexity and number of messages for join and departure operations. For the master scheme, the uneven distribution and the 4-hop messages for the locate operation can degrade the system performance. Moreover, the significant amount of memory overhead and the bottleneck caused by the master make this method unscalable. When all factors are considered together, the two indexed hashing scheme work better than the other two. The on-demand scheme can reduces the number of messages for join operations with the cost of the uneven distribution and possible 3-hop messages for locate operations.

## 5 Related Work

Many DSM systems have been developed in the past to provide parallel applications with a shared memory or shared object abstraction. An early paper [14] introduced the concept of shared virtual memory and compared several centralized and distributed directory approaches in the context

| properties | | full redistribution | master | indexed hashing | on-demand |
|---|---|---|---|---|---|
| distribution | | even | uneven | even | uneven |
| memory overhead | | 0 | big | small | medium |
| number of messages | locate | 1 | 1 or 4-hops | 1 | 1 or 3-hops |
| | join | $\Theta(n^2)$ | 0 | $O(n)$ | 0 |
| | departure | $\Theta(n^2)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| complexity | locate | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| | join | $\Theta(m)$ | $O(1)$ | $\Theta(\frac{N}{n})$ | $\Theta(\frac{N}{n})$ |
| | departure | $\Theta(m)$ | best $O(1)$;worst $O(m)$ | $\Theta(\frac{N}{n})$ | $\Theta(\frac{N}{n})$ |

Table 1: Comparison($m$ is the number of directory entries, $n$ is the number of participants and $N$ is the maximal number of participants the system supports)

of sequential consistency. Recent shared virtual memory systems [3, 11, 4, 16, 18, 12] used various relaxed consistency models. Shared object systems [1, 2, 9] maintain memory consistency in the same way as the shared virtual memory approach except at object level. None of these systems, to the best of our knowledge, is used for distributed and collaborative applications because of lacking the ability to allow nodes to dynamically join and departure.

There are several recent proposals to use shared object or replicated objects for Internet collaborative applications such as discussion forums, collaborative editors and Internet games. For example, Objent has provided a framework for high-end collaborative applications [15]. [17] has proposed and analyzed a dynamic replication algorithm. They did not describe how to support dynamic join and departure operations.

Other distributed systems has also encountered the directory management problem for dynamic join and departure operations. The global memory management project [8] has a centralized method similar to the master scheme in this paper. It mentioned that when a new workstation adds in, each machine distributes appropriate portions of the global-cache-directory to the new workstation; and when a machine departs, the global-cache-directory is also redistributed. However, no further detail about the join and departure algorithms has been given.

Indexed extendible hashing is a modification to the extendible hashing scheme. Indexed extendible hashing and extendible hashing are mainly used in database systems or files systems to allow hashing tables to grow or shrink dynamically without sacrificing small cost of the retrieval time [7, 6, 5]. We use the indexed hashing method to manage object directories in the presence of dynamic join and departure operations in shared object middlewares.

Universal hashing function for shared memory was analyzed in [10]. But it did not consider dynamic join and departure operations.

## 6 Conclusions

This paper presented four methods to manage distributed object directories in the presence of dynamic join and departure operations with shared objects middleware: full redistribution, master, indexed hashing and on-demand indexed hashing.

We have compared these four schemes with four criteria: even distribution, memory overhead, number of messages and algorithm complexity. Among these four schemes, the two indexed hashing schemes on average are better than the other two schemes. Compared with the basic indexed hashing scheme, the on-demand scheme can reduce the number of messages for join operations with the cost of the uneven distribution and possible 3-hop messages for locate operations. The full redistribution scheme can maintain even distribution but the join and departure operations are expensive and requires many messages. Although the master scheme can reduce the complexity and the number of messages for join and departure operations, but it loses the even distribution property and the memory overhead and the centralizer limits its scalability.

*Note to reviewers: We are working on our implementation to evaluate the four schemes. The experimental results will be included in our final version, if the paper is accepted.*

## References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[3] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Seattle, Washington, March 1990.

[4] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93 Conference*, February 1993.

[5] Soon M. Chung. Indexed Extendible Hashing for Databases,. Technical Report WSU-CS-91-02, Washington State University, 1991.

[6] Carla Schlatter Ellis. Extendible Hashing for Concurrent Operations and Distributed Data. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.*

[7] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing — A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[8] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 201–212, December 1995.

[9] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[10] Anna R. Karlin and Eli Upfal. Parallel Hashing—An Efficient Implementation of Shared Memory (Preliminary Version). In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 160–168, Berkeley, California, 28–30 May 1986.

[11] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[12] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michal Cierniak, Srinivasan Parthasarathy, Wagner Meira, Sandhya Dwarkadas, and Michael Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. *Proc., 24th Annual Int'l. Symp. on Computer Architecture*, June 1997.

[13] Lesile Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[14] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986. A revised version appeared in *ACM Transactions on Computer Systems, 7(4):321–359, November 1989.*

[15] K.Schwan F. Bustamante T. Rose M. Ahamad, R. Das and D. Zhou. Objent: A Framework for High-end Collaborative Applications. Technical report.

[16] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, SOftware-Only Approach for Supporting Fine-Grain Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[17] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, June 1997.

[18] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.