Yale University
Department of Computer Science

Posters Presented at the Twenty-Fourth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing

Rida Bazzi and James Aspnes, editors

YALEU/DCS/TR-1328
July 17th, 2005

FOREWORD

This technical report contains the papers that were presented in the poster session of the ACM Symposium on Distributed Computing that was held on July 17-July 20 2005 in Las Vegas, Nevada. The goal of the poster session is to provide another venue for presentation of results that might not be appropriate for the long track or brief announcement tracks, but that nevertheless have the potential of generating interesting discussions and exchanges of ideas.

The posters were screened by the poster committee, but were not formally reviewed. Moreover, this informal collection is not intended to serve as a publication. It is expected that these posters will be published in the future in conferences and journals.

Rida Bazzi (Posters Chair)
James Aspnes (Program Chair)

TABLE OF CONTENTS

An Optimization of the Buddy Model for Securing Mobile Agents[†]

Yueh-Hua Lee and Hyunyoung Lee
Department of Computer Science, University of Denver
Denver, CO 80208, U.S.A.
{ylee7, hlee}@cs.du.edu

1 Introduction

This paper proposes a scheme to improve a security model, namely the "buddy model", for an agent community. Mobile Agents are autonomous programs that can migrate from hosts to hosts. A mobile agent can decide when and where to move. When migrating, a mobile agent brings the implementation, data, and execution state with it. The concept of agent community results from the maturity of mobile agent technology. The agent community is a Multi Agent System (MAS) with a specific community goal [1]. A community integrates different mobile agents that perform different functions in order to achieve one common goal. An agent community is a dynamic entity: Mobile agents join and leave their community frequently. Furthermore, an agent community may consist of several sub-communities with different secondary community goals; several communities may form a super-community that has a primary community goal. It is natural that an agent community is designed in a hierarchical way; however, hierarchical implementation brings vulnerabilities. Attackers can easily target the mobile agents in the top level of the hierarchy or create malicious mobile agents that assume the identities of administrative mobile agents in order to ruin the whole community.

The "buddy" model is proposed by Page et al. [1, 2]. The term "buddies" refers to the neighbors of a mobile agent in a pre-assigned group. In the model, the mobile agents within a group generate tokens and send the tokens to their buddies periodically. By using the tokens, they protect buddies and monitor the health condition of buddies. Page et al. argue that the model avoids the vulnerabilities of a hierarchical scheme for mobile agent security: Because each mobile agent performs an identical role in the security function, it is hard for an attacker to find and attack a central coordinator.

This paper explores an improvement of the "buddy" model. The *optimization issue* should be taken into account when the model is applied to a large-scale scenario. If the mobile agents move far away from one another, the network suffers from their frequent token delivery. We try to reduce the network traffic by grouping nearby agents together dynamically. When a mobile agent migrates to a far away host, the agent leaves the old group and joins a new nearby group. We also note that the size of a group dynamically changes in time. Our improvement scheme provides means to merge two small groups into a larger group and to split a large group into two or several smaller groups. The scheme can also be applied to other MAS if the system can be divided into groups and requires frequent communication between group members. A system gains better performance because the group members that need to communicate are geographically near to one another.

There are five algorithms in our optimization approach. The "Join", "Leave", and "Accept" algorithms collect nearby mobile agents into a group. Our "Merge" and "Split" algorithms maintain reasonable size and diameter of a group by using a temporary central coordinator.

2 The Algorithms

We assume an asynchronous network with arbitrary topology. A security group is represented by an undirected graph $S(V, E)$. Each vertex represents a mobile agent and each edge represents buddy relationship. In other words, if two mobile agents are buddies, there is an edge that connects them. We assume that the mobile agents have ability to compute the distance between any two mobile agents. The distance is measured by the number of hops between them. We define D as the maximum diameter of a group. The size of a

[†]Full version is available from `www.cs.du.edu/~hlee/Research/pub.html`

group is between L and U where L is the lower bound and U is the upper bound. We choose $L < U/2$ to avoid iterative merge and split.

When a mobile agent broadcasts a request, it waits for a threshold T of time. $T < 2\delta$ where δ is the maximum message delay for a communication channel whose length is D. If a reply message cannot arrive within T time period, we assume that the source of the reply message is unable to respond or is far away from the mobile agent that sends the broadcast message. However, we have to synchronize mobile agents that dock at different machines. The threshold T is measured by clock ticks. When a mobile agent moves to a new host, it sends a synchronization request to its home platform (the home platform is the origin of the agent). The home platform will send two replies, which are "start" and "end", and the time between the two replies is T. The mobile agent counts the number of local clock ticks, which is T', between receiving "start" and "end". Using this approach, T is evaluated by T' and mobile agents on different hosts are synchronized. The system parameters T, D, L, and U are known to every agent.

In the "Merge" and "Split" algorithms, we use a temporary coordinator to lead the merge and split processes. We assume that a randomized leader election algorithm can choose a unique leader at random. Therefore, there is still no way for an attacker to find and attack the temporary coordinator. Ramanathan et al. propose a randomized leader election algorithm that works in arbitrary network topology and requires only $O(n)$ messages where n is the number of processes [3].

Each mobile agent knows the addresses and identities of its direct buddies and buddies of buddies. The information is stored in an array, *Neigh[]*. Each entry in *Neigh[]* has three fields. The first field records the identity; the second field records the address; the third field indicates the agent is a direct buddy or buddy of a buddy. If a mobile agent senses that the size of the group or *Neigh[]* changes, it appends the information to its token. Upon receiving the token, the buddies modify their local information according to the token. If two variables have the same name, the term "this" refers to the local variable of a mobile agent.

Join, Accept, and Leave. When a mobile agent p moves far away from its buddies and decides to join a new group, p broadcasts a "join request" and waits for T time to receive "accept messages" form q. The "accept message" contains the size of q's group and the distance between p and q. When the timeout occurs, p chooses the nearest agent q. If there are two or more agents that have the same distance and the distance is the smallest, p chooses the agent that resides in a smaller group. If the sizes of the groups are also the same, p chooses one of them arbitrarily. Then, p sends a confirmation, to the nearest agent q and receives *Neigh[]* of q. After all, p joins the group, which q resides in, and leaves the old group. If no accept message is received, p ceases its work and returns to the home platform.

In the Accept algorithm, an agent p computes the distance from p to q when it receives a join request from q. If the distance is smaller than D, p sends an accept message to q. When p receives a confirmation, q becomes a buddy of p. At the same time, p appends *size_group* and *Neigh[]* in the token. When p's buddies receive the token, they update their *size_group* and *Neigh[]* accordingly.

Merge. When a group becomes smaller than L, it should merge with a nearby group. In order to have efficient communication among the buddies, we want to minimize the total path lengths of the new group. There are three phases in the Merge algorithm. In phase one, the mobile agents elect a temporary coordinator C at random. In phase two, each mobile agent p in the group H finds an agent u that belongs to another group G_i and is nearest to p. We use the same approach in the Join algorithm to find such agent u: p broadcasts a message and waits for T time. Other agents who receive the message report the distance and *size_group*. If there are two or more mobile agents that report the same distance and the distance is the smallest, the one with smaller *size_group* is chosen. If *size_group* are also the same, p chooses arbitrarily. Then, p finds the agent w, which is the farthest agent of p in G_i. Agent p queries all agents in G_i to find such w. Then, p computes the distance d between p and w. We call d the "longest distance", which indicates the diameter of the new group after merging. Next, p reports d, G_i and the size of G_i to the coordinator C.

In phase three, C chooses the group G among G_i according to d and leads the merge process. Again, if there are more than two groups that have the same d and such d is the smallest, C compare the sizes of the groups and chooses the one with smaller size. If the tie situation happens again, C chooses arbitrarily. The merge process is to gather addresses of all agents in H and G, assign buddies for all agent, and update local variables of all agents. Application developers decide how to assign buddies. The subroutine Gather_Address uses a depth-first search to gather addresses of all mobile agents. When gathering addresses, the subroutine

uses an array, *All_Address[]*. Each entry in *All_Address[]* contains two fields: The first field is the identity of mobile agents and the second field is the address.

Split. The Split algorithm consists of three phases. In phase one, we randomly elect a temporary coordinator C and build a spanning tree over the mobile agents where C is the root. Each node represents a mobile agent and has one parent variable and several child variables. Parent/child stores the identity of parent/child node respectively. The subroutine Build_Tree is a breadth-first algorithm. Every mobile agent sends message $\langle$*your_parent, identity*$\rangle$ to its direct buddies. The buddies reply message $\langle$*your_son, identity*$\rangle$ or $\langle$*not_your_son*$\rangle$ to indicate whether the buddy is a child of the mobile agent that sends $\langle$*your_parent, identity*$\rangle$.

In phase two, we compute *size_subtree*, *split_factor*, *flag_split_candidate*, and *num_split_candidate*. Computing *split_factor* and *flag_split_candidate* requires no external messages. On the contrary, computing *size_subtree* and *num_split_candidate* at an agent p requires the *num_split_candidate* and *size_subtree* of all children of p. The approach is straightforward. If the node is a leaf node, the *size_subtree* is one; otherwise, the *size_subtree* is the sum of *size_subtree* of all children plus one. If the node is not a candidate, the *num_split_candidate* is zero; otherwise, the *num_split_candidate* is the sum of *num_split_candidate* of all children. Of course, each node other than the root should report *size_subtree* and *num_split_candidate* to the parents.

A split candidate (or candidate) is a node whose size of subtree is greater than or equal to L. Obviously, the coordinator C must be a candidate. The algorithm splits the group by cutting edges of the tree. If there is only one candidate, which is the coordinator C, we preserve the edge to the child whose *size_subtree* is the smallest. If a tie situation occurs, C chooses arbitrarily. All other edges between C and its children are cut. If the new groups are too small, they run the Merge algorithm. If there are two candidates, we cut the edge between them. If there are three or more candidates, the algorithm chooses according to *split_factor*, where $split\_factor = |size\_subtree - (size\_group/2)|$. It makes the sizes of two new groups roughly half of the original group. In phase three, C gathers addresses, assigns buddies for all mobile agents and update their local variables.

Failure and Malicious Agents Handling. In our scheme, an agent waits for T time after sending a request. The agent ignores all replies after the timeout occurs. In the Merge and Split algorithm, we need to gather all addresses using subroutine, Gather_Address. Because we maintain the information of direct buddies and buddies of buddies in *Neigh[]*, Gather_Address fails only when all direct buddies and buddies of buddies fail. The redundant information makes the algorithm robust.

The buddy model avoids hierarchical vulnerabilities because all mobile agents act an identical role with respect to security function. It is undeniable that the temporary coordinator in the Merge and Split algorithms is more important than other mobile agents. We exploit randomized leader election algorithms to hide the coordinator from the outsiders. However, a malicious mobile agent inside a group can blindly claims itself as the winner of the election and become the coordinator. We employ a variation of Ramanathan's algorithm [3] to solve the problem: in the competition, each contender must act for another randomly-chosen contender. When a contender wins the election, the leader is the mobile agent that the winner represents. Under this condition, a malicious mobile agent becomes the coordinator only if the mobile agent that acts for it wins the election and the malicious mobile agent has no control of its representer. Thus, the algorithm can resist the malicious mobile agent.

References

[1] J. Page, A. Zaslavsky, and M. Indrawan. *"A buddy model of security for mobile agent communities operating in pervasive scenarios"*. Proc. of the Second Australasian Information Security Workshop (AISW2004), 32:17–25, 2004.

[2] J. Page, A. Zaslavsky, and M. Indrawan. *"Countering security vulnerabilities using a shared security buddy model schema in mobile agent communities"*. Proc. of the First International Workshop on Safety and Security in Multi-Agent Systems (SASEMAS 2004), pages 85–101, 2004.

[3] M. Ramanathan, R. Ferreira, S. Jagannathan, and A. Grama. *The "Randomized Leader Election"*. Purdue University Technical Report, http://www.cs.purdue.edu/homes/rmk/pubs/leader1.pdf, 2004.

Network Awareness and Buffer Management in Epidemic Information Dissemination

Ali Alagöz* Emrah Ahi* Öznur Özkasap^[1]
{alialagoz, eahi, oozkasap @ku.edu.tr}

* Department of Computational Science and Engineering
^Department of Computer Engineering
Koc University, Istanbul, Turkey

Abstract. Epidemic algorithms are easy to deploy, robust, and resilient to failure, in contrast to traditional deterministic approaches. Mainly due to their simplicity of implementation and deployment, robustness and scalability, epidemic algorithms have become popular in distributed systems. However, there exist problems such as membership, network awareness, buffer management, and message filtering that should be taken into account during the development of an epidemic information dissemination protocol. In this study, we examine the network awareness and buffer management issues. In a large scale system, any epidemic algorithm that does not address actual network topology may lead to a dramatic increase in the underlying network load. Moreover, to ensure reliability, a loss recovery mechanism exploiting an efficient buffer management technique must be considered. We describe the existing solutions to network awareness and buffer management as well as our approaches, and give our mathematical evaluation and comparative simulation results.

Network Awareness: In a simple epidemic algorithm, all processes are assumed to be equally reachable. It is therefore possible for a process to forward a message to a randomly selected remote one. In a flat membership protocol, partial views are constructed without considering actual network topology. It is easy to implement, but it can not be applied to Internet-wide settings. Hence, network overhead may increase unnecessarily which is a major problem especially in large scale applications. Most of the solutions addressing this problem rely on hierarchical organization of processes that attempts to reflect actual network topology. Information is disseminated to other processes via the hierarchy. Hence, load on the network is limited. The hierarchical model proposed in [3] assumes that in each cluster there are a few nodes having nodes from other clusters in their views. There are only a small number of links among clusters. Parameters of the model, namely the *intracluster fan-out* denotes the number of links each node has with other nodes in the same cluster, and the *intercluster fan-out* denotes the number of remote links each cluster must maintain with nodes outside the cluster.

Our approach to network awareness problem is based on clustering processes. There is a server process in each cluster which is responsible for communication with the other clusters. Therefore, message traffic among clusters is limited to connections constructed by these servers. Our simulation results show that organization of processes in this manner leads to a decrease in the network overhead during epidemic information dissemination. An illustration of our proposed clustered epidemic is given in Fig. 1. Our approach differs from the idea in [3] in fan-out process of information dissemination. In [3], a process chooses targets among the nodes in its own cluster and forwards it to these targets. Information dissemination starts in other clusters only when one of the nodes in that cluster gets the message. This constraint not only leads to latency in information dissemination to whole system, but also more message traffic in the network is generated to achieve reliable information dissemination. In our approach, it is assumed that all the processes in a cluster know the identification of their server. To start information dissemination, a process sends the message to the server firstly. Then classical epidemic information dissemination continues within the cluster. When a server gets a message from its cluster, it forwards it to the other servers. Any reliable epidemic or multicast algorithm may be used for information exchange among clusters. If incoming message is from another server, a server starts epidemic information dissemination in its cluster. Also epidemic algorithm parameters such as fan-out and buffer capacity may differ from one cluster to the other. The parameters can be adjusted with respect to cluster size to achieve a different level of reliability in each cluster independently.

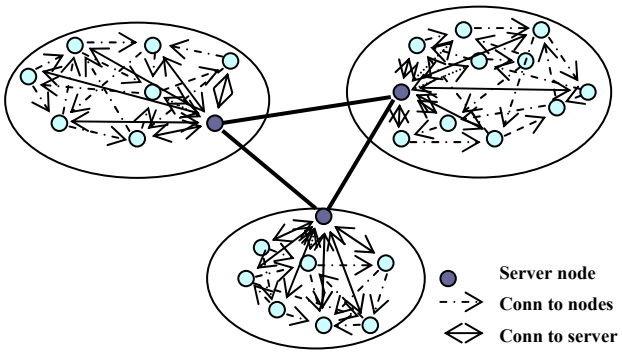

Figure 1. Clustered Epidemic: Clusters are connected by links among servers

[1] *Contact person*. oozkasap@ku.edu.tr, Phone: +90 212 338 1584, Fax: +90 212 338 1548
Department of Computer Engineering, Koc University, 34450 Istanbul, Turkey.

Buffer Management: One approach for buffer management [1] assumes that there is a repair server per cluster that handles message retransmissions. However, if the number of transmitted messages increases, capacity of the server may be exceeded and this leads to a situation in which messages are lost. Another protocol proposed in [2] reduces the amount of buffering by choosing a suitable subset of the group to buffer a given message. In this approach, each member knows the approximation of the entire membership. The approximation needs not be accurate, but it should be of good enough quality that the probability of the group being logically partitioned into disconnected subgroups is negligible. The determination of whether a member should buffer a message is done by using a hash function. The hash function takes the message source and the sequence number of the message as input to determine bufferers. If a node detects a message loss, it determines the bufferers for the lost message via the hash function, chooses one of them randomly and requests the message from it. But this algorithm is not applicable to dynamic groups since dynamic redefinition of hash function in case a new process joins the system is not considered. Moreover, this approach does not consider network awareness.

In this study, we propose an efficient buffering algorithm that ensures reliability in epidemic information dissemination, adaptable to dynamical groups and considers network awareness. The algorithm is similar to the one in [2], but it has no use of a hash function and it targets epidemic information dissemination. For buffer management, when a message is generated, a set of bufferers for the message is determined by the source and ids of these bufferers are piggybacked to the message. Messages are directly forwarded to bufferer processes. These bufferer nodes are determined randomly among the ones message source knows. In addition, the bufferer number can be chosen large enough to handle failures in some bufferers. The bufferer processes hold the corresponding messages in their long term buffers. The long term buffer is used for retransmissions. If a process detects that it has missed a message, it requests the message from one of the bufferers of that message. Determination of lost messages again relies on gossiping. A process periodically chooses *n* processes from its view in a random manner. Then it gets ids and bufferers of messages received by these *n* processes. If it realizes that it has missed some messages, it requests missed ones from one of the bufferers. If that bufferer is crashed or can not retransmit that message, request can be forwarded to another bufferer.

We apply our buffering technique to clustered epidemic where for each cluster there exist a fixed number of bufferers that can be determined by the server of the cluster. When a server gets a message from other clusters, it determines bufferers for that message in its cluster. Then, it directly sends the message to these bufferers. Our algorithms for a server and a message source are given below.

```
Waits for messages
If incoming message is from its cluster
        Disseminate  message to other servers
        If one of servers is crashed
            Buffers message until a new server is
            selected for isolated cluster
                Sends message to failed cluster
If incoming message is from another cluster
        Assigns bufferers for that message in
       the cluster and send messages to  bufferers
       Starts epidemic dissemination in its cluster
```

Algorithm for a server

```
Generates a message
Determines bufferers for the message
If server is failed
        Starts an election algorithm for server
Sends message to server
Sends message to bufferers
Starts epidemic dissemination in the cluster
```

Algorithm for a message source

Simulation Results: We developed a simulation model for our approach for network awareness and efficient buffering as well as two other approaches, flat epidemic and clustered epidemic proposed in [3]. Within the simulation study, we first compare flat epidemic, clustered epidemic [3], and our clustered epidemic in terms of network loads. Secondly, we apply our buffer management technique to flat epidemic and our clustered epidemic model and show how it efficiently handles reliability of dissemination.

In our simulations, nodes in a cluster are fully connected, and partial view of a node includes all the nodes in its cluster. In the flat epidemic model, group size is 90. Nodes are constructed such that links between nodes have weights uniformly distributed between 1 to 15 units. In the clustered model defined in [3], there are three clusters each of with size 30. In each cluster, 3 nodes are responsible for connection to other clusters. We assume that when pivot node gets the message, it sends directly to other two clusters. In our clustered model, again there are three clusters each of size 30. In each cluster there is a server node to connect the cluster to other clusters. Weights of links in a cluster are uniformly distributed between 1 to 5 units, and the links between servers are set to 10 units. We measure the number of transmitted messages and total load on the network. In all three models, a node generates 100 messages periodically, and we calculate number of message losses at the end of the dissemination.

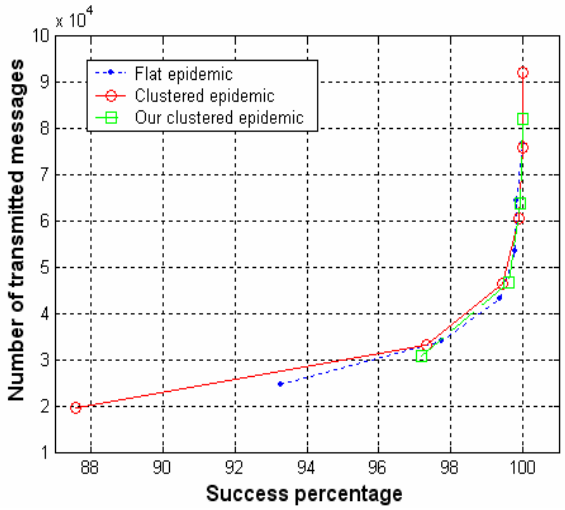

Figure 2. Number of messages transmitted

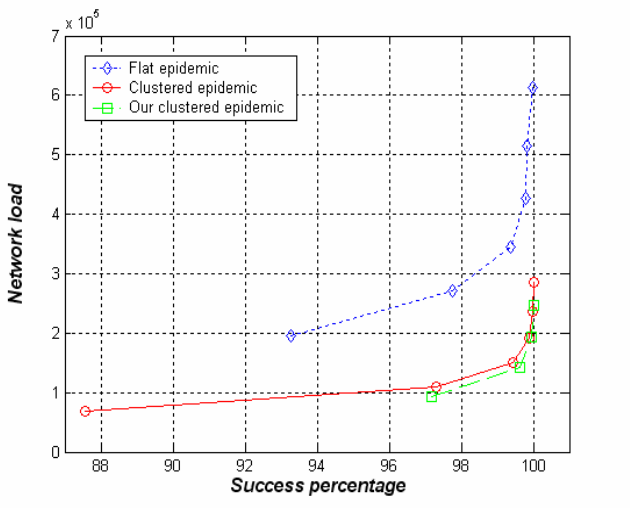

Figure 3. Network load

As illustrated in Fig. 2 and Fig. 3, we observe that to achieve reliability in our clustered epidemic, number of transmitted messages is close to the number for the flat epidemic. However, network load in flat epidemic is approximately 2.5 times of network load in our clustered epidemic. When we compare network loads in clustered model [3] and in our clustered epidemic, network load in our model is equal to approximately 85% of network load of the other. Latency in information dissemination to whole group is also reduced. Therefore, our clustered epidemic improves the network performance of information dissemination significantly.

Then, we test our approach addressing buffering problem using flat epidemic model. We consider a group of size 100 and where source generates 100 messages. We investigate required long term buffer capacity for each process in the group. For each generated message, 2 bufferer processes are chosen from the group. Results are given in Fig. 4 that show the balance of buffering load among processes. For different group sizes (from 10 to 60), we see that mean buffer requirement decreases as group size scales up (Fig. 5). Applying buffering technique to flat epidemic model leads to a 45% decrease in the load of underlying network. We also applied buffering technique to clustered epidemic. It yields approximately 30% decrease in network load of the system with respect to our clustered epidemic model.

Conclusions: Network awareness and buffer management are two major design constraints in developing epidemic algorithms. Approaches presented in this paper, as part of our ongoing work, are easy to implement and suitable to dynamic structures. Our results show that they are scalable and decrease network load dramatically compared with flat epidemic and the other clustered epidemic approaches.

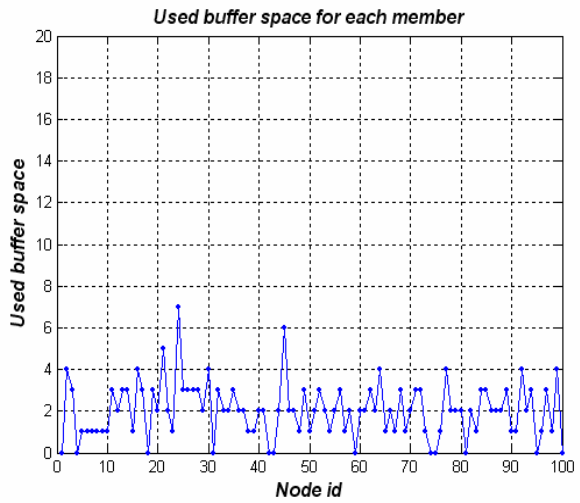

Figure 4. Buffer space

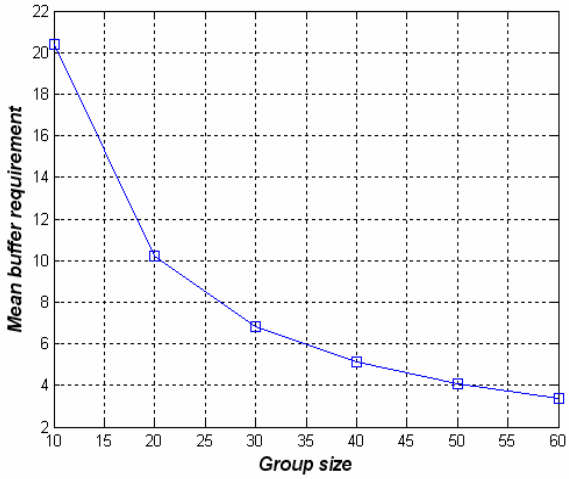

Figure 5. Mean buffer requirement

1. S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya, *Reliable multicast transport protocol (RMTP*), IEEE Journ. on Selected Areas in Communication, special issue on Network Support for Multipoint Communication, 1997.
2. O. Ozkasap, R. van Renesse, K. P. Birman, and Z. Xiao, *Efficient buffering in reliable multicast protocols,* International Workshop on Networked Group Communication, Nov. 1999.
3. A-M. Kermarrec, L. Massoulié, and A.J. Ganesh, *Probabilistic Reliable Dissemination in Large-Scale Systems,* IEEE Trans. Parallel and Distributed Systems, 14(3), pp. 248-258, 2003.

Using LL/SC to Simplify Word-based Software Transactional Memory*

Virendra J. Marathe and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{vmarathe, scott}@cs.rochester.edu

February 2005

Abstract

While *compare-and-swap* (CAS) and *load-linked / store-conditional* (LL/SC) are equally powerful in principle, there are circumstances in which one or the other is significantly easier to use. We highlight one example in this paper. Specifically, we use LL/SC to significantly simplify the "stealing" and "merging" mechanism used to ensure correct nonblocking behavior in Harris and Fraser's word-based software transactional memory system (WSTM). Our simplification exploits the observation that LL/SC, as conventionally implemented, provides a natural atomic implementation of a restricted form of *k-compare-single-swap*.

1 Introduction

An atomic k-compare-single-swap (KCSS) [3, 5] atomically verifies k memory locations and updates one of them (call it t). KCSS is useful for nonblocking implementations of concurrent data structures whose atomic updates require consistent *snapshots*. If the locations involved in the snapshot are always updated in an appropriate order, then ideal LL/SC provides a natural implementation of a restricted form of KCSS: t is load-linked, the remaining locations are read and, if the values satisfy some appropriate predicate, t is updated using store-conditional (SC). The restriction is that whenever the k locations satisfy the predicate, the application must refrain from modifying locations other than t until t itself has been updated.

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories.

Real implementations of LL/SC impose additional restrictions: most specify that SC can fail spuriously under certain circumstances (e.g. hardware interrupts), in which case software must retry the atomic sequence. More significantly, some allow SC to fail *deterministically* if the instructions between the LL and the SC attempt to access a location that maps to the same cache set as t.

While deterministic failure limits the generality of the technique, LL/SC-based restricted KCSS can still be highly useful if we can guarantee that t lies in a different cache set from the $k-1$ other locations. In particular, LL/SC-based 2CSS can be used to good effect in the word-based software transaction memory system (WSTM) of Harris and Fraser [1, 2]. We describe the original WSTM in Section 2, drawing attention to a potentially significant scalability problem that can arise when contention is high. In Section 3 we present a modification to WSTM that significantly simplifies the design and eliminates the scalability issue, at the expense of one additional atomic instruction on the low-contention critical path. Empirical evaluation of our modification is in progress.

2 Word-based STM

Software Transactional Memory refers to a family of general purpose constructions that can be used to mechanically transform correct sequential code into nonblocking concurrent code. STM systems generally attempt to maximize concurrency by allowing threads to access disjoint sets of *blocks* concurrently. The WSTM of Harris and Fraser [1, 2] makes each memory word a separate block. The API for WSTM

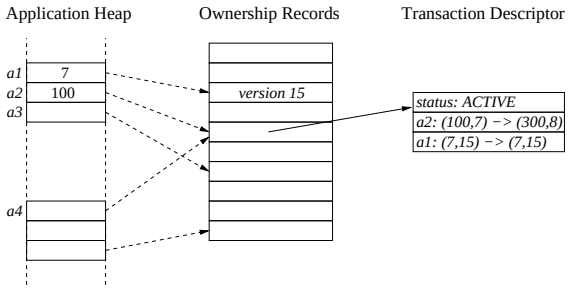

Figure 1: WSTM Heap Structure

has six main entry points: STMStart( ) begins a new transaction. STMRead(addr a) and STMWrite(addr a, stm_word w) are used to read and write shared memory words. STMCommit( ) and STMAbort( ) are used to finalize an ACTIVE transaction. STMValidate( ) verifies that the transaction is still able to commit, which implies that all read locations are mutually consistent.

Figure 2 illustrates the design of WSTM. The *Application Heap* is the shared memory region that holds the actual data. The structure in the middle of the figure is a hash table of *Ownership Records* (orecs). Each orec stores ownership (permission-to-modify) information for all memory words that hash to its index in the table. Each *unacquired* orec stores a version number. A transaction has to *acquire* an orec before modifying any corresponding memory words. Acquiring consists of atomically replacing the orec version number with a pointer to the acquiring transaction's descriptor.

A transaction descriptor contains a list of *transaction entries*, one for each shared word in the application heap accessed by the transaction. Each transaction entry in turn has five fields: the address of the shared word, the original contents of that word, the version number of the corresponding orec, the new contents of the memory word (to be written back at commit time), and the new version number (to be stored in the orec).

A transaction may be ACTIVE, ABORTED, or COMMITTED. An STMRead or STMWrite creates a transaction entry (corresponding to the accessed memory location) if one does not already exist in the transaction descriptor. To maintain consistency, a transaction descriptor must either contain at most one entry corresponding to an orec, or all the entries corresponding to an orec must have the same old and new version numbers. The STMCommit operation attempts to acquire all orecs named in the transaction descriptor. If successful it uses an atomic primitive (the linearization point of the transaction) to switch to COMMITTED state. It then updates the shared heap and *releases* all acquired orecs by swapping in their new version numbers.

An STMRead or STMWrite to a previously unaccessed location inspects the corresponding orec. It the orec points to an ACTIVE transaction, the contender is immediately aborted (this uniform aggressiveness raises the possibility of livelock, making WSTM obstruction-free [4]). The current transaction creates a transaction entry using the appropriate orec version number (old if ABORTED, new if COMMITTED) found in the contender's descriptor.

An STMCommit that discovers a conflict also aborts the contender if it is still ACTIVE. It then *merges* transaction entries (corresponding to the orec under conflict) from the contender's descriptor into its own. Merging allows the current transaction, once it finalizes, to appropriately update any locations for which the contender was responsible, even if the contender is preempted or otherwise inactive. After merging, the current transaction *steals* the orec from its contender by using an atomic primitive to flip the pointer over to its own transaction descriptor.

Use of stealing leads to the problem of stale updates, where a transaction that is a victim of stealing may update words in the heap after the stealer has already done so. A victim realizes this potential problem when it tries to release the stolen orec and its CAS or SC fails. The victim then chases the orec pointer to its stealer's descriptor and *redoes* all updates made by the stealer for the stolen orec. No orec is released until it is guaranteed that the orec is not referenced by any other transaction. This is enforced by the use of a reference count for each orec. An orec is released by a transaction only when the orec reference count goes down to zero. Atomic update to the orec and its reference count requires a double-wide CAS or LL/SC.

Bounded Memory Blow-up

WSTM uses stealing to ensure nonblocking semantics. Stealing entails merging, which in turn leads to potentially long *merge chains* of transaction entries

due to *false sharing*. Let the ratio of the application heap size to the orec hash table size be $M:1$. If hashing is uniform, each orec covers approximately M different shared memory words. In the worst-case scenario, for each memory location that a transaction may access, it may end up possessing $M-1$ extra transaction entries. If a transaction needs to acquire N orecs, it may end up possessing $N \times M$ transaction entries in the process. Memory blow-up may be significant if M is large. Responsibility for extra memory words also increases the worst-case overhead of write back by a factor of M, and introduces the overhead of redos, which may cause severe interconnect contention as cache lines bounce among processors. Although the worst case scenario may rarely occur, its likelihood increases with increasing contention.

3 An Alternative Stealing Approach

WSTM's stealing mechanism leads to the bounded memory blow-up problem and potentially slower transactions. It also introduces significant complexity, and requires a double-wide atomic primitive (not currently available on 64-bit processors) to update version number / pointer pairs. We propose an alternative mechanism that uses *helping* during stealing instead of *merge-redo*. On detecting a conflict, a potential stealer transaction first scans through its victim's descriptor looking for the transaction entries corresponding to the orec under conflict. For each such entry, the stealer updates the corresponding shared memory location using LL/SC to implement a restricted 2-compare-single-swap: the stealer LLs the heap location, verifies that the orec is the same as the one in the transaction entry, and then stores the right value with SC. After the stealer has scanned through its victim's descriptor, it steals the orec under conflict as before. The victim will continue with its release phase normally (without updating memory words corresponding to the stolen orec) even when it sees that some of its orecs have been stolen.

The intuition behind our approach is as follows: If the orec changes after an LL, it is guaranteed that some other stealer has successfully stolen the orec after making correct updates to the heap. The transaction will have to chase the new stealer to resolve the new conflict with its new contender. If the orec is still valid, but the SC fails non-spuriously, it is guaranteed that some other potential stealer has made a correct update to the target memory location. Spurious failures are handled by a retry loop. Stale updates are avoided by verifying the orec contents in between the LL and SC. With our approach the bounded memory blow-up problem is eliminated since no merging of transaction entries happens. The commit operation for a transaction is also simplified considerably. Finally, no reference counts or double-wide atomic operations are required. To avoid deterministic SC failures, we need only ensure that a heap location and its orec never map to the same set in the cache. This is easily achieved, for all reasonable cache line and page sizes, and for both virtually and physically indexed caches, by selecting an appropriate hash function.

Our modification has a downside: a transaction updating N memory words requires $N+2M+1$ LL/SC operations (where M is the number of orecs acquired by the transaction), versus $2M+1$ CASes in the original WSTM. Experimental evaluation of this tradeoff is currently in progress.

References

[1] K. Fraser and T. Harris. Concurrent Programming without Locks. *Submitted for publication*, 2004.

[2] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, October 2003.

[3] T. L. Harris, K. Fraser, and I. A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 265–279. Springer-Verlag, 2002.

[4] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of 23rd International Conference on Distributed Computing Systems*, pages 522–529, May 2003.

[5] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 314–323, June 2003.

Randomization in STM Contention Management *

William N. Scherer III and Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{scherer, scott}@cs.rochester.edu

Abstract

The obstruction-free Dynamic Software Transactional Memory (DSTM) system of Herlihy et al. allows only one writing transaction at a time to access an object. Should a second require an object currently in use, a *contention manager* must determine which may proceed and which must wait or abort.

In this case study, we consider the impact of randomization when applied to our "Karma" contention manager. Previous work has shown that Karma tends to be a good choice of managers for many applications. We analyze randomized Karma variants, using experimental results from a 16-processor SunFire machine and a variety of benchmarks. We conclude that randomizing either abortion decisions or gain can be highly effective in breaking up patterns of livelock, but that randomized backoff yields no inherent positive benefit.

1 Introduction

Although early software transactional memory systems (STMs) were primarily academic curiosities, more modern STMs [1, 2, 3] have reduced runtime overheads sufficiently to outperform coarse-grained locks (with at least moderate contention). Dynamic software transactional memory (DSTM) [3] is a practical STM system novel in its support for dynamically allocated objects and transactions, and for its use of modular contention managers to separate issues of progress and correctness in data structures.

At its heart, contention management in DSTM is the question: how do we mediate transactions' conflicting needs to access a block of memory? In previous work [4], we have shown that the choice of contention management policies dramatically affects overall system throughput and that the Karma manager frequently gives top performance.

In the present work, we explore the impact of randomization in contention manager design. We study Karma as a top contention manager with many facets that can be randomized.

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories.

2 Contention Management

The contention management interface for the DSTM [4] includes notification methods for various events that transpire during the processing of transactions, plus two request methods that ask the manager to make a decision. Notifications include events such as beginning a transaction, successfully/unsuccessfully committing a transaction, attempting to open a block, and successfully opening a block. The request methods ask a contention manager to decide whether a transaction should (re)start and whether enemy transactions should be aborted.

Many researchers have found randomization to be a powerful technique for breaking up repetitive patterns of pathological behaviors that hinder performance. We evaluate this potential by randomizing facets of the Karma manager.

2.1 The basic Karma scheme

The Karma contention manager [4] tracks the cumulative number of blocks opened by a transaction as its *priority*. It increments this priority with each block opened, and resets it to zero when a transaction commits. It does not reset priorities if the transaction was aborted; this gives a boost to a transaction's next attempt to complete. Karma manages conflict by aborting an enemy transaction when the number of times a transaction has attempted to open a block exceeds the difference in priorities between the enemy and itself. Between attempts, it backs off for a

fixed period of time. Intuitively, Karma prefers not to abort a transaction that will take a large amount of effort to redo, but tries to maintain some ability for short transactions to eventually gain enough priority to finish even when competing with longer ones.

2.2 Randomized Backoff

The original Karma scheme backs off for a fixed period of time T between attempts to acquire an object. Randomized, we instead sleep for a uniform random amount of time between 0 and $2T$.

2.3 Randomized Abortion

In response to a `shouldAbort` query, the basic Karma manager returns `true` when the difference Δ between the current and enemy transactions' accumulated priorities is less than the number of times it has attempted to open a block. We randomize this abortion decision with a sigmoid function that returns `true` with probability biased to the higher-priority transaction: $(1 + e^{-\frac{1}{2}\Delta})^{-1}$.

2.4 Randomized Gain

The basic Karma manager gains one point of priority with each object that it successfully opens. Randomized, we instead gain as priority an integer randomly selected from the uniform interval $0..200$.

3 Methodology

All results were obtained on a SunFire 6800, a cache-coherent multiprocessor with 16 1.2Ghz UltraSPARC III processors. We tested in Sun's Java 1.5 HotSpot JVM.

We present experimental results for six benchmarks. IntSet, IntSetUpgrade, and RBTree are implementations of a set of integers; LFUCache simulates web caching [4]. Stack supports push and pop transactions. ArrayCounter transactions either increment each shared counter 0..255 in an array or decrement them in the opposite order; it is a "torture test" that exacerbates any tendency towards livelock.

We implemented all eight combinations of randomizing three facets of the Karma manager. We crossed each variant and benchmark, running for a total of 10 seconds. We display throughput results for eight threads: previous experiments suggest that eight threads is enough for inter-thread contention to affect scalability in the benchmarks, yet few enough that limited scalability of the benchmarks themselves does not skew the results. Figure 1 displays throughput results for the various benchmarks.

4 Analysis

In every benchmark, some combination of randomization improves throughput. In the ArrayCounter, IntSetUpgrade, and IntSet benchmarks, randomizing just abortion decisions yields the best performance. Randomizing both abortion and backoff gives very poor performance in ArrayCounter and RBTree; yet, it improves performance for LFUCache and Stack. Randomizing gain improves performance both alone, and in every combination with other types of randomization, for LFUCache and RBTree.

4.1 Interpretation of results

Randomizing abortion is particularly helpful for the ArrayCounter, IntSet, and IntSetUpgrade benchmarks. Why is this the case? One possible explanation is that randomizing abortion decisions is very powerful for breaking up semi-deterministic livelock patterns. Such livelocking patterns are particularly visible in ArrayCounter: an increment and a decrement that start at roughly the same time are very likely to have similar or identical priorities when they meet; they are thus prone to mutual abortion.

The combination of randomizing backoff and abortion produces great variance in how long a thread waits to abort an enemy transaction. In times when this wait period is shortened, a longer enemy transaction will have less of a chance to complete; transactions in RBTree and ArrayCounter are particularly long. In times when this wait period is lengthened, multiple shorter enemy transactions can complete, competing with one fewer enemy. Indeed, LFUCache and Stack transactions are very short; and with higher thread counts (not shown due to space limitations), this combination is less effective.

There is no obvious analogous deterministic pathology associated with tranaction priority levels. While backoff randomization helps in locking algorithms that have multiple contenders (which can get into simultaneous retry pathology), this problem does not arise in the 2-transaction case. Instead, one continues oblivious to the conflict and the other backs off. This is why randomizing backoff yields comparatively little direct benefit.

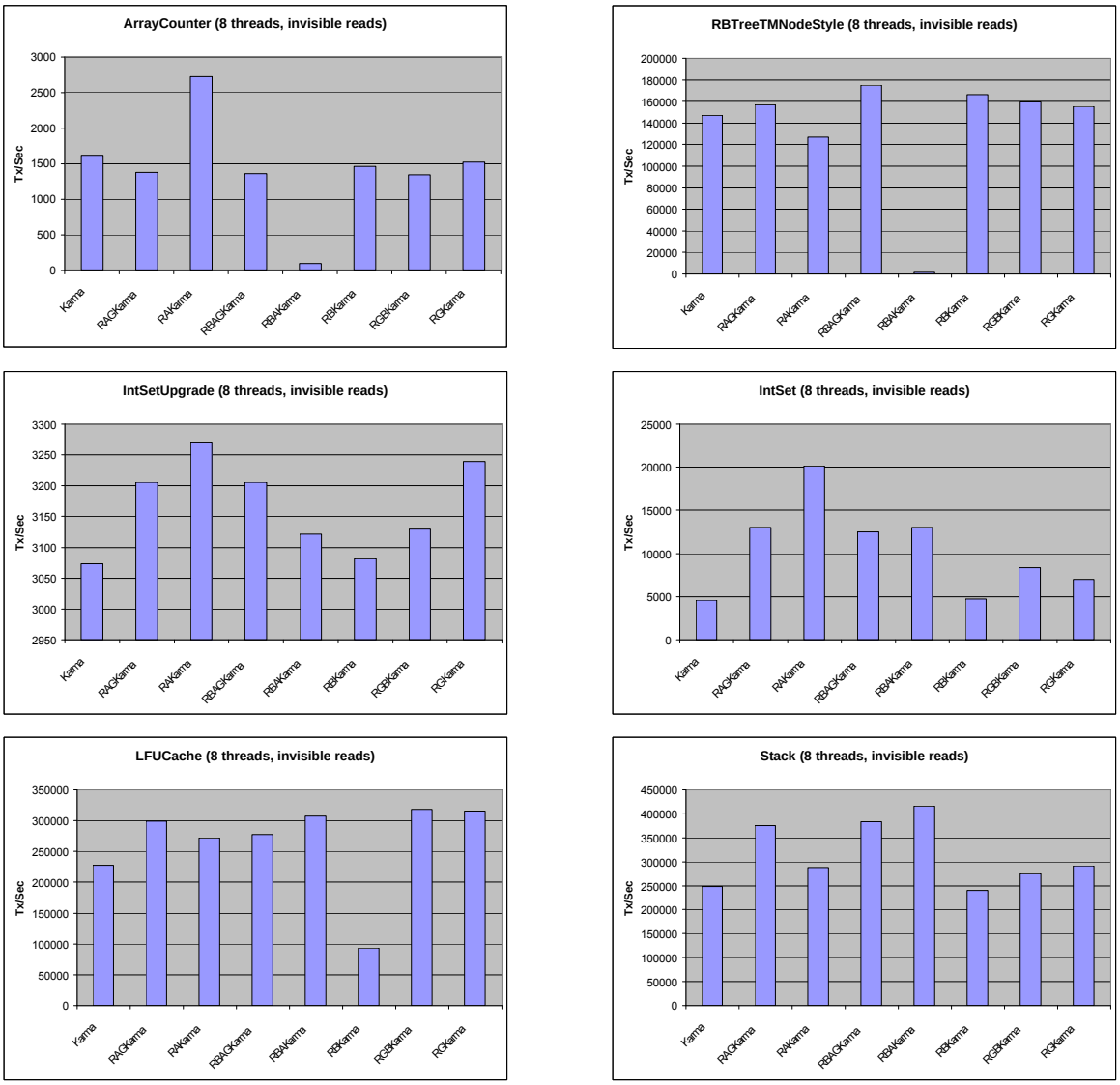

Figure 1: Throughput results for 8 threads and each combination of randomizing backoff (B), abortion (A) decisions, and/or gain (G) upon opening a block (ordered alphabetically)

4.2 Future work

As future work, we plan to analyse other randomized contention managers, more benchmarks, and systems with greater variability in transaction type.

Acknowledgments

We are grateful to Sun's Scalable Synchronization Research Group for donating the SunFire machine.

References

[1] K. Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, Feb. 2004.

[2] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA 2003 Conf. Proc.*, Anaheim, CA, Oct. 2003.

[3] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

[4] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proc. of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July, 2004.