```
get (Vote_Seg, weak-read) with update option
        on time-out begin
            SS log ← abort record;
            return;
        end;
while VOTE ≠ VOTE_REQUEST do nothing;
if participant decides to commit then begin
    SS log ← yes record;
    get (Reply_Seg, read-write)
        REPLY_FROM_NODE[j] := YES;
    discard (Reply_Seg)
    get (Vote_Seg, weak-read) with update option
        while STATUS_OF_TRANSACTION ∉ {COMMIT, ABORT} do nothing;
        on time-out begin
            SS log ← abort record;
            return;
    if STATUS_OF_TRANSACTION = COMMIT then
        SS log ← commit record;
    else
        SS log ← abort record;
end
else begin
    SS log ← abort record;
    get (Reply_Seg, read-write)
        REPLY_FROM_NODE[j] := NO;
    discard (Reply_Seg)
end;
return
```

Appendix B: DSM implementation of the Two Phase Commit Protocol

```
/* The coordinator and participants communicate using
 * DSMC primitives on two shared segments. Vote_ Seg
 * contains two arrays VOTE and STATUS_OF_TRANSACTION.
 * Reply_Seg contains the array REPLY_FROM_NODE
 * for communication between the coordinator and
 * participants for each transaction
 * SS denotes stable storage
 */
```

Co-ordinator's algorithm

```
get (Vote_Seg, read-write)
        VOTE := VOTE_REQUEST;
discard (Vote_Seg)
SS log ← start two phase commit record;
/* On reading VOTE, each participant node N_j writes its decision
 * in REPLY_FROM_NODE[j]
 */
get (Reply_Seg, weak-read) with update option.
∀ j in participants
        while REPLY_FROM_NODE[j] not arrived do nothing;

        on time-out begin
                SS log ← abort record;
                get (Vote_Seg, read-write)
                        STATUS_OF_TRANSACTION := ABORT;
                discard (Vote_Seg)
                return;
        end;
if all REPLY_FROM_NODE[j] were YES and co-ordinator decides to commit then begin
        SS log ← commit record;
        get (Vote_Seg, read-write)
                STATUS_OF_TRANSACTION := COMMIT;
        discard (Vote_Seg)
end;
else begin
        SS log ← abort record;
        get (Vote_Seg, read-write)
                STATUS_OF_TRANSACTION := ABORT;
        discard (Vote_Seg)
        end;
return
```

Participant's algorithm

On receiving discards, write the modified segment on stable storage as a shadow copy

```
Read_complete(segment)
        discard (R);
exit ;

Intent_to_write(segment)
        if self in the majority partition then
                for all nodes in set I do
                        get (R, read-write);
                if any node fails to reply then goto abort;
        endif ;
        exit ;
        abort:
                Return segment R to nodes that replied successfully;
exit ;

Write_complete(segment)
        Update version number for R;            /* VN = VN + 1 */
        Set update nodes cardinality for R;         /*NC = |I| */
        for all nodes in set I do
                discard (R);
exit ;
```

```
        get (R, read-only);
    M = Max{VN_i | 1 ≤ i ≤ m};
    set I = {A_j | VN_j = M, 1 ≤ j ≤ m};
    N = Max{NC_k | A_k ∈ I}.
    for all remote nodes that can communicate with self do
        discard (R);
    if |I| > N/2 then          /* I am in the majority partition. */
        return(true).
    else
        return(false).
exit ;

Updating_copies_of_old_data()
    if self not in the majority partition then abort transaction;
    for all nodes in set I do
        get (R, read-write);
    Update the local copy of segment R.
    Update the version number of local copy of segment R to M;
    Modify the update nodes cardinality of local copy of segment R;
    for all nodes in set I do
        Modify the update nodes cardinality;
        discard (R);
    endfor ;
exit ;

Add_node()
    Set version number of local copy of segment R to −∞;
    Set update nodes cardinality of local copy of segment R to −∞;
    Use the "Updating_copies_of_old_data" procedure.
exit ;

Delete_node()
    if self in the majority partition then
        for all nodes in set I do
            get (R, read-write);
            Increment version number of the segment by 1;
            Decrement update nodes cardinality of the segment by 1;
            discard (R);
        endfor ;
    endif ;
exit ;



Intent_to_read(segment)
    if self in the majority partition then
        get (R, read-only) from some node in set I;
        if time-out then abort transaction;
    endif ;
exit ;
```

```
exit ;

Write_complete(segment)
      Update the version number for this segment;
      if the view number has changed then goto abort;
      for all nodes in set I do
            discard (R);
      exit ;
      abort:
            for all nodes in set I do
                  discard (R) without enclosing segment;
exit ;

Actions_of_View_Object_on_detecting_a_change_in_view()
      get (local View-Seg, read-write);
      Increment the view number value;
      Update the local View-Seg to include/delete nodes;
      discard (local View-Seg);
exit ;

View Update transaction()
      Let set V denote the set of nodes in the new view;
      for all nodes in set V do
            get (View-Seg, read-write);
      for all nodes in set V do
            if remote VID < local VID then
                  Update the remote view by inserting itself in the list;
                  Update the remote VID by making remote counter value equal to its own;
                  discard (View-Seg);
            else
                  discard (View-Seg);
            endif ;
      Eliminate nodes whose VID > local VID from local View-Seg;   /* Only V' ⊆ V accepted the invitation.*/
      for each segment J that has a read quorum in V' do
            Let set I denote the read quorum;
            for all nodes in set I do
                  get (J, read-write);
            Select the most up-to-date copy of J;
            Update all copies of J;
            Update the version number of all copies of J;
            for all nodes in set I do
                  discard (J);
      endfor ;
exit ;
```

Dynamic Voting Algorithm

```
Membership_In_Majority_Partition()
      Let m copies exist in the system;
      for all remote nodes that can communicate with self do
```

```
Read_complete(segment)
        for all nodes in set I do
                discard (R);
exit ;

Intent_to_write(segment)
        Let set I denote a write quorum for segment R;
        for all nodes in set I do
                get (R, read-write);
        if any node fails to return R then goto abort;
        Return the most up-to-date copy of R;
        exit ;
        abort:
                Issue discard (R) to all nodes that returned R;
exit ;

Write_complete(segment)
        for all nodes in set I do
                Increment version number by 1;
                discard (R);
        endfor ;
exit ;
```

Virtual Partition Algorithm

```
Intent_to_read(segment)
        Check if some read quorum exists for segment R within its view;
        if not (read quorum) then abort transaction;
        Let set I denote such a read quorum;
        get ( R, read-only) from some node in set I;
        if time-out occurs then abort transaction;
exit ;

Read_complete(segment)
        discard (R);
exit ;

Intent_to_write(segment)
        Check if some write quorum exists for segment R within its view;
        if not (write quorum) then abort transaction;
        Let set I denote such a write quorum;
        Note the current view number;
        for all nodes in set I do
                get (R, read-write);
        if any node fails to return R then goto abort;
        exit ;
        abort:
                for all nodes in set I that returned R do
                        discard (R) without enclosing segment;
```

Appendix A: DSM implementation of Replication Algorithms

Available Copies Algorithm

```
Intent_to_read(segment)
      loop:
            Pick some node j that has a copy of segment R;
            get (R, read-only) from node j;
            if time-out then goto loop or exit ;          /* If node failure occurs retry or exit */
exit ;

Read_complete(segment)
      discard (R);
exit ;

Intent_to_write(segment)
      Let K denote the set of nodes that contain a copy of segment R;
      for all nodes in set K do
            get (R, read-write);
      Let set I denote the set of nodes that did not return R;
      Let set J denote the set of nodes that returned R;
exit ;

Write_complete(segment)
      /* Set J denotes the set of nodes that are alive and set I denotes the set of nodes that are dead. */
      for all nodes in set J do
            discard (R) enclosing the modified segment;
      if |I| ≠ 0 then          /* Perform missing writes validation if |I| ≠ 0 */
            for all nodes in set I do
                  get (R, weak-read);
            if received R from any node in I then abort transaction;
      endif ;
      for all nodes in set J do          /* Access validation.*/
            get (R, weak-read);
      if did not receive R from any node in J then abort transaction;
exit ;
```

Quorum Consensus Algorithm

```
Intent_to_read(segment)
      Let set I denote a read quorum for segment R;
      for all nodes in set I do
            get (R, read-only);
      if any node fails to return R then goto abort;
      Return the most up-to-date copy of R;
      exit ;
      abort:
            Issue discard (R) to all nodes that returned R;
exit ;
```

[RS79] D. Ries and M. Stonebraker. Locking granularity revisited. *ACM Transactions on Database Systems*, 4(2):210 – 227, June 1979.

[WF90] Kung-Lung Wu and W. Kent Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4 – 24, March 1990.

[BG84] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. In *ACM Trans. on Database Systems*, volume 9, pages 596–615, December 1984.

[BGH87] P. A. Bernstein, N. Goodman, and V. Hadzilacos. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Series in Computer Science. Addison Wesley Publishing Co., Reading, Massachusetts, 1987.

[BL82] H. Breitwieser and M. Leszak. A distributed transaction processing protocol based on majority consensus. In *Proceedings of the 1st ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 224–237, Ottawa, August 1982.

[DGS+90] David DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44 – 61, March 1990.

[ES83] D.L. Eager and K.C. Sevcik. Achieving robustness in distributed database systems. In *ACM Trans. on Database Systems*, volume 8, pages 354–381, September 1983.

[Gra78] Jim Gray. Notes on database operating systems. Technical report, RJ 2188, IBM Res. Lab, San Jose, CA, February 1978.

[JM87] S. Jajodia and D Mutchler. Dynamic voting. In *ACM SIGMOD International Conference on Data Management*, pages 227–238, 1987.

[KS87] A. Kumar and M. Stonebraker. Performance evaluation of an operating system transaction manager. In *Proceedings of the 13th Very Large Databases Conference*, pages 473 – 481, 1987.

[Nel81] B. J. Nelson. *Remote Procedure Call.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1981. Technical Report CMU-CS-81-119.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and Yousef Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 160–169, August 1989.

[RK89] Umakishore Ramachandran and Yousef Khalidi. An implementation of distributed shared memory. In *Proc. of the USENIX Workshop on Experiences in Building Distributed and Multiprocessor Systems*, pages 21–38, October 1989.

[RKW+89] U. Ramachandran, M. Y. A. Khalidi, C.J. Wilkenloh, S. Menon, R.J. Leblanc, P. W. Hutto, P. Dasgupta, R. C. Chen, J. M. Bernabeu, W. F. Appelbe, and M. Ahamad. The Clouds Experiance: Building an object-based distributed operating system. In *Proc. of the USENIX Workshop on Experiences in Building Distributed and Multiprocessor Systems*, October 1989.

Scheme	Request-type	Using Message-passing	Using DSMC
Available Copies	Read (nacks)	$2k+3$	
	(time-outs)	$k+4$	$k+2$
	Write	$7k-5$	$5k-3$
Quorum Consensus	Read	$5r$	$3r$
	Write	$5w$	$3w$
Virtual Partition	Read	5	3
	Write	$5w$	$3w$
	Update	$3(n-1)+5rj$	$3(n-1)+3rj$
Dynamic Voting	Read	5	3
	Write	$5k$	$3k$
	Majority	$2(k-1)$	$3(k-1)$
	Update	$2k+3$	$3(k-1)$
	Addition node	$2k+3$	$3(k-1)$
	Deletion node	$2(k-1)$	$3(k-1)$

Table 1: Summary of message count

Legend.

k = Number of copies of segment **R**. n = Number of nodes in the system.

r = Read Quorum for segment **R**. w = Write Quorum for segment **R**.

tation.

References

[ACL87] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609 – 654, dec 1987.

[ADC85] A. El Abbadi, D.Skeen, and F. Christian. An efficient fault-tolerant protocol for replicated data management. In *Proc. of the 4th ACM SIGACT-SIGMOD Symp. on Principles of Database System*, pages 215–228, Portland, Oregon, March 1985.

[AT86] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proc. of the 5th ACM SIGACT-SIGMOD Symp. on Principles of Database System*, pages 240–251, Cambridge, MA, March 1986.

[BAC+90] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. Prototyping Bubba, a highly

5.4 Dynamic Voting Algorithm

Let us assume that k copies exist in the system. Using message-passing, a read costs 5 messages as data is read from only one node. However, a write costs $5k$ messages as data is written to all the copies. The cost of determining membership to the majority partition costs $2(k-1)$ messages as the information about version numbers and update nodes cardinality is required. The cost of updating old copies of data is the sum of the cost of getting the latest copy (5 messages) and the cost of updating the version number and update nodes cardinality at all nodes ($2(k-1)$ messages), yielding a total cost of $2k+3$ messages. The cost for adding a node to the set of nodes having a copy of a particular item is same as the cost of updating copies of old data, i.e., $2k+3$ messages. The cost of deleting a node that has a copy involves sending new version number and update nodes cardinality to the other $k-1$ nodes, yielding a total cost of $2(k-1)$ messages.

Using similar arguments, in the implementation using distributed shared memory a read costs 3 messages and a write costs $3k$ messages. The cost of determining membership to a majority partition, updating copies of old data, adding new nodes, and deleting nodes are all the same, and are equal to $3(k-1)$ messages.

The cost of performing reads and writes are less using distributed shared memory as compared to message-passing. However the cost of computing the majority function, updating old copies, adding nodes and deleting nodes are cheaper using message-passing since these operations do not require as much data access synchronization.

6 Conclusion

We believe that the abstraction of distributed shared memory offers an elegant alternative to RPC for structuring distributed systems. We showed how the primitives provided by the DSMC simplify the implementation of atomic transactions, which require the one-copy serializability property. The implementations of four well-known replication algorithms on top of DSMC were given and compared against message-based implementations of the same. The fact that the DSMC primitives combine synchronization and data transfer in coherence maintenance results in reducing the overhead of replication management compared to a message-based implemen-

Reads using distributed shared memory cost 2 message less than using message-passing with time-outs and $k+1$ messages less with negative acknowledgements. On the other hand writes costs $2k-2$ messages less than message-passing, a significant reduction in overhead as the value of k, the number of copies, increases.

5.2 Quorum Consensus Algorithm

Let r and w be the number of nodes in the read quorum and write quorum of a segment, respectively. Using message-passing a read costs $5r$ messages as the data is read from the read quorum set. Similarly, a write costs $5w$ messages.

Using distributed shared memory, each read (write) amounts to performing a get and a discard operation from the node that is in the read (write) quorum. Thus a read costs $3r$ messages while a write costs $3w$ messages. In either case our scheme turns out to be cheaper than using message-passing. The difference is proportional to the number of nodes in the read or write quorum set.

5.3 Virtual Partition Algorithm

Let us assume r to be the read quorum, w to be the write quorum, j to be the number of segments that need to be updated as a result of change in view for a node, and n to be the number of nodes in the system.

Using message-passing, a read costs only 5 messages as data is read from only one node. However a write costs $5w$ messages as data is written to the write quorum set. Assuming that $n-1$ nodes participate in forming the new view then the cost of updating one's view is the sum of cost of sending the join view message to $n-1$ nodes ($2(n-1)$ messages) and the cost of sending $n-1$ view formed messages ($n-1$ messages). Additional cost is incurred due to updating copies of old data that now have some read quorum in the new view. Adding this cost of $5rj$ messages, the total cost of view update is $3(n-1)+5rj$ messages.

Using similar arguments as above, the distributed shared memory implementation of read costs 3 messages, write costs $3w$ messages, and view update costs $3(n-1)+3rj$ messages, all of which are cheaper than the corresponding message-passing implementation.

A successful read/write from any node using message-passing costs 5 messages while using DSM costs 3 messages. These costs may not be apparent from the replication algorithms but are present in the implementation. Table 1 summarizes the total costs associated with each operation for the message-passing and distributed shared memory implementations. Note that the costs are for the worst case scenario for the operation in a particular implementation.

5.1 Available Copies Algorithm

Let us assume that k copies exist in the system, $k-1$ nodes fail to process a read request, and the node having the k^{th} copy processes the request.

For the message-passing implementation the associated cost is computed as follows. For an implementation using negative acknowledgements, a read would require the node to send a read-lock request to k nodes. If $k-1$ nodes respond with a negative acknowledgement and the k^{th} node responds with a confirmation then this phase costs $2k$ messages. The cost of obtaining the data from the k^{th} node is 2 messages. The cost of releasing the lock is 1 message. Thus, the total cost of read is $2k+3$ messages. Similarly the total cost for read using time-outs would be the sum of costs of sending a read-lock request to k nodes (k messages), receiving confirmation (1 message), obtaining data from the k^{th} node (2 messages), and releasing the lock (1 message), yielding a total cost of $k+4$ messages.

For writing a value, the node would need to issue k write-lock requests (k messages). Let us assume that $k-1$ nodes grant the request ($k-1$ messages). Data transfer would cost another $2(k-1)$ messages. The cost of the releasing $k-1$ write-locks is $k-1$ messages. The cost of performing missing writes validation is 1 message and that of access validation is $2(k-1)$ messages, giving a total cost of $7k-5$ messages.

Assuming the read is processed by the k^{th} node, an implementation of read using distributed shared memory would include $k-1$ messages for $k-1$ unsuccessful reads and 3 messages for the k^{th} successful read. Therefore, the total cost of read is $k+2$ messages. A write would require the issue of k get requests. Let $k-1$ requests be successful. Including the discards, the successful writes cost $3(k-1)$ messages, and the unsuccessful request costs 1 message. The cost of performing missing writes validation for one node is 1 message and that of access validation is $2(k-1)$ messages. Thus, the total cost of a write is $5k-3$ messages.

Read operation

If TM_A is in the majority partition then it requests segment R in mode read-only from some node in set I. If the request to any node times-out then the transaction is aborted. After processing the read, the segment is returned.

Write operation

If TM_A is in the majority partition then it requests segment R in mode read-write from all the nodes in set I. If the request times-out before a reply is received from any one node then the transaction is aborted. After the write is processed from the current copy of R, the version number is incremented by 1 and the update nodes cardinality is set to $|I|$. The modified segment R is returned to all nodes in set I.

5 Evaluation

In this section we quantify the cost of implementing the four replication algorithms on DSMC and contrast it with a message-passing implementation. In this analysis, the following costs (in number of messages) are assigned to the DSMC primitives.

- *get* primitive – A get primitive that is successful requires two messages, one for the initial request and the second for data transfer. If a get primitive fails then the cost is one message for the initial request.
- *discard* primitive – For a discard primitive, the cost is one message for the return of the segment.

For a message-passing implementation of the four replication algorithms, the following costs are assigned for synchronization and data transfer.

- Lock – Obtaining a lock requires two messages, one to transmit the request and one for the acknowledgement.
- Data transfer – After the lock is obtained, data transfer requires two messages: one message to transmit a request for data and one message for transfer of data.
- Unlock – Releasing a lock requires one message to transmit the request.

update is done to f_i, then NC_i is set equal to the number of nodes that participated in this update.

5. Let $\mathbf{N} = \text{Max}\{NC_k | A_k \in \mathbf{I}\}$. A node is said to be in a *majority partition* if $|I| > \text{N}/2$, i.e., this partition contains the majority of the current copies of object *f*.

In Dynamic Voting [JM87], reads can be done by a TM_A if a simple majority of nodes that can communicate with it have the current copy; similarly writes can be done if a simple majority of nodes that can communicate with it have the current copy. A write is done to all the nodes that have the current copy. Number of successful nodes participating in the write is recorded as the update nodes cardinality of the segment.

In the implementation sketched below, the transaction manager assumes that each segment contains its version number and update nodes cardinality.

Determining membership in the majority partition

TM_A issues a request for segment R in mode read-only to the nodes that have a copy. The version number and update nodes cardinality of the segments is used to compute the current version number (M), set I, and N. If $|I| > \text{N}/2$ then TM_A is in the majority partition. All the segments are returned at the end of the transaction.

Updating copies of old data

If TM_A is in the majority partition then it requests segment R in mode read-write from all the nodes that have a copy of R. TM_A updates the local copy of R from the most up-to-date copy of R, increments its version number, and sets its update nodes cardinality to $N + 1$. The update nodes cardinality of all other copies of R is set to $N + 1$. All the segments are returned at the end of the transaction.

Addition of new nodes

TM_A sets the version number and update node cardinality of its copy of R to $-\infty$. It then follows the procedure "Updating copies of old data".

Deletion of nodes

If TM_A is in the majority partition then it requests the copy of segment R in mode read-write from all the nodes. TM_A increments segment R's version number by 1 and decrements the update nodes cardinality by 1. The segments are returned to the respective owner nodes.

get request in mode read-write for segment R to all nodes in the write quorum. If the request times-out for any node then the transaction is aborted. Once TM_A has received the copies of segment R, it notes the present view number. On issue of write_complete, TM_A determines whether its view has changed since the write was initiated. If the view did change then the write is aborted. All the segments are discarded if the write is successful.

Virtual Partition's View Update Transaction

The View Object, on detecting a change of view, requests the local view segment in mode read-write and increments the view number. Whenever a transaction manager detects a change in its view it needs to perform a View Update transaction to update its view and to update the copies of data that now have a read quorum in the new view.

TM_A requests the view segment in mode read-write from all other nodes that are in the new view. New nodes are added to the view if their *view identifier* (VID) are less than its VID. If a new node (say node B) is added to the view then TM_A updates its local view and also writes the new VID in node B's view segment. All view segments are discarded at the end.

Once the new view has been established then TM_A identifies all the segments that have some read quorum in the new view. For these segments, TM_A requests all copies of R in mode read-write from all the nodes that form the respective read quorums and updates the copies of R using the latest copy. The segments are then returned to their respective owner nodes.

4.5 Dynamic voting

This algorithm uses the following terms:

1. There are m copies of an object f. VN_i is the version number associated with the i^{th} copy.

2. The current version number (M) for an object is equal to the maximum version number for all its copies, i.e., $\mathbf{M} = \text{Max}\{VN_i | 1 \leq i \leq m\}$.

3. A copy of an object f is said to be current if its version number is equal to the current version number (M) for that object. Let set $\mathbf{I}$ denote the set of nodes that have a current copy of an object f. Set $\mathbf{I} = \{A_j | VN_j = M, 1 \leq j \leq m\}$ where A_j represents node j.

4. Update nodes cardinality (NC_i) reflects the number of nodes participating in the most recent update to f_i. Initially NC_i is set equal to n, the number of nodes. Whenever an

Write operation

TM_A issues a get request in mode read-write for segment R to all nodes in the write quorum set. If any node fails to return R then TM_A aborts the write request. Before discarding, the version number of each copy of R in incremented. The segments are discarded on the issue of write_complete.

4.4 The Virtual Partition Algorithm

The Virtual Partition algorithm [ADC85, AT86] is designed such that a transaction never has to access more than one copy to read a data item. Each node (A) maintains a view, consisting of nodes with which it believes it can communicate. This set is denoted by *v(A)*. The view for a transaction T executing on node A must contain the read (or write) quorum for x for a Read(x) (or Write(x)) to be successful. Within the view in which T executes, the database system uses the write-all approach. It translates Write(x) into writes on all copies of x in *v(A)*; it translates Read(x) into a read of any copy of x in *v(A)*. Whenever a node's view changes, all active transactions at that node must abort. As a result of change in view for the node, some data items may now have a read quorum within the new view. These data items are updated using a View Update algorithm.

The implementation of the Virtual Partition algorithm using DSMC assumes that the version number associated with a segment is contained in the segment itself, and that the transaction manager at each node has a view segment that contains the list of nodes that can communicate with that host.

Read operation

TM_A checks whether a read quorum exists for segment R within its view. If a read quorum does not exist then TM_A aborts the transaction. If a read quorum exists then TM_A issues a get request in mode read-only for segment R to any node in the read quorum. If the request times-out before a reply is received then the transaction is aborted. The segment is discarded on the issue of read_complete.

Write operation

TM_A checks whether a write quorum exists for segment R in its present view. If a write quorum does not exist then TM_A aborts the transaction. If a write quorum exists then TM_A issues a

algorithm uses a validation protocol (missing writes validation and access validation) to ensure correctness.

Read operation

The transaction manager at node A (TM_A) selects one node, say node B, that has a copy of segment R. TM_A issues get request for segment R to node B in mode read-only. If TM_A times-out before a reply comes from node B then TM_A assumes node B to have failed. TM_A can either terminate the read or select another node that has a copy of R to read from. Once segment R is obtained, TM_A can process the read request. The requested segment is discarded on the issue of read_complete.

Write operation

TM_A issues a get request for segment R in mode read-write to all nodes that have a copy of segment R. Nodes that return segment R are assigned to set J while nodes that fail to return R are assigned to set I. The segment is returned to the requesting transaction. On write_complete, TM_A discards segment R to all nodes in set J. After doing discards, TM_A needs to perform the missing writes validation and access validation to ensure correctness. This phase is done by requesting segment R from all nodes in mode weak-read. The successful replies are compared with set J and unsuccessful replies are compared with set I. If any discrepancy is found then the transaction is aborted.

4.3 Quorum Consensus Algorithm

In Quorum Consensus algorithm [BL82], the transaction manager translates each read into a set of Read(x) on each copy of some read quorum of x, and it returns the most up-to-date copy of R that it read. Similarly, each write is translated into a set of writes on each copy of some write quorum of x.

In the implementation that is sketched, the algorithm assumes that each segment contains its version number.

Read operation

TM_A issues a get request in mode read-only for segment R to all nodes in the read quorum set. If any node fails to return R then TM_A aborts the read request. The segments are discarded on the issue of read_complete.

4 Fault-Tolerance

Replication has been proposed as a way of achieving fault-tolerance [BGH87]. Replicated data introduces additional complexity in data management as the transaction manager must now maintain data consistency between multiple copies of the same data.

Many replication algorithms have been proposed to maintain data consistency. We have implemented four replication schemes on top of DSMC. The focus of the rest of the paper is in evaluating the implementation of these replication schemes with respect to DSMC. Appendix A gives the pseudo code for each of the four schemes.

4.1 Assumptions

To implement the replication schemes the following assumptions have been made:

1. The transaction manager has knowledge of all nodes where a given segment is replicated.

2. The transaction manager has knowledge of the read and write "threshold" associated with each segment.

3. The lower level communication layer returns two kinds of time-out error codes to the DSMC. The first kind of time-out occurs when the remote node is down and the communication layer is unable to get the request across. The second kind of time-out error is returned to the DSMC when a node goes down and stops transmitting any "I am alive" messages. As a result of these time-out exceptions, it is the responsibility of the DSMC to reinitiate all transactions that are pending with the failing node.

4. Each transaction manager keeps track of the segments requested by individual transactions. This information is used to perform discards on the issue of read_completion and write_completion.

4.2 Available Copies Algorithm

In Available Copies algorithm [BG84, ES83], the transaction manager translates every Read(x) into a read of any copy of x and every Write(x) into writes on all available copies of x. The

3.2.1 Recovery Manager

The recovery manager provides recovery of processes from node failures and performs the two-phase commit protocol on behalf of a transaction. Recovery from process failures is achieved through checkpointing. Checkpoints are taken whenever a process performs a discard of a modified segment. Checkpoints are maintained on stable storage. Our scheme for checkpointing is similar to [WF90].

Atomicity of a transaction is guaranteed through the use of the two-phase commit protocol. When a transaction commits, the transaction manager performs discards of the segments held by the transaction and signals the recovery manager to commit the transaction. The DSMC treats discards that are part of a transaction differently from the normal discards. While a normal discard immediately releases the lock on the segment and updates the segment (if modified), a discard that is a part of a transaction is passed to the recovery manager by the DSMC. The recovery managers at the owner nodes maintain shadow copies of the discarded segments (if modified) on stable storage. Upon receiving end-transaction from the transaction manager, the recovery manager at the transaction's home node initiates a two-phase commit protocol (Appendix B). If the transaction commits successfully then the recovery managers that participated in the two-phase commit overwrite the original segments with the modified segments (shadow copies on stable storage) and release the locks on all the segments that were touched by this transaction.

Upon receiving abort-transaction from the transaction manager, the recovery managers immediately release the locks on all the segments that were touched by this transaction and disregard the modifications to the segments that may be on stable storage.

3.2.2 Issues of granularity

DSM implements locks at the page level. This level of locking may be coarse for certain applications such as transaction processing systems (Bank records) [BAC+90]. On the other hand, there are several other distributed applications that advocate and use page-level locking [DGS+90, KS87, RS79, ACL87]. For such applications the mechanisms provided by DSMC would prove beneficial for implementing two-phase locking.

3.2 Transactions Using Distributed Shared Memory

With a DSMC at each node the software architecture for implementing atomic transactions is organized as follows: Memory accesses subsequent to the issue of begin-transaction from a user process are treated as part of the current transaction by the transaction manager. Upon a page-fault the transaction manager acquires the requested segment in the appropriate mode from the owner node using the DSMC primitives. The mode is determined by the nature of the page-fault (read or write). On a write-fault to a segment that has been previously acquired in a read-only mode, the transaction manager issues an upgrade request to the owner. The transaction manager translates the read and write requests into *intent_to_read* and *intent_to_write* calls. The transaction manager keeps track of all the segments that have been requested by a particular transaction. When the transaction commits, the transaction manager performs *read_complete* and *write_complete* for the appropriate segments on behalf of the transaction. In essence, the transaction manager implements a two-phase locking protocol for concurrency control, thus resulting in one-copy serializability [BGH87]. The recovery manager (section 3.2.1) ensures that the distributed shared memory contains all the effects of the committed transactions and none of the effects of the aborted ones.

It is easy to see that intent_to_read and intent_to_write directly translate to DSMC get request with modes read-only and read-write, respectively. The read-complete and write-complete operations correspond to DSMC discard. The new software organization consists of just three modules: the transaction manager, the recovery manager, and the DSMC. The tasks of the scheduler and the cache manager are now subsumed by the DSMC. As a result, the architecture for implementing atomic transactions is simplified. It is assumed that the distributed data is organized as DSMC segments, with appropriate attributes such as read-only and read-write. The transaction manager is the interface between the user transactions and the DSMC. It issues gets and discards corresponding to the operations executed by the transactions. Since the DSMC primitives combine locking and data transfer, transaction-level locking of segments comes for free. The key point to note here is that the DSM abstraction provides the mechanisms needed for implementing the two-phase locking protocol thus relieving the work that needs to be done at a higher level (by the scheduler and the transaction manager) in a more traditional implementation.

transaction and *end-transaction* respectively.

The transaction manager performs any required preprocessing of the read and write operations on shared data that it receives from a transaction. The scheduler controls the relative order in which the transaction operations are executed. It is the task of the scheduler to provide one-copy serializable execution of transactions by either delaying, executing, or rejecting each operation from the transaction manager. The scheduler uses information local to itself to decide when to execute each operation. The recovery manager is responsible for transaction commitment and abortion. Finally, the cache manager performs the interaction between the physical memory and the stable storage. Recently, replication of data has been advocated to achieve fault-tolerance in distributed systems. In a distributed system with replicated data, the transaction manager is also responsible for maintaining multiple copies of data and determining the node that should process each operation submitted by a transaction.

3.1 Implementation of Atomic Transactions

Given the two paradigms for communication and synchronization in a distributed system (section 1), atomic transactions can be implemented in one of two ways. In a traditional implementation, the transaction manager would send messages to the scheduler at the remote node for any operation that it needs to perform on the data at that node. The scheduler would then decide whether to execute, delay, or reject the request. The scheduler needs to implement some kind of synchronization to control access to shared data. The response (control message, data) is sent back to the requesting transaction in a set of messages.

From the point of view of one-copy serializability each transaction requires that its execution appear as if it happened without any interference from other concurrent transactions. In a distributed system, this requirement amounts to providing a view of the distributed data as one large shared memory with individual transactions holding appropriate locks (read or write) over parts of the memory that they need to access atomically. The transaction manager acquires these locks for the transactions using a two-phase locking protocol [Gra78]. The scheduler at each node is responsible for ensuring that these lock requests do not violate the single-copy serializability requirement of atomic transactions.

Upon receiving a get request in mode weak-read, the owner DSMC sends a copy of the segment to the requesting DSMC. Note that this may not be an up-to-date copy if the segment is currently write-locked by some other node. If the option to receive updates is set, then the owner automatically sends updates of the segment when the write-lock is released. On receiving a get in mode none, the owner DSMC does one of the following: the none mode request is queued if the segment is currently locked in either read-only or read-write mode; if the segment is available, it is sent to the requesting DSMC who now becomes the *keeper* of the segment; a subsequent request for the same segment (in modes read-only, read-write, or none) is forwarded to the *current keeper* who forwards the segment to the requesting DSMC. A segment held in mode none can be returned to the owner by using a discard primitive, or it can be taken away by its owner when the keeper DSMC is instructed to forward the segment to another node. The upgrade primitive is provided for changing the permission on a currently held segment from read-only to read-write. The algorithms for coherence maintenance are in [RAK89].

The DSMC primitives have been implemented on Unix and Clouds[1] [RK89]. On Unix it has been implemented as a user process that communicates with other DSMCs using a client-server type transport protocol. Implicit in the implementation is the assumption that each DSMC can locally determine the owner of each segment in the system. The DSMCs consult a location object that provides them with this information. For details of the implementation the reader is referred to [RK89].

3 Atomic Transactions

Atomic transaction are an important and useful abstraction in distributed computing. Usually, operating systems do not provide explicit support for this abstraction thus necessitating some higher level software entity to implement the mechanisms needed for this abstraction. The software architecture for implementing atomic transactions typically consists of four modules at each node [BGH87]: a *transaction manager*, a *scheduler*, a *recovery manager*, and a *cache manager*. The computational model is as follows: User processes executing at each node signal the start and end of a new atomic transaction to the transaction manager through *begin-*

[1]Clouds is a distributed operating system being developed at Georgia Institute of Technology [RKW+89]

1. *get(segment, mode).*

2. *discard(segment).*

3. *upgrade(segment).*

4. *P(segment, semaphore).*

5. *V(segment, semaphore).*

The get primitive is used for fetching a segment from the owner while discard is used to return a segment to its owner. To provide distributed synchronization the P and V primitives have been provided. The DSMC can discern the owner of a segment by consulting a *segment locator*.

The get primitive can be used to acquire a segment in one of the following four modes: read-write, read-only, none, and weak-read. *Read-write* mode signifies exclusive access to a segment guaranteeing that the segment will not be thrown away until the node explicitly discards the segment. *Read-only* mode indicates non-exclusive access with the guarantee that the segment will not change until the node explicitly discards the segment. *None* mode (the default) indicates exclusive access with no guarantee whether the segment will be thrown away or not. *Weak-read* mode signifies non-exclusive access with no guarantee whether the segment will change or not. A node can obtain a segment in mode weak-read with an option to receive updates of the segment implicitly.

The DSMCs implement a First-Come-First-Served queue discipline for processing remote segment requests. If the request cannot be honored immediately, it is queued on the segment until the relevant lock is released.

When a get primitive is issued in mode read-write or read-only the local DSMC sends a request to the remote DSMC requesting the segment. The remote DSMC locks the segment (if it is currently unlocked) with a read-lock (for read-only) or a write-lock (for read-write) and returns the segment to the requesting DSMC. The owner node keeps a count of the number of readers associated with a segment. Thus concurrent access of a segment is allowed for reading, and exclusive access for writing. The segment is kept write-locked until an explicit discard is received, and read-locked until all the readers have discarded the segment.

1 Introduction

There are two paradigms for interprocess communication and synchronization: message-passing and shared memory. In a distributed system, the former takes the more structured form of remote procedure call (RPC) [Nel81]. Traditionally, in a distributed system, synchronization between concurrently executing atomic transactions has been implemented using the message-passing paradigm.

Distributed Shared Memory (DSM) is an alternative to RPC for structuring distributed systems. This paradigm gives the abstraction of a shared memory in a physically non-shared memory (distributed) architecture. Given that several software architectures assume the existence of such an abstraction, it is worthwhile to provide mechanisms to support such an abstraction.

Section 2 presents the details of our DSM abstraction. Given this abstraction we show how elegant and simple it is to implement atomic transactions on top of these mechanisms (Section 3). To achieve fault-tolerance, we study the implementation cost of several replication schemes on top of distributed shared memory (Section 4). We show that DSM-based implementation of fault-tolerant atomic transactions results in much less overhead than RPC-based ones.

2 Distributed Shared Memory

The hardware environment consists of a collection of computing nodes interconnected by a local area network. Each node has one or more processors and memory. Nodes do not share memory physically. Stable storage is available at each node to back the physical memory. The DSM abstraction implements a single level store (collapsing the stable storage and physical memory) allowing the virtual memory space in the entire distributed system to be viewed as one globally shared memory. The memory space is composed of segments, where a segment is a logical entity that has attributes such as read-only and read-write. There is a concept of ownership and the node where the segment is created (the owner node) is responsible for guaranteeing the consistency of the segment. There is an entity, *distributed shared memory controller* (DSMC), at each node that provides the mechanisms for managing these segments.

DSMC provides the following five primitives:

Implementing Fault-tolerant Atomic Transactions Using Distributed Shared Memory*

Ajay Mohindra *Umakishore Ramachandran*

GIT-CC-91/13
January, 1991
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

The abstraction for supporting the notion of shared memory in a non-shared memory (distributed) architecture is referred to as distributed shared memory. We have implemented a set of mechanisms for maintaining the coherence of distributed shared memory. In this paper we show how distributed fault-tolerant atomic transactions can be implemented elegantly using these mechanisms. Since our mechanisms combine the tasks of synchronization and data transfer into one, transaction-level locking of segments comes for free. Fault-tolerance is achieved by using replication on top of distributed shared memory. Atomicity of transactions is guaranteed through use of a two-phase commit protocol. We show that the resulting implementation incurs much less overhead compared to message-passing schemes.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.