Atomic Data Access in Content Addressable Networks
A Position Paper

Nancy Lynch, Dahlia Malkhi, David Ratajczak
(lynch@theory.lcs.mit.edu, dalia@cs.huji.ac.il, dratajcz@yahoo.com)

Abstract

While recent proposals for *content addressable networks* address the crucial issues of communication efficiency and load balancing in dynamic networks, they do not guarantee strong semantics of concurrent data accesses. Given that many of these proposals involve aggressive data replication without a strategy for concurrency control, they admit the possibility of returning stale data, or falsely reporting certain data as unavailable, if certain sequences of joins, leaves, and updates/lookups occur. While it is well known that guaranteeing availability and consistency in an asynchronous and failure prone network is impossible, we believe that guaranteeing atomic semantics is crucial for establishing CANs as a robust middleware service, and this property should not depend on the timing of the underlying system.

In this paper, we describe a simple CAN algorithm that maintains the atomicity property regardless of timing, failures, or concurrency in the system. The liveness of the algorithm, while not dependent on the order of operations in the system, does require that node failures are masked (through state machine replication or through recovery from persistent storage) and that the network eventually delivers all messages to intended recipients.

1 Introduction

Several groups have proposed *content addressable networks* [RFH+01] as a building block for large-scale distributed systems, sometimes under the alias of *distributed data structures* [GBH+00, LNS96], *resource lookup services* [SMK+01], or *peer-to-peer routing services* [ZKJ01]. CANs are composed of *nodes* that are allowed to join and leave the system and that share the burden of implementing a distributed hash table of *data objects*. For large networks, only limited portions of the data set and/or membership set might be known to any particular node; thus it is possible that accesses to the data structure are forwarded between nodes until an appropriate handler of that data object is found. CAN proposals are generally distinguished by the way in which the data set is partitioned and sparse routing information is maintained.

The design of efficient CANs is confounded by opposing design goals. First, the set of nodes is assumed to be large, dynamic, and vulnerable to failure, so it is imperative not only to manage joins and leaves to the network efficiently while maintaining short lookup path lengths and eliminating bottlenecks, but also to replicate data and routing information to increase availability. Most CAN proposals focus primarily on these objectives. However, another design goal, and one which is essential for maintaining the illusion of a single-system image to clients, is to ensure the *atomicity* of operations on the data objects in the system. Stated simply, submissions and responses to and from objects (values) in the CAN should be consistent with an execution in which there is only one copy of the object accessed serially [Lyn96]. Because of the complexity of dynamic systems, and because many CANs assume an environment in which leaves and failures are equivalent, most proposals focus on the first design goal and are designed to make a "best effort" with respect to atomicity. They

violate the atomicity guarantee by allowing stale copies of data to be returned, or skirt around the problem by allowing only write-once semantics.

It is well-known that simultaneously guaranteeing availability (liveness) and atomicity in failure-prone networks is impossible. Therefore, we assume a system in which the network is asynchronous but reliable (messages are eventually delivered) and where servers do not fail. They can, however, initiate a join or leave routine at any time, thus admitting possible concurrent modifications along with concurrent data accesses. We feel that this model is not incompatible with the "real world" because network outages are typically repaired, and crashed machines typically recover (and can retrieve committed information from stable storage), or are actively replicated so that partial crashes are not visible externally. Therefore the appropriate use of transactional storage, reliable message queues, and server redundancy at the system level can vastly simplify the environment seen at the algorithmic level. This also allows such mechanisms to be removed when the CAN algorithm is deployed in settings with stronger environmental assumptions. It should be noted that developing an atomic CAN in this model does not preclude the system from returning stale data if this is useful to the application when the system is unresponsive.

Given these strong assumptions, we seek an algorithm that yields atomic access to data when there is only one copy of each piece of data in the system. The challenge will be to ensure that as data migrates (when nodes join and leave), requests do not access a residual copy nor do they arrive at the destination before the data is transferred and mistakenly think the data does not exist. Another challenge is to ensure that once a request has been initiated by a node, a result is eventually returned. Because we have assumed an asynchronous network, we must ensure that requests are not forwarded to machines that have left the system (and thus will never respond). Furthermore, we must ensure that routing information is maintained so that requests eventually reach their targets as long as there is some active node.

2 Guarantees

The goal of a CAN is to support three operations: join, leave and data update. Joins and leaves are initiated by nodes when they enter and leave the service. An update is a *targeted request* initiated at a node, and is forwarded toward its target by a series of *update-step* requests between nodes.

join(m): This operation is initiated by a node wishing to join the network, and includes as an argument the physical machine address of a currently active node.

leave(): This operation is initiated by an active node wishing to leave the network.

update(op,x): This operation is initiated by an active node wishing to perform a data operation, *op*, on a value in the CAN that is stored under the logical identifier x.

As far as liveness is concerned, we are primarily interested in the behavior of the algorithm when the system is quiescent (when only a small number of concurrent joins and leaves are occuring during a sufficiently long period and not all nodes have tried to leave the system).

Stated informally, we require that the system guarantee the following properties:

Atomicity: Updates to and the corresponding values read from data objects must be consistent with a sequential execution at one copy of the object.

Termination: If after some point no new join or leave operation is initiated, then all pending join and leave operations must eventually terminate and all updates eventually terminate (including those initiated after that point).

Stabilization: If after some point no new join or leave operation is initiated, then the data and link information at each node should eventually be the same as that prescribed by the chosen hashing and routing schemes,

with the expected lookup/update performance.

It should be noted that these conditional guarantees could be refined to include some notion of *spatial independence* that would highlight the fact that many CAN algorithms, including the one we propose here, allow for concurrent progress of updates even while join and leave operations make some data temporarily unavailable. However it can be seen that the above definitions do not readily admit trivial solutions, and therefore suffice for this paper.

3 Algorithm

In this section we describe an algorithm that implements the guarantees described above. Here we focus on a particular implementation that stems from Chord [SMK+01]. In this implementation, objects and nodes are assigned logical identifiers from the unit ring, $[0,1)$, and objects are assigned to nodes based on the *successor* relationship which compares object and node identifiers. Moreover, nodes maintain "edge" information that enables communication (e.g., IP addresses) to some of the other nodes. Specifically, nodes maintain edges to their successor and predecessor along the ring (they can also keep track of other *long-range contacts* with only a simple modification to the algorithm). We augment the basic ring construction of Chord to include support for atomic operations on the data objects, even in the face of concurrent joins and leaves.

Each node, n, keeps track of its own status (such as *joining, active, leaving*, etc.), and its identifier. Every node maintains a table of physical and logical identifiers corresponding to its "in-links" and "out-links." In-links are nodes from which requests are allowed to be entered into the input queue. Out-links are nodes to which a connection has been established and requests can be forwarded. The data objects controlled by the node are kept in a local data structure, *data*. Each node accumulates incoming requests and messages in a FIFO queue, InQ. Requests in the InQ include all action-enabling requests, including self-generated leave and join requests, other nodes' requests involved with their joining or leaving, and data update requests. Figure 1 illustrates a single node with its local data structures.

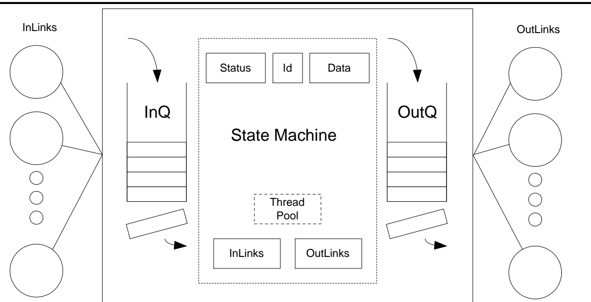

Figure 1: In our node model, requests are performed in a serial manner at each node, with its execution determined entirely by the order in which external events are received.

Each node has a simple dispatch loop, which takes a message off the InQ and runs the appropriate procedure for that message, awakens any suspended procedure waiting for that message, and checks if any suspended procedures can be run due to a change of status. Thus each procedure will be initiated from some message arriving on the InQ, may produce outgoing messages between waiting points — when control is returned to the dispatch loop — and will eventually terminate. Only a single procedure has control at any time.

We now describe the algorithm at a high level. (The appendix provides a psuedocode description of the actions performed by each machine.) We assume that the system is initialized with one or more nodes with an initial set of edges and data objects. We will describe the LEAVE, JOIN, and UPDATE-STEP operations in order below.

When a node wishes to leave, it first changes its status to *transferring*, and stops handling pending requests; incoming requests meanwhile accumulate in the input pending queue. The node then atomically transfers its data to its successor in a nice big message. From this point on, it changes its status to *leaving*, so that all requests that would be meant for the current node will be forwarded along to the successor. After

transferring the data, the node sends a "leaving" message to its in-links (predecessor and any others) informing that it is going away. These nodes will route "connecting" requests on the network to add an edge to their new closest active machines as a replacement for the leaving edge. When the "connecting" request finally reaches the closest active successor, it is processed, a "connecting" acknowledgment is returned, and once processed, the new edge is added. When the new edge is added, the leaving node receives an acknowledgment that the edge to it is removed, so that no more requests will be forwarded along this edge. When the leaving node has collected "leaving" acknowledgments from all in-links, and it has no more pending requests requiring a response, then it can drop out of the system. This operation is illustrated in Figure 2.

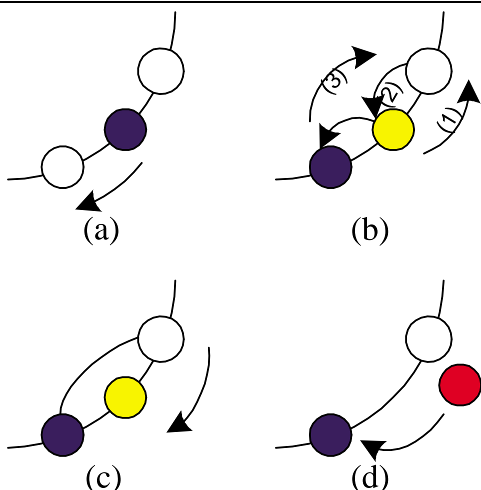

Figure 2: For a leave operation (a) the leaving node transfers its state to its successor (b) it tells its predecessor to find its new active successor [1], which the successor does by submitting a targeted request [2] that is forwarded until a response [3] from an active node is returned, (c) an edge is added, and the leaving node is informed that its in-edges are flushed, and (d) it drops out.

A joining node will attempt to send a join request to a node for which it has a priori knowledge.[1] The request, if the node has not left, will be acknowledged and atomically put on the queue with the rest of the requests. The join message, similar to other *targeted requests*, will be routed around the system until the closest active target processes the message. At this time, the target node will separate its data, modify its bucket, and send a big message to the joining node that it is processing. It will also not be allowed to leave or handle other joins until the entire join procedure has completed. It creates a *surrogate* pointer to the joining node so that all requests for the new joining node are forwarded along this link during this period. It then contacts each of its in-neighbors telling them to update their pointers to the new node. Each of them sends a "connecting" message to the new node, updates its out-neighbors table, then sends an acknowledgment to the host node after removing the host from its out-neighbors table. When the host node collects acknowledgments from all of the neighbors in question, it can remove its surrogate pointer and start entertaining more join requests. This is illustrated in Figure 3.

When an UPDATE-STEP is invoked on a node, it is either forwarded or processed locally depending on its target identifier. When a response is generated, it is sent back to the return address specified in the request. All other targeted requests are either forwarded or processed locally depending on their target identifier.

4 Discussion

Certain aspects of our algorithm are worth mentioning. First, the particular choice of a ring structure as the underlying routing/hashing scheme was somewhat arbitrary. The presented algorithm is readily adaptable to other routing schemes, such as the d-dimensional torii described in [RFH+01] (of which the connected ring is a special case). They are also adaptable to ring-based routing schemes that include different long-range edges from Chord such as in [MNR02].

[1] Because this information may be stale, a node might never succeed in joining. However, if the joining node has knowledge of some active node, joins will complete, and in any case they will not disrupt the safety properties of the system.

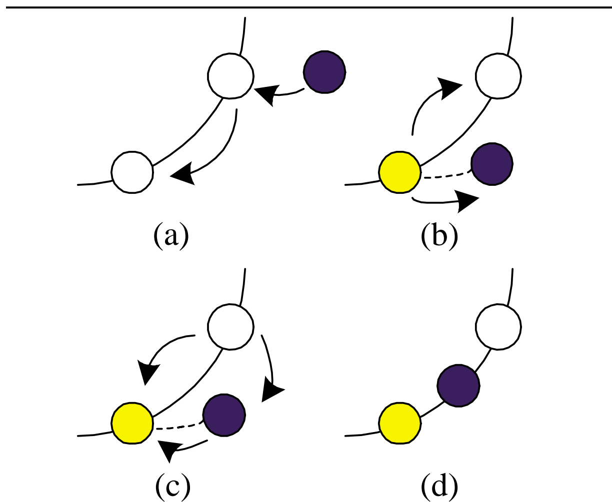

Figure 3: For a join operation (a) the joining node initiates a targeted request to the successor, where (b) a response is returned to the joining node including the relevant state and the predecessor is notified of a new node. At this point the joining node has a surrogate edge pointing to it. After this, (c) the predecessor will contact the joining node to add an edge, it will remove an edge to the old successor, and the surrogate edge will then be removed (d).

Another aspect is that we have modeled each node as a state machine dependent only on the order of its inputs. This means that we can employ well understood state machine replication algorithms to produce a fault-tolerant version of our algorithm where the abstract nodes of the algorithm are implemented by a replicated set of machines. The details of a full implementation remain to be fully worked out.

References

[GBH+00] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. "Scalable, distributed data structures for Internet service construction. In the *Fourth Symposium on Operating System Design and Implementation (OSDI 2000)*, October 2000.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LNS96] W. Litwin, M.A. Neimat, D. A. Schneider. "LH*-A scalable, distributed data structure". *ACM Transactions on Database Systems*, Vol. 21, No. 4, pp 480-525, 1996.

[Lyn96] Lynch, N. *Distributed Algorithms*, Morgan Kaufmann, San Francisco, CA 1996.

[MNR02] D. Malkhi, M. Naor and D. Ratajczak. "Viceroy: A Scalable and Dynamic Lookup Scheme". Submitted for publication.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. "A scalable content-addressable network". In *Proceedings of the ACM SIGCOMM 2001 Technical Conference*. August 2001.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for Internet applications". In *Proceedings of the SIGCOMM 2001*, August 2001.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz and A. D. Joseph. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing". U. C. Berkeley Technical Report UCB/CSD-01-1141, April, 2001.

A Pseudocode

---

local data:
  $ID = \{id, addr\}$, $id \in \mathcal{R}$ randomly chosen, *addr* is a physical address
  InLinks and *OutLinks*, set of $\{id, addr\}$ logical/physical address pairs, initially empty
  data, set of named data objects, initially empty
  $myrange = (low, high) \in \mathcal{R} \times \mathcal{R}$, initially $(ID.id, ID.id)$
  InQ and *OutQ*, FIFO queues containing requests/msgs. *InQ* initially contains JOIN(*someAddr*)
  $status \in \{inactive, joining, active, transferring, leaving\}$, initially *inactive*

definitions:
  $successor = closest(OutLinks)$;clockwise closest on ring
  "(msg,ID)" a message of type *msg* from a machine with logical/physical address of *ID*

main program:
  do forever
    if a (data-trans-ack,*) is on *InQ* then dispatch any procedure waiting for it
    else if there is any waiting procedure that may resume, dispatch one of them
    else
      remove head request from *InQ*
      if *status* is *leaving* then forward request to *successor*
      else
        dispatch the first procedure waiting for that message (if any)
        else dispatch a new procedure to handle request

LEAVE(): ; handle self leaving
  wait until *status* is *active* ; yield
  $status \leftarrow transferring$
  send ((data-trans,*data*),*ID*) to *successor*
  $myrange \leftarrow (ID, ID)$
  wait for (data-trans-ack,*successor*) ; yield
  $status \leftarrow leaving$
  send (leaving,*ID*) to all machines in *InLinks*
  wait for (leaving-ack,*m*) from all machines in *InLinks* ; yield
  forward all UPDATE-STEP requests in *InQ* to *successor*
  $status \leftarrow inactive$

JOIN(*someAddr*): ; handle join to machine with someAddr
  wait until *status* is *inactive* ; yield
  $status \leftarrow joining$
  send (joining,*ID*) to the machine denoted by *someAddr*
  wait for ((join-ack-and-data-trans,*datainfo*),*surrogate*) ; yield
  send (data-trans-ack,*ID*) to *surrogate*
  include *surrogate* in *OutLinks*
  set *data* and *myrange* based on *datainfo*
  wait for (join-complete,*surrogate*)
  $status \leftarrow active$

UPDATE-STEP$(x, op)_{retaddress}$:
  if x is in *myrange* (contained within $[low, high)$ on the unit ring) then perform *op* on x and send result to *retaddress*
  else forward to *successor*

receive-msg T:
  if T is (data-trans,*m*)
    if *status* is *leaving* forward message to *successor*
    else
      merge *data* and *myrange* with incoming data and range information by taking the union
      send (data-trans-ack,*ID*) to *m*

```
else if T is (joining,m)
    if m is in myrange
        oldstatus ← status
        status ← transferring
        group data and modify range between m and ID into datainfo msg
        send ((join-ack-and-data-trans,datainfo),ID) to m
        wait for (data-trans-ack,m)                              ; yield
        status ← oldstatus
        include m in OutLinks                                    ; surrogate pointer
        send ((notify-of-new,m),ID) to all machines in InLinks
        wait for (new-ack,m') from all machines in InLinks       ; yield
        remove those machines from InLinks
        remove m from OutLinks                                   ; remove surrogate pointer
    else forward request to successor

else if T is (leaving,m)
    send ((connect,m),ID) to successor
    wait for (connecting-ack,substitute)                         ; may differ from successor if it is leaving
    replace m in OutLinks with substitute'
    send (leaving-ack,ID) to m

else if T is ((connect,x),m)
    if x is in myrange
        add m to InLinks
        send (connecting-ack,ID) to m
    else forward to successor

else if T is ((notify-of-new,x),m)
    send ((connect,x),ID) to closest link in OutLinks
    wait for (connecting-ack,new)
    replace m with new in OutLinks
    send (new-ack,ID) to m
```