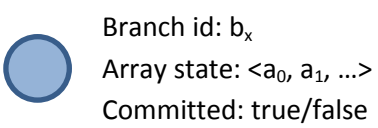
The Branching Append-Only Dynamic Array (BAODA) Data Type

In this document, the *branching append-only dynamic array* (BAODA) data type is described. An abstract distributed algorithm that implements a BAODA is presented, and the algorithm is then stepwise refined into a concrete algorithm.

The BAODA data type is a concurrent data type, designed to be accessed by multiple processes, that is especially well suited to systems with high latencies (e.g. distributed systems). A BAODA is similar to an *append-only dynamic array*, but instead of allowing concurrent appends to the end of the array, a process creates its own branch and appends elements to that branch.

A BAODA is a directed rooted tree, where each node corresponds to the state of an array. An *array state* is a sequence of elements taken from some set. Each node has the following properties:

- A branch identifier,
- An array state,
- A committed flag.

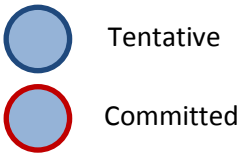


When a BAODA is created, it contains a single *root node*, belonging to the *root branch*. The root node is the only node in the root branch. The array state of the root node is that of an empty array (the sequence of length zero), and the root node is committed. The root node is colored orange in our diagrams.

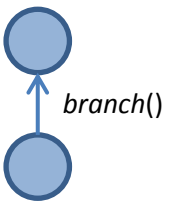
A BAODA must satisfy the following correctness properties:

- **Prefix invariant:** For each node *n* in the tree, each ancestor of *n* has an array state that is a prefix of the array state of *n*.
- **Commit invariant:** All committed nodes lie on the same path.
- **Progress:** If a node is created in the maximum branch, and no greater branch is created after that, then eventually that node will become committed.

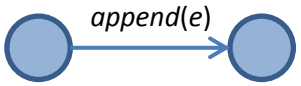
All nodes, except for the root node, are initially non-committed, but may later become committed, as long as the commit invariant holds. We refer to a node that is not committed as tentative. A committed node is visualized with a red border, and a tentative node is visualized with a dark blue border.



A process can create a new node in the tree using one of two operations: *branch* and *append*. The branch operation creates a new branch, and creates an initial node in that branch. During the creation of a branch, a branching point node is found, from which an edge to the newly created node is added. The array state of the initial node is set (copied) to the array state of the branching point node.

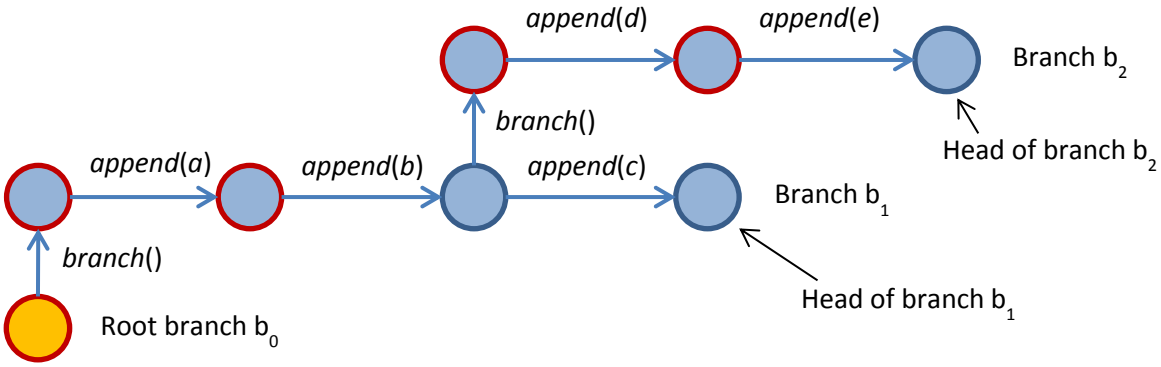


After the initial node in a branch is created, new nodes can be added to the tree by the process that created the branch, by using the append operation, which creates a new node and an append edge from the



previous head of the branch to the new head of the branch. The array state of the new node is the array state of the preceding node with the element appended to it. We draw append edges horizontally, and branch edges vertically, to easily distinguish between them.

An example tree:



A node is uniquely identified by a pair <*b, l*> where *b* is a branch identifier, and *l* is the length of the array state of the node. In the example above, the node <b_2, 3> is the max node that is committed. Because of the commit invariant, node <b_1, 3> can never become committed, and we say that branch b_1 is *dead*, and branch b_2 is *alive*. Node <b_2, 2> is the initial node of branch b_2, node <b_1, 2> is the branching point for branch b_2, and node <b_2, 4> is the head of branch b_2.

As the array state of a node can be found by following the path from the root to that node and accumulating all append operations on the edges leading there, the prefix invariant is preserved by construction. The array state of node <b_2, 4> in the example is <a, b, d, e>.

Nodes are committed asynchronously. The process that created a branch gets a notification if a node becomes committed. A process that has created a branch may get a notification that the branch is dead. The process can, even after the branch is dead, add nodes to the branch by appending elements, but if the process knows that the branch is dead, and those nodes can never become committed, then there is no point in doing so.

A process can access the array state of a node using the *size* and *get* operations: size returns the number of elements in the array state of that node, and get returns a particular element in the sequence.

Using BAODA to implement atomic broadcast

Atomic broadcast (AB) allows a process to broadcast a message. The AB guarantees that all processes (that haven't crashed) will deliver the broadcasted messages, and if one process delivers the messages in one order then all processes will deliver the messages in the same order.

To implement AB on BAODA requires a leader election service, such as Omega. The leader election service suggests, at each point in time, which process is the leader. That process creates a branch in BAODA. When a process wants to broadcast a message, it sends it to the leader, which appends it to the branch that it has created. At first the created node is tentative, but if no other process creates a new branch then eventually that node (with the array state including the message) will be committed, and at that point the message can be delivered in the order that it occurs in the array.

If another process comes to believe itself to be the leader then it will create a new branch. When a new branch is created, that process has to look through the array state of the initial node to see which messages are already been appended to that array state, so that it does not append the message a second time.

A process that broadcasts a message is responsible for resending the message to the new leader if the process finds out that a new leader has been elected, and the message wasn't committed in the BAODA already.

Note that the BAODA naturally lends itself to pipelining. More than one node can be tentative (non-committed), and messages can therefore be appended even before the array state including the previous message has been committed.

An abstract algorithm

We now describe an algorithm that implements a BAODA. There are many possible algorithms that implement the BAODA interface and guarantees. This algorithm is essentially the Paxos algorithm. We first describe an abstract algorithm, and then do stepwise refinements into a concrete algorithm.

This implementation imposes a restriction that a process can access only the array state of the head node of the current branch and the array state of the max known committed node. This restriction makes it possible for the implementation to avoid storing nodes that cannot be accessed any longer.

There are `N` replica processes, and they communicate via perfect FIFO links. `E` is the value type of elements in array states. First we describe the data structures used:

```
struct BranchId: lt: int, pid: int
struct NodeId: bid: BranchId, length: int
struct Node: nid: NodeId, arrayState: Sequence[E]
```

Both BranchIds and NodeIds are compared lexicographically. We define these messages:

```
struct BlockReq: bid: BranchId
struct BlockRes: bid: BranchId, cb: BranchId, msn: Node
struct StoreReq: bid: BranchId, node: Node
struct StoreRes: bid: BranchId, cb: BranchId, nid: NodeId
struct KnownCommittedReq: bid: BranchId, nid: NodeId
```

We define the following constants:

```
val rootBranchId: BranchId = BranchId(0, 0)
val rootNodeId: NodeId = NodeId(rootBranchId, 0)
val rootNode: Node = Node(rootNodeId, [])
```

These variables store information about the BAODA tree:

```
var cb: BranchId = rootBranchId     # Current branch id.
var msn: Node = rootNode            # Max stored node.
var mkc: NodeId = rootNodeId        # Max known committed node id.
```

The following variables are related to leading a branch:

```
var leading: bool = False # True if leading the current branch.
val maxStoredIn: Dict[int, Node] = {}
val nodeStoredBy: Dict[NodeId, List[int]] = {}
```

Every function and message handler implements logical clocks (not shown in the algorithm). A process that wants to create a new branch does:

```
def branch():
   cb = BranchId(logicalTime(), myPid)
```

```
leading = True
maxStoredIn.clear()
maxStoredIn[myPid] = msn
for to in neighbors:
     send(BlockReq(cb), to)
```

A process that receives a BlockReq does:

```
def receiveBlockReq(req: BlockReq, src: int):
   if cb < req.bid:
        if leading:
             leading = False; notify('Branch abandoned', cb)
        cb = req.bid
   send(BlockRes(req.bid, cb, msn), src)
```

A process that receives a BlockRes does:

```
def receiveBlockRes(res: BlockRes, src: int):
   if not leading or cb != res.bid: return
   if cb < res.cb:
        leading = False; notify('Branch abandoned', cb); return
   maxStoredIn[src] = res.msn
   if msn.nid.bid < cb and 2 * len(maxStoredIn) > N:
        bp = max(maxStoredIn.values())
        msn = Node(NodeId(cb, bp.length), bp.arrayState)
        nodeStoredBy.clear()
        nodeStoredBy[msn.nid] = [myPid]
        for to in neighbors:
             send(StoreReq(cb, msn), to)
```

A process that has started a branch and wants to append an element does:

```
def append(e: E):
   if not leading or msn.nid.bid < cb:
        return # Not leading or didn't store initial node yet.
   arr: List[E] = list(msn.arrayState)
   arr.append(e)
   msn = Node(NodeId(cb, len(arr)), arr)
   nodeStoredBy[msn.nid] = [myPid]
   for to in neighbors:
      send(StoreReq(cb, msn), to)
```

A process will only store a node in the branch that it considers to be the current branch. When receiving a StoreReq, the process does:

```
def receiveStoreReq(req: StoreReq, src: int):
   if cb == req.bid:
        msn = req.msn
   send(StoreRes(req.bid, cb, msn.nid), src)
```

When receiving a StoreRes, a process does:

```
def receiveStoreRes(res: StoreRes, src: int):
   if not leading or cb != res.bid: return
   if cb < res.cb:
        leading = False; notify('Branch abandoned', cb); return
   nodeStoredBy[res.nid].append(src)
   if mkc < res.nid and 2 * len(nodeStoredBy[res.nid]) > N:
        mkc = res.nid
        notify('Max known committed node was updated', mkc)
        for to in neighbors:
             send(KnownCommittedReq(cb, mkc), to)
```

When a process receives KnownCommittedReq it does:

```
def receiveKnownCommittedReq(req: KnownCommittedReq, src: int):
   if cb == req.bid:
        mkc = req.nid
        notify('Max known committed node was updated', mkc)
```

To access the max stored node, and the max known committed node, we have the functions:

```
def sizeTentative() -> int:
   return msn.nid.length

def getTentative(index: int) -> E:
   assert index >= 0 and index < msn.nid.length
   return msn.arrayState[index]

def sizeCommitted() -> int:
   return mkc.length

def getCommitted(index: int) -> E:
   assert index >= 0 and index < mkc.length
   return msn.arrayState[index]
```

Proof of correctness

We now argue why the above algorithm satisfies the correctness properties.

We define the relation < on nodes, which is the lexicographical ordering on node identifiers. The < relation on nodes is a total order. Node *n* is *between* node n_1 and node n_2 if $n > n_1$ and $n < n_2$. By *parent*(*n*) we denote the node from which there exists an edge in the tree going to *n* (each node has a unique parent in a directed rooted tree).

Theorem 1: For each node *n* in the tree, each ancestor of *n* has an array state that is a prefix of the array state of *n*.

Proof: The edge between the parent of *n* and *n* is either an append edge or a branch edge. If it is an append edge then the array state is a prefix of the array state of *n* (same elements but parent's array state is one element shorter), and if the edge is a branch edge then the array state of the parent is a prefix of the array state of *n* (same elements and same length).

We must prove that the commit invariant is satisfied. We state the following lemma:

Lemma 2: For every node *n*, no node between *parent*(*n*) and *n* is, or will ever become, committed.

Proof: We must prove lemma 2 for two cases: when the edge from *parent*(*n*) to *n* is an append edge, and when it is a branch edge. The append edge case is trivial, as there are no nodes between *n* and its parent. The branch edge case is slightly more involved.

Node *n* is the initial node of a branch. Let *b* be the branch id of node *n*, and let *p* be the process that created *b*. Before the initial node in *b* is created, *p* sends block requests to all neighbors, and waits until a majority of processes have responded saying that they have blocked all branches below *b*, and also says what is the greatest node stored in a branch below *b*. Let *m* be the greatest node reported by any of those processes. This means that a majority of processes guarantee that they will not store any node between *m* and *n*. Even if a node between *m* and *n* is stored in a process that hasn't yet responded to the block request, that node can be stored in, at most, a minority of processes, and can therefore never become committed.

Theorem 3: All committed nodes lie on the same path.

Proof: Consider the maximal committed node in a tree. Going back one step to its parent, we see, by lemma 2, that no committed nodes can exist between those two nodes. The parent may or may not be committed. Then recursively keep going backwards towards the root, and we see that no committed node can lie off that path.

Theorem 4: If a node is created in the maximum branch, and no greater branch is created after that, then eventually that node will become committed.

Proof: The max branch will not become blocked, and eventually the node will be stored in a majority of processes, and therefore be committed.

A more effective algorithm

The abstract algorithm above is inefficient in that it sends complete array states with every node, and the array states can be arbitrarily large.

Fixing this is relatively straight forward. The messages that contain unbounded data are BlockRes and StoreReq.

We change the BlockRes message to:

```
   struct BlockRes: bid: BranchId, cb: BranchId, msnId: NodeId, mkc: NodeId
```

A process that receives a BlockReq responds with:

```
send(BlockRes(req.bid, cb, msn.nid, mkc), src)
```

That is, instead of the whole max stored node, it sends simply the node id of the max stored node. The response also contains the id of the max known committed node. The process that is creating a branch will be able to decide a branching point node without having the array state of the branching point node. When the branching point has been found, the process either already stores the branching point locally, or it can go and download the array state for the branching point from a process that stores it. Ideally, a process decided to create a new branch if it believes that it already stores what will become the branching point.

For the StoreReq message, we also don't want to send the entire array state of the sent node. Instead we can base the new array state off of the array state of a node that the sending process knows that the receiving process already stores. The StoreReq is changed to:

```
   struct StoreReq: bid: BranchId, nid: NodeId, copyFrom: NodeId, length: int, suffix: Sequence[E]
```

The process receiving the StoreReq creates a node with id nid, and then creates the array state by copying the length first elements from the array state of the node with id copyFrom, and then appends the elements in suffix, to get the array state for the stored node.

The process sending the StoreReq will decide which node that the receiver should copy from as follows:

- If the sent node is the initial node of a branch, the sender knows what is the receiver's max stored node, and max known committed node. It picks the one of those two that it knows is on the path to the sent node.
- If the sent node is a node that was created using an append operation, the node to copy from is the parent node of that node. As channels are FIFO, the sender knows that the receiver must have stored that node already (unless that node was blocked, in which case the currently sent node will be blocked as well).