RAMBO for Dummies

Jeannie R. Albrecht[1], Yasushi Saito
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto

HPL-2005-39
February 24, 2005*

RAMBO, consensus, distributed computing

RAMBO and RAMBO II are algorithms for emulating shared memory in a distributed asynchronous environment with nodes joining and leaving the system dynamically. This report tries to describe RAMBO and RAMBO II to people without background in I/O automata.

* Internal Accession Date Only

[1] Dept. of Computer Science, University of California, San Diego

Approved for External Publication

RAMBO for Dummies

Jeannie R. Albrecht and Yasushi Saito

Abstract

RAMBO and RAMBO II are algorithms for emulating shared memory in a distributed asynchronous environment with nodes joining and leaving the system dynamically. This report tries to describe RAMBO and RAMBO II to people without background in I/O automata.

1 Introduction

In dynamic, distributed environments where storage devices may join or fail at any time during the course of computation, ensuring liveness, fault tolerance, and availability is usually achieved through various levels of data replication. This distributed replication is used to perform atomic register emulation. Atomic register emulation is the process by which atomic accesses to shared memory are emulated in message-passing systems. The result is a distributed system that ensures linearizable accesses to the object, i.e., a system that gives the illusion of having shared global memory, but in reality actually shares nothing. The concept of atomic registers have recently been proved useful in building distributed, redundant storage systems, such as FAB [9] and DStore [4], and SSM [7].

Atomic registers were first introduced in [1] by Attiya, Bar-Noy, and Dolev. They used a protocol that was based on static quorums of participants to guarantee consistency. However they assumed that the system would not undergo major membership changes, which proved to be a limitation.

Lynch and Shvartsman extended the idea further and introduced a consensus protocol based on a dynamic quorum that allowed reconfigurations and major membership changes. RAMBO [8] (Reconfigurable Atomic Memory for Basic Objects) and RAMBO II [2] provide formal specifications for reconfigurable atomic shared memory as a global service based on dynamic quorums. These systems allow linearizable read and write accesses [3], even when nodes join and leave the system dynamically.

The original RAMBO papers are written using I/O automata, which is difficult to crack for people who are not intimately familiar with them. Our report tries to explain these algorithms in layperson's terms, in the hope that the RAMBO algorithms receive the attention they deserve.

2 Overview

RAMBO consists of two components (algorithms) running independently. One is the *Reader-Writer* component, which is the core of RAMBO, and the other is the *Recon* component, which is responsible for initiating a reconfiguration. The key underlying ideas behind these components are listed in this section.

> *Note: The original RAMBO paper defines a third component, "Joiner", that bootstraps a new node into the system. We omit this component in this report.*

- The system consists of a set of nodes that communicate through a fully connected network. Some of these nodes store replicas of the object. We do not assume that the set of the nodes in the system is fixed—nodes can be be added or removed over time, and RAMBO handles these events by changing the quorum system dynamically.
- Each replicated object stores a totally ordered *tag* that shows the "newness" of the replica. The tag acts as a Lamport clock. The algorithm ensures linearizable read and write accesses to the object in the order of the tag.
- A read or write operation uses a variation of traditional atomic-register emulation algorithm [1]. Both types of operations run in two phases. In the first "Query" phase, the request coordinator discovers the newest value and the tag from a quorum of nodes. In the second "Propagate" phase, the coordinator writes the new value and tag to a quorum of nodes.
  The difference between RAMBO and [1] is in the definition of "quorum". The algorithm described in [1] assumes the fixed set of nodes that replicate an object. In contrast, RAMBO allows the set to drift over time. For this purpose, RAMBO allows multiple quorum systems, or *configurations*, to coexist simultaneously, as described next.
- *Reconfiguration*, or the act of changing the set of nodes that replicate the object, happens asynchronously from read and write operations. Each replicated object stores the variable *cmap*. *cmap* is a list of *configurations*, each describing the set of nodes that are considered live.

Reconfiguration starts when someone adds a new configuration to the *cmap*. This "someone" is the Recon component. As we elaborate more in Section 4, for all practical purposes, we can assume that Recon is a Paxos-based state machine [5, 6], running on the set of all possible nodes in the system, independently from RAMBO's main Reader-Writer component.

- When there are multiple configurations in *cmap*, the older ones must be removed eventually.
  RAMBO provides a background garbage-collection (GC) algorithm for this purpose. In the original RAMBO algorithm, the GC component runs sequentially—it removes the oldest configuration in the list (on a particular node) before removing the next. On the other hand, RAMBO II allows removing multiple configurations at once. This is the only difference between RAMBO and RAMBO II.
  GC process must ensure linearizability, that is, the users observe successively newer values even when the set of nodes that store the object change over time. To achieve this goal, the GC algorithm runs the two-phase voting algorithm, described previously, to find the newest tag and value from a quorum of the old configuration(s), and writes that value back to a quorum of the new quorum. This way, any read or write operation that starts in the future will be able to discover a value written in any GCed configuration.

3 Data structures

Algorithm 1 Variables on each node

```
type
  Tag = record
    ts: Timestamp                                  {Lamport clock}
    node: NodeID                                   {Tie breaker}
  Configuration = record
    r: a set of set of nodes                       {Read quorum system}
    w: a set of set of nodes                       {Write quorum system}
var
  tag: Tag, initially [0, NIL]
  val: Value, initially some agreed-on value
  cmap: a set of {int ↦ Configuration}, initially {i ↦ ⊥}
  for all integer i.
```

Algorithm 1 shows the persistent variables kept for each replica of the object.

3.1 Tag

Type **Tag** stores a timestamp and the ID of the node that generated the tag. It is used to uniquely identify different versions of the object. The RAMBO algorithm ensures that tags assigned to a replica increase monotonically.

3.2 Value

Type **Value** represents the object value. RAMBO is agnostic about the structure of the value; it's anything that can be read and written locally and atomically.

3.3 Configuration

Type **Configuration** identifies a particular membership, i.e., the list of nodes that are considered to be live and replicate the object. The "r" field defines the quorum system used in the first "Query" phase, and the "w" field defines the quorum system used in the second "Propagate" phase. A *quorum* is a generalization of a majority—it defines the set of nodes from which an operation must receive replies to make progress. A *quorum system* is a set of quorums—it defines the possible set of combinations of respondents that allow an operation to make progress. The two fields in any configuration c must satisfy the following *quorum-intersection* property:

$$\forall q \in c.r,\ \forall q' \in c.w \bullet q \cap q' \neq \phi \qquad (1)$$

> *Note: RAMBO does not require two write quorums to intersect, because it only needs to ensure linearizability between an operation* α *that completes the "Write" phase, and another operation* β *that starts the "Read" phase.*

We use the symbol C to denote the set of all possible configurations, and use the phrase "a valid configuration" to refer to any element in C. In addition, RAMBO uses the following two special configurations that are not in C. Their uses are discussed in more detail in Section 3.4.

- "$\mp$" denotes a configuration that was installed in the past, but has been garbage-collected since then.
- "$\perp$" is a placeholder for a configuration that is yet unknown.

3.4 cmap

Variable *cmap* describes the list of configurations known to a node. It is a mapping from a sequence number (0, 1, ...) to a configuration. Initially, *cmap* does not contain any valid configuration. For notational convenience, we represent this state by setting $\perp$ for every possible index in *cmap*.

The Recon component (i.e., Paxos; Section 4) adds a valid configuration after a membership change. The garbage-collection algorithm (Algorithm 3) removes a configuration from *cmap* by setting it to $\mp$.

Thus, any entry in *cmap* will transition, over time, from $\perp$ to a valid configuration to $\mp$. For this reason, we define the following partial order between configurations to simplify our presentation.

$$\forall c \in C, \perp < c < \mp. \tag{2}$$

4 Recon component

The Recon component is responsible for generating new configurations. RAMBO proposes using a concensus protocol, such as Paxos [5, 6], for the Recon component. We find this to be a reasonable suggestion.

From a theoretical viewpoint, the only thing that RAMBO requires from the Recon component is a slightly stronger form of eventual agreement: once two nodes learn about the identity of the k'th configuration (for whatever k), they must agree on what that is. In other words, for any k, and any two nodes i and j with their corresponding $cmap_i$ and $cmap_j$, the following must hold at all times.

$$\begin{aligned} &cmap_i[k] \in C \wedge cmap_j[k] \in C \\ &\quad \Rightarrow cmap_i[k] = cmap_j[k] \end{aligned}$$

RAMBO tolerates out-of-order installation of configurations—that is, it's OK for a node to receive the 3rd configuration before receiving the 2nd one—but neither read/write operation processing nor configuration garbage-collection will make progress with such a "gap" in the *cmap* (Section 6).[1] Thus, for all practical purposes, the Recon component should install the configurations in the order of the indexes at each node. Paxos can ensure this property easily. We will discuss the significance of gaps in more detail in Section 6.

Even with Paxos, nodes receive configuration updates asynchronously with respect to read, write, or GC operations. Thus, during a read or write operation, a node may not yet know about configurations other nodes have already learned. RAMBO handles this situation by piggybacking the node's *cmap* on every read or write message and letting the recipient update its *cmap* accordingly. Old configurations in the *cmap* are eventually removed by a garbage collection service (see Section 5).

5 Reader–Writer component

Algorithm 2 shows the algorithm for reading and writing the object. Algorithm 4 shows the "backend" algorithm that responds to the messages from coordinators.

A read/write operation can be initiated by any of the nodes that store the object and the associated *cmap*. Each operation runs in two phases: a query phase and a propagation phase.

In the query phase (line 17), node i, the initiating node, contacts all read quorums found its *cmap* to determine the most recent available **tag** and **value**. This ensures that the operation discovers a value that is no older than those seen by prior completed operations.

In the propagation phase (line 27), node i contacts write quorums. If the requested operation was a "read", the propagation phase propagates the largest tag and its associated value that was discovered in the query phase, and all nodes in the write quorums update their local tags and values accordingly. If the requested operation was a "write", node i creates a new tag that is greater than the most recent tag discovered during the query phase, and the operation propagates the new tag and new value to the write quorums. The purpose of the propagation phase in the case of a write operation is to make sure "enough" members acquire the new tag and value pair.

Line 39 checks whether there is a gap in the *cmap*. A *cmap* is *gapless* if, for some i and j ($i < j$), $cmap[0]$ through $cmap[i]$ are all $\mp$, $cmap[i+1]$ through $cmap[j]$ are all valid, and $cmap[j+1]$ and beyond are all $\perp$.

In addition to reading and writing, the Reader–Writer component is also responsible for garbage collection, or removing old configurations from the system. Algorithm 3 shows the garbage-collection algorithm. Like reading and writing, garbage collection requires both a query phase and a propagation phase. The query phase contacts read and write quorums from the old configuration that is going to be removed. This allows the initiating node to gather tag and value information from both read and write quorums, in addition to making sure that all read and write quorums know which configurations have already been garbage collected. The propagation phase then contacts a write quorum from the new configuration. This propagates the newest tag and value to the write quorums in the new configuration to ensure that they have all acquired the most recent information.

Note that the garbage collection component, unlike reader-writer component, uses read and write quorums of only two configurations in the *cmap*. Further, it is important to realize that garbage collection and reading–writing can run in parallel. This is due to the fact that $cmap[k]$, for any k, moves only from $\perp$, to some valid configuration in C, to $\mp$.

[1] Note: The "gapless" property is called *"Truncated"* in the original RAMBO paper.

Algorithm 2 Reading and writing the object

```
12: procedure doRead()                          {Entry point for "read"}
13:   return doIo(NIL)
14: procedure doWrite(newVal)                   {Entry point for "write"}
15:   doIo(newVal)
16: procedure doIo(newVal)
17:   replies, opCmap ←ϕ, cmap                  {Query phase start}
18:   repeat
19:     sendReceive([Query, cmap], replies, opCmap)
20:   until received replies from a read-quorum of each valid
      configuration in opCmap
21:   if newVal = NIL then
22:     newTag ←maximum tag replies             {Read operation}
23:     newVal ←value corresponding to newTag
24:   else
25:     newTs ←max(tag in replies).ts + 1       {Write operation}
26:     newTag ←[newTs, MY_ID]
27:   replies ←ϕ                                {Propagate phase start}
28:   repeat
29:     sendReceive([Propagate, newTag, newVal, cmap],
        replies, opCmap)
30:   until received replies from a write-quorum of each valid
      configuration in opCmap
31:   return newVal
32:             {Send a request & receive replies from quorums in cmap}
33: procedure sendReceive(msg, replies, opCmap)
34:   Send msg to all nodes
35:   Wait for a while and add replies to replies
36:   for all reply ∈ replies do
37:     updateCmap(cmap, reply.cmap, true)
38:     updateCmap(opCmap, reply.cmap, false)
39:   if opCmap has a gap then
40:     replies ←ϕ
41:                  {Update cmap by computing elementwise maxima}
42: procedure updateCmap(cmap, newCmap, doGc)
43:   for all $(n \mapsto c) \in$ newCmap do
44:     if $c >$ cmap[n] then
45:       if ¬doGc and $c = \mp$ then Do nothing
46:       else cmap[n] ←c
```

Algorithm 3 Removing an old configuration in RAMBO-I

```
47: procedure gc(k)                     {GC the configuration cmap[k]}
48:   precondition
49:     $\forall i < k$, cmap[i] $= \mp$
50:     $\{cmap[k], cmap[k+1]\} \subseteq C$
51:   old ←cmap[k]                      {Configuration to be removed}
52:   new ←cmap[k+1]                    {Next configuration in cmap}
53:   replies ←ϕ                        {Query phase start}
54:   repeat
55:     gcSendReceive([Query, cmap], old, replies)
56:   until received replies from read- and write-quorum of old
57:   newTag, newVal ←maximum tag and value in replies
58:   replies ←ϕ                        {Propagation phase start}
59:   repeat
60:     gcSendReceive([Propagate, newTag, newVal, cmap],
        new, replies)
61:   until received replies from a write-quorum of new
62:   cmap[k] ←$\mp$
63: procedure gcSendReceive(msg, targets, replies)
64:   Send msg to targets
65:   Wait for a while and add replies to replies
66:   for all reply ∈ newly received replies do
67:     updateCmap(cmap, reply.cmap, true)
```

Algorithm 4 Passive part of the algorithm

```
68: when receive [Query, newCmap]
69:   updateCmap(cmap, reply.cmap, true)
70:   send reply [value, tag, cmap]
71: when receive [Propagate, newTag, newVal, newCmap]
72:   if newTag > tag then
73:     tag, value ←newTag, newVal
74:     updateCmap(cmap, reply.cmap, true)
75:   send reply [cmap]
```

6 Why does a cmap need to be gapless?

Part of the complexity in RAMBO is that it allows the Recon component to create a gap in the *cmap*. This looks like purely an academic artifact, in that a protocol like Paxos can easily avoid creating such gaps. But the original authors allowed gaps anyway, and they handle them roughly by stopping and restarting operations once a gap is found.

The problem with a gap is that it may create a "split-brain" behavior in which two values are read or written on disjoint sets of nodes. Consider the following scenario with six processes p_1 to p_6. Define read and write quorums to be a majority of the configuration. Let's assume that at some moment, the *cmaps* of these nodes are like below:

$$cmap_{p_1} = cmap_{p_2} = cmap_{p_3} = \{0 \mapsto \langle p_1, p_2, p_3 \rangle\}$$
$$cmap_{p_4} = cmap_{p_5} = cmap_{p_6} = \{1 \mapsto \langle p_4, p_5, p_6 \rangle\}$$

Without the gap checking in line 39 of Algorithm 2, one set of read/write requests could use only the configuration $\langle p_1, p_2, p_3 \rangle$, and another set of read/write requests could use only only the configuration $\langle p_4, p_5, p_6 \rangle$.

RAMBO solves the "gap" problem simply by restarting the operation, and not starting the GC of a configuration just before the gap. An alternative, simpler solution would have been to demand that the Recon component installs configurations in the order of their indexes.

7 RAMBO II

In RAMBO, garbage collection is done sequentially. This means that nodes remove obsolete configurations one at a time, in order, until only the most recent configuration remains. This technique performs poorly when communication is unreliable, or when reconfiguration is frequent. RAMBO II [2] aims to fix this problem by allowing old configurations to be removed in parallel.

Most of the pseudocode for the read and write operations in RAMBO is still used in RAMBO II. However the garbage collection phase in RAMBO is replaced with a new configuration upgrade operation. The pseudocode for this method is shown in Algorithm 5.

Algorithm 5 The configuration garbage-collection algorithm for RAMBO-II

```
procedure upgrade($k$)                    {GC configurations up to $k$}
  $replies \leftarrow \phi$
  repeat
    sendReceive([Query, $cmap$], $replies$)
  until (received replies from read- and write-quorums for $cmap$[j] for $\forall j < k$)
  $newTag$, $newValue$ $\leftarrow$ maximum tag and value in $replies$
  $replies \leftarrow \phi$
  repeat
    sendReceive([Propagate, $newTag$, $newValue$, $cmap$], $replies$)
  until (received replies from a write-quorum in $cmap[k]$)
  for $\forall j < k$ do
    $cmap$[j] $\leftarrow \mp$
```

8 Conclusion

This paper described RAMBO and RAMBO II algorithms for readers without deep background in the theory of distributed computing. RAMBO is an algorithm for emulating replicated shared memory in a distributed shared-nothing environment. It supports dynamic changes to the set of replicas by allowing multiple configurations to coexist and removing older configurations in order. RAMBO II improves RAMBO by supporting removing multiple configurations at once. These algorithms are simpler than they look in the original papers [8, 2]. We hope that this paper managed to convey the essence of these algorithms.

We have used a variation of RAMBO in the FAB distributed disk array, with changes to support high-throughput, high-capacity storage systems. Interested readers should also consult our companion paper [9].

References

[1] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[2] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 259–268, 2003.

[3] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 12(3):463–492, July 1990.

[4] A. Huang and A. Fox. Dstore: self-managing, crash-only persistent hash table. http://swig.stanford.edu-/public/projects/dstore/, 2004.

[5] L. Lamport. The part-time parliament. *ACM Trans. on Comp. Sys. (TOCS)*, 16(2):133–169, 1998.

[6] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[7] B. C. Ling, E. Kiciman, and A. Fox. Session state: beyond soft state. In *1st Symp. on Network Sys. Design and Impl. (NSDI)*, pages 295–308, San Francisco, CA, USA, March 2004.

[8] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *16th Int. Conf. on Dist. Computing (DISC)*, pages 173–190, Toulouse, France, October 2002.

[9] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *11th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS-XI)*, Boston, MA, USA, October 2004.