Evaluating a Dependable Sharable Atomic Data Service on a Planetary-Scale Network*

Chryssis Georgiou[1], Nicolas Hadjiprocopiou[1], and Peter M. Musial[2]

[1] Dept. of Computer Science, University of Cyprus, Nicosia, Cyprus
[2] Dept. of Computer Science, University of Puerto Rico Rio Piedras, San Juan, PR, USA

Abstract. Practical implementations of atomically consistent read/write memory service are important building blocks for higher level applications. This is especially true when data accessibility and survivability are provided by a distributed platform consisting of networked nodes, where both nodes and connections are subject to failure. This work presents an experimental evaluation of the practicality of an atomic memory service implementation, called RAMBO, which is the first to support multiple reader, multiple writer access to the atomic data with an integrated reconfiguration protocol to replace the underlying set of replicas without any interruption of the ongoing operations. Theoretical guarantees of this service are well understood; however, only rudimentary analytical performance along with limited LAN testing were performed on the implementation of RAMBO – neither representing any realistic deployment setting. In order to assess true practicality of the RAMBO service, we devised a series of experiments tested on PlanetLab – a planetary-scale research WAN network. Our experiments show that RAMBO's performance is reasonable (under the tested scenarios) and under the somewhat extreme conditions of PlanetLab. This demonstrates the feasibility of developing dependable reconfigurable sharable data services *with provable consistency guarantees* on unreliable distributed systems.

Keywords: Atomic Memory Service, Distributed Architecture, Performance Evaluation, Planetary Scale Experiments, Provable Guarantees.

1 Introduction

Implementation of services that ensure data survivability and consistency in highly dynamic environments, where transient and permanent changes and failures may occur, is critical to many important applications. An example of such application is sharing information about the physical environment and status of the shared objectives in civilian operations that follow natural disasters and in military operations in hostile territories. In both of these cases, computer networks are created hastily and the computing, communicating devices (to which we refer as nodes) are often susceptible to physical damage

* This work is partially supported by the European Union project EGEE (#INFSO-RI-031688) and the University of Cyprus.

A. Hua and S.-L. Chang (Eds.): ICA3PP 2009, LNCS 5574, pp. 580–592, 2009.
© Springer-Verlag Berlin Heidelberg 2009

and network irregularities. To this end, theoretical groundwork has been laid by researchers who have addressed this problem in numerous works [10, 3, 22, 2, 16].

In order to achieve survivability in dynamic networks several replicas of data must be stored and maintained at different network locations. This approach introduces parallelism issues such as memory failure, message loss, and dynamic node participation. By dynamic participation we mean that old nodes may depart and fail,and new nodes may join the computation. Therefore, replication of data raises challenges of maintaining consistency among the replicas, handling their manipulation by dynamic sets of data owners as well as accommodating atomic operations on the data. Providing practical implementations of such services is a separate but equally challenging undertaking.

A reconfigurable distributed (linearizable) atomic data service for dynamic systems, called RAMBO, was introduced by Lynch and Shvartsman [17]. RAMBO's novelty originates from the fact that it is the first specification supporting multiple reader, multiple writer (MRMW) access to the atomic data with an integrated *reconfiguration* protocol to replace the underlying set of data owners without any interruption of the ongoing operations. The authors of RAMBO [17] consciously traded mathematical elegance and simplicity of presentation for an inefficient implementation. To make implementation of RAMBO algorithm robust and practical a series of extensions followed [11, 18, 6,8] that improve the efficiency of the service and focused on reducing communication complexity, a better use of computational resources, and improving system liveness in various deployment settings. Implementation of RAMBO algorithm and its extensions were produced by Musial [19, 18] along with preliminary empirical performance results [18]. Additional experimental performance comparison of versions of RAMBO were carried out by Georgiou et al. [9]. However, the empirical results in [18, 9] are obtained by evaluations carried out in LAN settings and do not reflect delays and failures experienced in more realistic deployments. Therefore, in order to asses the *practicality* of a distributed shareable data service with provable consistency guarantees, such as RAMBO, its performance must be evaluated in a dynamic environment where net- work delays and node availability fluctuate over time. In this work PlanetLab [4, 20], a planetary-scale network, is used as the experimental platform.

Contributions. Shareable data services providing consistency guarantees, such as atomicity (linearizability), make building distributed systems easier. This reason, and the observation made in the preceding paragraph are precisely the motivation of this work. To the best of our knowledge, the RAMBO framework is the latest and most versatile specification of an atomic MRMW service for dynamic environments. In addition, implementation of RAMBO algorithms have been verified to *preserve* correctness [18]. Therefore, there is a need to analyze the performance of RAMBO implementation in a realistic deployment setting and asses its *true practical utility*. To this end we con- duct a series of experiments executed on the PlanetLab research network, where we test RAMBO implementation tolerance to node failures, message loss, and sensitivity of geographical distribution of RAMBO nodes on the overall performance of the service. The experiments measure system responsiveness to user requests in terms of system throughput – including read/write operation throughput, reconfiguration throughput, and join request throughput where each of these operation types is discussed in later sections.

The takeaway from our experiments is that RAMBO implementation can perform well in a realistic setting and copes well with failures of individual sites and local communication interrupts. However, in order to achieve good performance some pre-computation is necessary, and we had to parameterize RAMBO implementation with information about network delays and node failure rate. These computations were performed manually based on observation of the target deployment system (i.e., Planet- Lab) behaviors (see Section 4 for details). Our experiments provide a complementary understanding of RAMBO system throughput for the supported operations in an actual and non-controlled deployment environment where many different variables contribute to operation latency. An equivalent theoretical analysis can be daunting. An analytical analysis was performed in [17] where under assumption of a steady-state, such that nor- mal timing behavior exists (or eventually reached) and all messages are delivered within a bounded time δ that is unknown to the algorithm, and that all locally performed actions take zero time. Result of that analysis is that the expected read/write operation latency is bounded by 8δ in the presence of concurrent failures and infrequent reconfigurations, *regardless* of how many nodes participate in the execution. Our experiments demonstrate that the latency of operations is a function of the number of participants, level of activity, and is influenced by constraints of the deployment platform. However operation latency on its own does not provide an accurate picture of system behavior. Therefore, our results are expressed as RAMBO system throughput, which is computed using collected averages of the measured operations on the participating in the experiment nodes.

2 Background

Atomic Memory Services. Several approaches have been used to implement consistent data in (static) distributed systems. Starting with the work of Gifford [10] and Thomas [21], many algorithms have used collections of intersecting sets of object replicas (such as quorums) to solve the consistency problem. Upfal and Wigderson [22] use majority sets of readers and writers to emulate shared memory. Vitányi and Awerbuch [3] use matrices of registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [2] use majorities of processors to implement shared objects in static message passing systems. Extension for limited reconfiguration of quorum systems have also been explored [7, 16]. These systems have limited ability to support long-lived data when the longevity of processors is confined. Virtually synchronous services [5], and group communication services (GCS) in general [1], can also be used to implement consistent data services, e.g., by implementing a global totally ordered broadcast. While the universe of processors in a GCS can evolve, in most implementations, forming a new view takes a substantial time [13], and client operations are delayed during view formation.

RAMBO (Reconfigurable Atomic Memory for Basic Objects), introduced by Lynch and Shvartsman [16], is the first to support multiple reader, multiple writer access to the atomic data combined with a reconfiguration protocol to replace the underlying set of data owners, where this is accomplished without any interruption to the ongoing operations. To achieve survivability, data (represented as an abstract object) are replicated at several locations. To maintain consistency in the presence of small and

transient changes, RAMBO uses configurations of nodes, each of which consists of a set of members plus sets of quorum sets (with specific intersection properties [10]). In order to accommodate larger and more permanent changes, the algorithm supports reconfiguration, by which the set of members and the sets of quorums are modified. Redundant configurations can be removed from the system without interfering with the ongoing read and write operations. The algorithm ensures atomicity and consistency of data despite occurrence of arbitrary patterns of asynchrony, node failure, and message loss. However, without (active) reconfiguration and in the presence of dynamic system behaviors and arbitrary delays read and write operations may not terminate or may experience indefinite delays. Note that consensus algorithms can be used directly to implement an atomic data service by allowing participants to agree on a global total ordering of all operations [14]. In contrast, RAMBO uses consensus to agree only on the sequence of installed configurations, where the non-termination of consensus does not affect the termination of read and write operations.

PlanetLab. PlanetLab is a distributed overlay network for deployment and assessment of distributed planetary-scale network services [4, 20]. As of March 2009, PlanetLab is composed of 977 machines spanning 484 locations worldwide provided by academic and industry institutions. Its resources are divided into slices where each can be viewed as a network of virtual machines. Up to 32 nodes can be assigned to a slice, whilst a fraction of that node's resources (CPU, local disk space, network bandwidth) is allowed to be consumed by a slice. The allocated resources are controlled on a per-slice, per-node basis. Slices expire after one month of their first creation (removing all the slice associated data), but can be renewed an unlimited number of times on a monthly basis. Malicious and buggy services can affect the communication infrastructure and other's slices performance; therefore, strict terms and conditions for providing security and stability in the PlanetLab are enforced. Access to PlanetLab nodes is feasible through SSH, providing encrypted and secure communication. Nodes may be installed or rebooted at any time turning the disk into a temporary form of storage, providing no guarantee regarding their reliability.

3 The RAMBO Algorithm

Following is only a brief foray of the cocktail of RAMBO extensions used in our implementation. The complete details of these specifications can be omitted as they are not integral to this work and we direct the interested reader to [16, 18]. We begin with the failure model assumed in the RAMBO framework.

Failure model. Assumed failure model is asynchronous dynamic distributed system of communicating nodes, where each node has a unique identifier. Nodes may experience stop-failures and arbitrary delays between processor steps, and a non-failed processor performs steps according to the input program. Messages may be lost, arbitrarily delayed, and delivered out of order, but are not duplicated or corrupted. We assume existence of abstract point-to-point (as opposed to physical) communication channels that allow each node to send a direct message to any other node in the system.

Read and Write protocol. As aforementioned, data survivability in the RAMBO framework is ensured via replication, where data is replicated and maintained at a

number of networked nodes. The caveat here is to ensure consistency while supporting atomic read and write operations. Atomicity of operation access is provided by use of quorum systems. The RAMBO algorithm uses *configuration*s that consist of a set of data owners, and a set of quorum sets imposed on these data owners. We now describe how these configurations are used to implement atomic read and write operations.

Both read and write operations are implemented in phases. First phase is called a *query* phase and is identical for both operations. During *query* phase a node contacts and awaits responses from some complete quorum set. Responses contain replica information that is timestamped (using a Lamport clock). When enough replies are collected, the initiating node chooses the value of a replica that is associated with the highest timestamp. Now the node is ready to enter the second phase called a *propagation* phase. In case of the read operation, a node will propagate the replica information with the highest timestamp to some quorum set. Again, when enough responses have been received, then the operation terminates with a read acknowledgment. In case of the write operation, a node will increment the highest timestamp and associate value being written with the new timestamp. The new replica information is then propagated to some complete quorum set, then the writer awaits appropriate responses from that set that once received are followed by a write acknowledgment. All configurations that have been installed but not yet removed from the system that have been discovered during an ongoing phase are used to complete that phase.

Reconfiguration protocol. Dynamic system behavior and failures may result in configurations that are unresponsive or slow. This means that read and write operations may be blocked or delayed. To avoid reaching this point, RAMBO algorithms implement re- configuration, which is a process of replacing old configurations with new ones, where new configuration consists of healthy nodes.

Reconfiguration is a three-phase [6] process, and uses an optimized variant of Paxos [15]. Each phase requires a coordinated effort lead by a leader node – an active node with the highest identifier. The three phases are: a *prepare* phase, during which a ballot is made ready, a *propose* phase, during which the new configuration is proposed, and a *propagate* phase, during which the results are distributed. The *prepare* phase accesses some quorum set of the current configuration, thus learning about any earlier ballots. When the leader concludes the prepare phase, it chooses a configuration to propose: if no configurations have been proposed to replace the current configuration, the leader can propose its own preferred configuration; otherwise, the leader must choose the previously proposed configuration with the largest ballot. The *propose* phase then begins, accessing some quorum set of the current configuration. This serves two purposes: it requires that the nodes in the current configuration vote on the new configuration, and it collects the most recent replica information. Finally, the *propagate* phase accesses some quorum set from the current configuration; this ensures that enough nodes are aware of the new configuration to ensure that any concurrent recon- figuration requests obtain the desired result, and at the same time current configuration is being marked as being obsolete. It has been shown in [18] that reconfiguration has very little effect on operation latency.

Putting everything together. In our experiments we use an implementation of RAMBO framework that is a cocktail of Long-lived RAMBO [8], Domain-RAMBO [9],

Pax-RAMBO [6], and RAMBO w/restricted gossip pattern [12]. Where collectively these improvements attempt to constrain RAMBO's all-to-all gossip, reduce size of messages, remove trailing configurations, and remove the overhead associated with maintaining a multi-cell atomically consistent memory system. Techniques that allow us to claim that our concoction of RAMBO extensions implements an atomic, reconfigurable memory service in dynamic systems can be found in [18], hence we forgo the details of correctness proofs in this document. From this point on whenever we refer to a RAMBO algorithm, we mean an algorithm that implements the RAMBO framework and supports the improvements stated above. RAMBO node specification consists of two kinds of components. The first is the *Joiner* component that implements a simple join protocol. Specifically, the new node broadcasts a join request to nodes that it believes to be active participants of the RAMBO service and awaits at least one acknowledgment of this request. The second is the *Reader-Writer* component that implements read, write, and reconfiguration operation protocols (described above).

Implementation. RAMBO algorithms used in this work are implemented in Java [18]. The individual memory locations of the implemented atomic memory system are represented as Java Objects, but in the experiments these memory locations are instantiated as Java Integers. Hence, a read and a write operation to a memory location is equivalent to reading an integer value and writing an integer value, respectively.

RAMBO algorithms are specified as a composition of non-deterministic automata. Automata, as state machines, can be executed using theoretical models, where there is no enforcement of when certain events (transitions) occur in any given state. Of course, in the implementation of RAMBO progress is important, hence fair scheduling techniques are used to execute (enabled) transitions.

Communication between RAMBO nodes is implemented using Java Sockets and TCP/IP. Messages in RAMBO fluxuate in size over the execution of the algorithm, there is no limit on the message size, and each state message contains many individual bits of information that describe the state of the local replica. The characteristics of messages in RAMBO make use of TCP/IP and Java Sockets to be a reasonable candidate, since messages can be marshalled as Java Objects and an entire object is then included as data of the message, and the process is reversed on the receiving end. Unfortunately, TCP/IP does not allow message broadcasts, hence the periodic all-to-all gossip is implemented as a sequence of direct messages.

User interacts with the RAMBO system either through a command prompt or a graphical user interface. Through these interfaces a user can initiate read, write, and reconfiguration requests – configuration information must be provided manually to RAMBO. For the purpose of the experiments user interface was augmented to automate invocation of operations along with means to collect operation latency information.

4 Experiments

In this section we describe and analyze each of the experiments conducted by running our RAMBO implementation on PlanetLab. We begin with a description of the experimental environment and practical issues that had to be taken into consideration. We conclude the section with presentation of six scenarios that were designed to test performance and robustness of RAMBO implementation under various system demands.

Preparation and limitations. PlanetLab as a planetary research network is exposed to node failures, message loss, varying processing and communication demands. Deployment of RAMBO implementation on PlanetLab required preparations and precomputations in order to reach acceptable performance. Precomputations involve parameterization of the implementation in order to best utilize the required resources – specifically bandwidth and memory.

The PlanetLab environment is unstable which is attributed to the fluctuation in the utilization of its resources by other slices, hardware, software, and network failures.During our experiments we observed node failures as well as node hangups. In particular, on average 10% of the nodes would fail or hangup during execution. However, since RAMBO tolerates failures and delays up to the point of quorum systems being disabled, experiments were able to terminate despite the observed adversities. (The presented results incorporate these node failures and hangups.)

Memory allocation was also an issue. Limitations imposed by PlanetLab on the amount of memory used by RAMBO system dictated which extensions should be used. For instance, use of restricted gossip pattern reduces communication complexity, but also reduces the amount of memory needed to buffer incoming messages that cannot be processed immediately. Of course, such limitations can be averted to a point, and eventually these will prohibit RAMBO system from being able to expand.

Communication delays. RAMBO uses point-to-point messaging to share replica information among its participants and during replica access operations. Hence, the duration of any operation is dependent on the average network message latency. We have measured the ping latency on PlanetLab to be 139 milliseconds. However, the actual message delay between individual RAMBO nodes is larger since TCP and sockets are used. Furthermore, RAMBO's messages vary in size and require marshaling and unmarshaling on send and receive.

Data points. Each presented scenario was run five times and took a period of three to four hours (for all 5 runs). We observed that deviation of the results collected for each scenario and each run was very small, and decided that five runs for each scenario suffices. Therefore, each data point found on the graphs that follow represent an average for each of the five runs.

Performance of the RAMBO system is expressed in terms of throughput – number of operations per second. Specifically, we test the system for each of the supported operation types: read and write, reconfigure, and join. However, we test these in isolation, meaning that for each scenario only one operation type is being invoked by the selected service participants. At each node we measure the average operation latency, where each operation is performed 400 times per each run. The averages are used to compute the system average throughput.

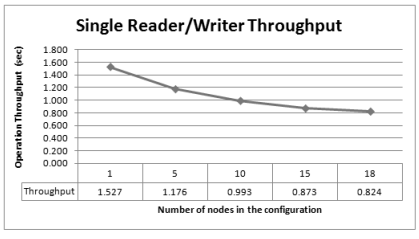

Fig. 1. Single reader/writer scenario

4.1 Scenario 1: Single Reader/Writer

In this scenario 18 nodes join the RAMBO system and do not depart voluntarily. During the experiment the number of nodes participating in the configuration is increased from

one to the maximum possible, and majority quorum system is used. A single node is used to issue read requests, then separately write requests. The memory system consists of two cells. The reads are performed on the first cell and the writes on the second cell. Fig. 1 depicts average system throughput for both read and write operations.

Since redundancy introduces communication overhead we expected to observe decrease in throughput as configuration size increases. Reason being that the termination of each operation depends on ability to communicate with some majority of the configuration members. Hence, larger configurations consist of larger majorities and require more message exchanges. This expectation is validated by the experimental data.

However, it is interesting to observe that the rate of throughput degradation decelerates after configuration size 10. We conjecture that in this scenario the operation throughput reflects the average communication throughput between the node initiating operations and the configuration members. In fact we expect that the operation throughput will stabilize until the point when the initiating node will become a communication bottleneck. However, since only 18 PlanetLab machines were available to us, this conjecture is unsupported by experimental data.

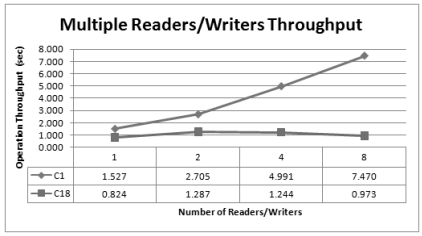

Fig. 2. Multiple readers/writers scenario

4.2 Scenario 2: Multiple Readers/ Writers

This experiment tests RAMBO's sensitivity to different system loads as well as operation concurrency. This is accomplished by gradu- ally increasing the number of nodes initiating read and write requests. This time, we used two configurations, one with configuration of size one (consists of one node), and then with configuration of size eighteen (denoted as C1 and C18 respectively). As in the previous experiment, RAMBO is used to implement two-cell memory system. Each node initiates read and write requests as was described in the previous scenario (reads in the first object and writes on the second). Hence, as the number of readers (resp. writers) increases so does the read and write operation concurrency. Fig. 2 depicts the average system throughput for C1 and C18.

The communication load experienced by the replica owners increases with addition of readers and writers. However, we do not expect this fact to be of significance since the number of nodes participating in the experiment is modest; therefore, the only member of configuration C1 will not be a bottleneck. Moreover, we expect that system throughput when C18 is used will be less than that of C1, since the node performing an operation must wait for responses from some majority (at least nine nodes). This speculation is supported by the experimental data.

It was expected that while using C1 the system throughput would increase with the number of nodes performing operations, however, this pattern is expected to stop as the only configuration member of C1 eventually becomes overwhelmed by the communication burden. In comparison, system throughput remains constant when C18 is used. The behavior in Fig. 4.2 can be explained by the following two observations: (a) replica owners experience light communication load since there are few reader/writers, and (b)

since at least nine nodes need to be contacted per each communication round the system throughput is dependent on the average communication throughput of PlanetLab. We do not expect this to hold as the number of reader/writers is drastically increased.

4.3 Scenario 3: Random Operations with Multiple Readers/Writers

In the previous scenarios read and write operations are conducted separately and are performed on fixed cells, meaning that a node repeatedly invokes reads on memory cell one, and later invokes write operations on cell two. In the first scenario there is no concurrency. In the second scenario, concurrency is limited to overlapping read (resp. write) operations that are invoked in parallel at different nodes.

In this experiment we test a more realistic setting by allowing each node to initiate read and write request at random to each memory cell. Note that each node can initiate only one operation at the time (either a read or a write). This approach allows read and write operations to overlap in time and possibly on the same cell. Additionally, experiment is performed using C1 and C18 in order to compare the observed system behaviors to that of Scenario 2. Fig. 3 depicts the average system throughput, performing read /write randomly and concurrently.

One would expect concurrency to increase the average execution time, as an operation request would possibly "delay" some other operation request (especially with over-lapping read and write requests). However, our experiments suggest that RAMBO's performance is not affected by operation concurrency, where the results plotted in Fig. 3 do not deviate from the ones presented in Fig. 2, except for the run with eight client nodes and configuration C1. However, this discrepancy can be explained by the fact that experiments were conducted at different times and other applications could have contributed to the additional communication and processing burden on the PlanetLab nodes. The observed behavior is supported by the design of RAMBO, where read and write operations are "treated equally": (a) write and read requests have equal prior- ity, (b) both operations require two communication rounds. Hence there is no essential difference if two read (or write) requests overlap as opposed to having a read request overlap with a write request.

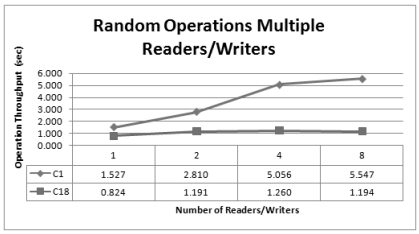

Fig. 3. Full concurrency scenario

Finally, experimental data supports our intuition about system throughput when different configuration sizes are used. However, this is not the case for the conditional theoretical analysis of [17] as explained in the Introduction; it was shown that each read/write operation is expected to complete by 8δ (δ being the maximum message delay) regardless of the configuration size and number of readers – and under the assumption of a steady state. Since system throughput is computed using average operation latencies, the above assumption would imply that system throughput should increase with the number of readers and writers regardless of the configuration size. We don't want to minimize the importance of results in [17], but rather emphasize that experimental results on global scale encompass many variables that are difficult to consider in theoretical analysis.

4.4 Scenario 4: Multiple Objects

This experiment tests whether the number of data objects (memory cells) constituting the memory system implemented by RAMBO influence its performance. The operation requests, per node, are invoked to random objects in the memory system. In addition, a node decides to perform a read or a write operation based on a random "coin flip." As in previous scenarios, configurations C1 and C18 are utilized. The number of objects in the memory was increased from two to eight. Fig. 4 depicts the collected average system throughput for each set of experiments.

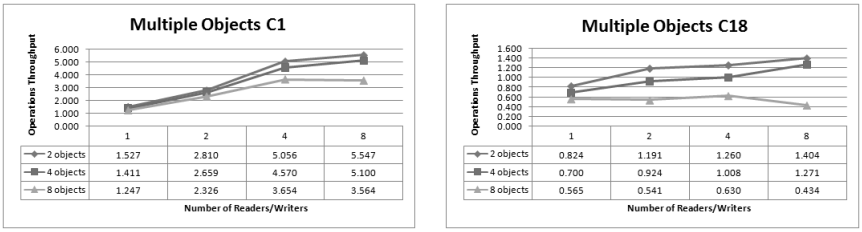

Fig. 4. Increasing number of memory cell and readers/writers scenario, C1 (left) and C18 (right)

In this experiment we expect the system throughput to degrade with the number of objects in the domain. Reason being that maintenance of each object introduces processing overhead. This expectation is supported by the experimental data. The overhead cost is made evident in the experiment involving C1, where with the number of objects system throughput decreased. However, the throughput variations in the experiment involving C18 is surprising, as one would expect lesser throughput gap between domain of size 2 and 8. One possible explanation for this behavior is the influence of periodic gossip on communication latency. Each gossip message exchanged between replicas includes information about the entire domain and with bigger messages operation latency may be negatively impacted.

4.5 Scenario 5: Reconfiguration

Reconfiguration is different from read and write operations in respect that any number of concurrent read and write operations can be performed at any given time where there these are independent. Reconfiguration requires consensus and configurations are installed in a specific sequence. Therefore, reconfiguration results are presented in terms of frequency rather than throughput. Recon- figuration duration is computed at the node that proposes the next configuration and it is defined as an interval from the proposal of the new configuration until the correspond- ing reconfiguration-request acknowledgment is received.

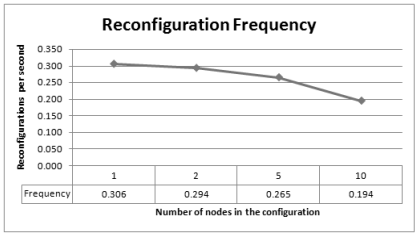

Fig. 5. Reconfiguration scenario

The experiment was conducted with participation of 20 nodes that periodically gossiped in the background. Each proposed configuration has the same size as the existing one, and consists of a disjoint set of nodes. For example, in the case of configuration of size 10, a node not belonging to the existing configuration proposes a new configuration consisting of the other 10 nodes – participating in the system (including itself). Fig. 5 depicts the average reconfiguration frequency for different configuration sizes.

Phases involved in reconfiguration require communication with some majority of configuration being replaced and the configuration being installed. Therefore, we expect the reconfiguration throughput to decrease as the size of configuration increases.

The experimental data demonstrate that the reconfiguration frequency is indeed influenced by the size of involved configurations. As explained in Section 3, the reconfiguration protocol requires three phases during which majority quorums are contacted. As the size of configurations is increased so is the size of the contained within quorums. Since progress of each reconfiguration phase depends on members of the current and the future configuration being updated (via message exchange), the resulting communication burden causes an increase in the reconfiguration duration for the larger sized configurations.

As it is the case with read and write operations, the increase in the observed reconfiguration frequency for the larger size configurations was expected. Again, this is not the case for the conditional theoretical analysis of reconfiguration latency in [17].

4.6 Scenario 6: Joining

This scenario seeks to determine whether the increasing configuration size affects the average join time of the system. The join throughput is the rate at which nodes join the RAMBO system, where the duration of the join request is measured from the time a join request is sent to the set of seeds until the join-acknowledgment is received. Furthermore, join time was assessed with the participation of twenty nodes. Initially, 10 nodes joined the system, and the remaining 10 joined RAMBO one at a time. Fig. 6 depicts the join throughput while in- creasing the size of configuration (each plot point is averaged over 5 runs).

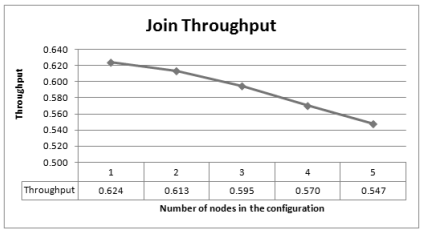

Fig. 6. Join scenario

As in the previous scenarios the through put of join operations is expected to decrease with the size of configuration. This is consistent with the collected experimental data. According to [6], during the join request, a node attempting to join the service submits join requests to the local Reader-Writer component and awaits an acknowledgment from some active RAMBO node. Thus, the augment in the size of configuration increases the number of communication complexity attributed to the periodic gossip, and thereby increases communication latency of the system and hence the join throughput decreases.

5 Conclusions

Our experiments with RAMBO implementation demonstrate its behavior in a realistic dynamic environment. The implementation deals well with the somewhat extreme conditions of PlanetLab; this is unsurprising as it was designed to cope with dynamic behaviors, delays, and failures. However, our experiments demonstrate its sensitivity to communication delays. This observation in itself is intuitive, but was not made evident by the prior theoretical analyses. In summary, our results demonstrate that read and write operation throughput is unaffected by concurrency, throughput decreases with the number of participants and when large size configurations are used, RAMBO is sensitive to the load demands where its performance scales well with these demands (for the tested scenarios), and estimation of average network delays is necessary in order to best throttle periodic gossip. Deploying RAMBO implementation on PlanetLab provided us with a better understanding of how RAMBO will behave in deployments outside of the controlled lab environment. Overall, this case-study demonstrates the feasibility of developing efficient and dependable reconfigurable sharable data services *with provable consistency guarantees* on unreliable distributed systems.

References

1. Special issue on group communication services. Communications of the ACM 39(4) (1996)
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. Journal of the ACM (JACM) 42(1), 124–142 (1995)
3. Awerbuch, B., Vitanyi, P.: Atomic shared register access by asynchronous hardware. In: Proc. of 27th IEEE Symposium on Foundations of Computer Science, pp. 233–243 (1986)
4. Bavier, A., Muir, S., Peterson, L., Spalink, T., Wawrzoniak, M., Bowman, M., Chun, B., Roscoe, T., Culler, D.: Operating system support for planetary-scale network services. In: Symposium on Networked Systems Design and Implementation, San Francisco, CA (2004)
5. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: Proc. of the 11th ACM Symposium on Operating Systems Principles (December 1987)
6. Chockler, G., Gilbert, S., Gramoli, V., Musial, P., Shvartsman, A.: Reconfigurable distributed storage for dynamic networks. Journal of Parallel and Distributed Computing 69(1), 100–116 (2009)
7. Englert, B., Shvartsman, A.: Graceful quorum reconfiguration in a robust emulation of shared memory. In: Proc. of International Conference on Distributed Computer Systems, pp. 454–463 (2000)
8. Georgiou, C., Musial, P., Shvartsman, A.: Long-Lived RAMBO: Trading Knowledge for Communication. Theoretical Computer Science 383(1), 59–85 (2007)
9. Georgiou, C., Musial, P., Shvartsman, A.: Developing a Consistent Domain-Oriented Distributed Object Service. IEEE Transactions of Parallel and Distributed Systems (2009)
10. Gifford, D.: Weighted voting for replicated data. In: Proc. of 7th ACM Symp. on Oper. Sys. Princ., pp. 150–162 (1979)
11. Gilbert, S., Lynch, N., Shvartsman, A.: RAMBO II: rapidly reconfigurable atomic memory for dynamic networks. In: Proceedings of International Conference on Dependable Systems and Networks, 2003, pp. 259–268 (2003)

12. Gramoli, V., Musiał, P., Shvartsman, A.: Operation liveness in a dynamic distributed atomic data service with efficient gossip management. In: Proc. 18th International Conference on Parallel and Distributed Computing Systems (August 2005)
13. Khazan, R., Yuditskaya, S.: A wide area network simulation of single-round group membership algorithms. In: Proc. 4th IEEE International Symposium on Network Computing and Applications, July 2005, pp. 149–158 (2005)
14. Lamport, L.: The Part-Time Parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
15. Lampson, B.: The ABCD's of Paxos. In: Proceedings of the 20'th annual ACM Symposium on Principles of Distributed Computing, p. 13. ACM Press, New York (2001)
16. Lynch, N., Shvartsman, A.: Robust emulation of shared memory using dynamic quorumacknowledged broadcasts. In: Symposium on Fault-Tolerant Computing, pp. 272–281 (1997)
17. Lynch, N., Shvartsman, A.: RAMBO: A reconfigurable atomic memory service for dynamic networks. In: Proceedings of the 16th International Symposium, pp. 173–190 (2002)
18. Musial, P.: From High Level Specification to Executable Code: Specification, Refinement, and Implementation of a Survivable and Consistent Data Service for Dynamic Networks. Ph.D thesis, University of Connecticut (2007)
19. Musial, P., Shvartsman, A.: Implementing a reconfigurable atomic memory service for dynamic networks. In: Proceedings of 18'th International Parallel and Distributed Symposium—FTPDS WS (2004)
20. Peterson, L.L., Bavier, A.C., Fiuczynski, M.E., Muir, S.: Experiences building planetlab. In: OSDI, pp. 351–366 (2006)
21. Thomas, R.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. on Database Sys. 4(2), 180–209 (1979)
22. Upfal, E., Wigderson, A.: How to share memory in a distributed system. Journal of the ACM (JACM) 34(1), 116–127 (1987)