

UNIVERSITY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

**EVALUATION OF ALGORITHMS IMPLEMENTING MULTIPLE
WRITER MULTIPLE READER ATOMIC REGISTERS ON
PLANET-LAB**

Andreas Savva

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of
Master of Science
At the
University of Cyprus

Recommended for Acceptance
By the Department of Computer Science

August, 2010

ABSTRACT

A lot of research has been conducted for studying efficient data survivability in distributed storage systems. A challenging question that researches attempt to address is “*How can a distributed system efficiently maintain data consistency among the data replicas despite system asynchrony and failures?*” Recent work introduced algorithm SFW where for the first time in the Multiple Writer Multiple Reader setting it allows for both read and write operations to be fast (the operation takes one communication round-trip to complete) but it does so by compromising the system robustness. A Server Side Ordering (SSO) technique and reader/writer predicates are utilized by algorithm SFW to allow fast operations.

The goal of this thesis is to evaluate the efficiency and practicality of algorithm SFW in a realistic network environment. For this purpose, a heuristic method is used to implement the reader and writer predicates in order to efficiently search the solution space. The algorithm is implemented in C and Sockets programming and an empirical evaluation of the algorithm is performed on PlanetLab, in respect to the percentage of fast operations, CPU consumption and operation latency. The efficiency of algorithm SFW is compared to that of algorithm SIMPLE - a robust, reliable algorithm that always performs slow operations (the operation takes two communication rounds-trips to complete). It is shown that the efficiency of algorithm SFW is minor over the SIMPLE algorithm in terms of operations latency, nevertheless network resources are reduced since they are essentially traded for CPU time consumption. Furthermore, the experiments suggest that algorithm SFW is best suited in environments that exhibit large communication delay, or when the number of readers and writers is relatively small.

APPROVAL PAGE

Master of Science Thesis

EVALUATION OF ALGORITHMS IMPLEMENTING MULTIPLE WRITER MULTIPLE READER ATOMIC REGISTERS ON PLANET-LAB

Presented by

Andreas Savva

Research Supervisor

Chryssis Georgiou, Assistant Professor

Committee Member

George Pallis, Lecturer

Committee Member

Demetris Zeinalipour, Lecturer

University of Cyprus

August, 2010

Acknowledgements

I would like to thank my research supervisor, Assistant Professor Chryssis Georgiou for providing me the opportunity to work with him. He was supporting and guiding me throughout the completion of this thesis. I would also like to thank Nicolas Nicolaou for his insight and ideas during the implementation. Finally, I am grateful to my partner and colleague Ioanna Savva for her support and help during the writing of this thesis.

TABLE OF CONTENTS

ABSTRACT.....	i
Introduction	1
1.1 Motivation and Related Work	1
1.2 Contribution	4
1.3 Chapter breakdown	5
Background	6
2.1 Atomic Read/Write Object	6
2.2 Quorum Systems	8
2.3 Prior Work.....	10
2.4 PlanetLab	16
MWMM Algorithms	17
3.1 Algorithm SIMPLE	17
3.2 Algorithm SFW	20
3.2.1 Server	23
3.2.2 Reader	24
3.2.3 Writer	25
Implementation.....	28
4.1 Design	28
4.2 Communication.....	30
4.3 Server	34
4.4 Writer	35
4.5 Reader.....	37
4.6 Code structure and Compilation	37

4.7 Read/write predicates of Algorithm SFW.....	39
4.7.1 Reader Predicate.....	39
4.7.2 Writer Predicate	41
4.8 Correctness and Robustness	43
Experimentation Setup	46
5.1 Methodology and configuration	46
5.2 Executing on PlanetLab.....	48
5.3 Problems and Limitations	52
Empirical Evaluation	57
6.1 Experiments and Scenarios.....	57
6.1.1 Experiment 1: Number of readers and writers effect	58
<i>Scenario 1: Number of Writers</i>	58
<i>Scenario 2: Number of Readers</i>	58
<i>Scenario 3: Operation Interval</i>	58
6.1.2 Experiment 2: Quorum Intersection Degree	59
<i>Scenario: Effect of Quorum Intersection Degree</i>	59
6.1.3 Experiment 3: Comparison of the SFW with the SIMPLE algorithm.....	59
<i>Scenario: Increasing readers and writers</i>	60
6.2 Results.....	60
6.2.1 Experiment 1.....	60
6.2.2 Experiment 2.....	70
6.2.3 Experiment 3.....	73
6.3 Conclusions	79
Epilogue	81

Bibliography	84
Appendix A	88
Appendix B	91
Appendix C	95
Appendix D	98
Appendix E.....	100
Appendix F.....	102
Appendix G	105

LIST OF TABLES

Table 1: the SFW reader predicate, where $ B $ is rounded down to the nearest integer [12].	25
Table 2: The writer predicate for the SFW algorithm, where $ B $ is rounded down to the nearest integer [12].	27
Table 3: Message header fields description	31
Table 4: Messages format used in communication between clients and servers	32
Table 5: <i>communicate</i> function details	33
Table 6: Message creation functions used in client and server implementation	33
Table 7: Main server functions	35
Table 8: Writer process main functions	36
Table 9: Pseudo code for the reader predicate	42
Table 10: Test cases	44
Table 11: PlanetLab nodes used for server processes	48
Table 12: PlanetLab machines minimum hardware specifications [28].....	48
Table 13: example of pssh command	50
Table 14: Example of steps for starting a scenario	53
Table 15: Example of steps to download results.....	54
Table 16: PlanetLab kill auto send message example	55
Table 17: example of configuration file: confic.ini.....	90
Table 18: Client PlanetLab Nodes.....	102

LIST OF FIGURES

Figure 1: Atomicity Example [17].....	7
Figure 2: Examples of quorum system types. Circles represent servers and the black line surrounds the servers that belong to a quorum.	9
Figure 3: the writer processes communication round in the SWMR model [17].....	12
Figure 4: Atomicity violation in the absence of reader second communication round [17]	Error! Bookmark not defined.
Figure 5: Writer protocol [17].....	18
Figure 6: Reader Protocol, the <i>maxTag</i> denotes the maximum tag that a reader process receives from the quorum. [17]	19
Figure 7: uniqueness of the SIMPLE tags [17].....	19
Figure 8: non-uniqueness of the SFW algorithm tags [17]	22
Figure 9: Writer communication messages and logic [17].....	26
Figure 10: write operation of the SIMPLE algorithm. [17].....	26
Figure 11: pseudo code for the reader predicate [12]	41
Figure 12: writer predicate implementation pseudo code [12].....	43
Figure 13: PIMan slice login, it request AuthString and PrivateKeyPassword	49
Figure 14: PIMan host selection screen on the right and <i>Overview</i> of connected hosts to the left.	50
Figure 15: Percentage of fast write operations with 80 readers	61
Figure 16: Percentage of fast read operations with 80 readers	62
Figure 17: client performance: average operation latency	63
Figure 18: percentage of client timeout failures	63
Figure 19: Percentage of fast write operations as the readers increase	65
Figure 20: percentage of fast read operations as the readers increase	66
Figure 21: The average operation latency while reader processes increase and writers are fixed	66

Figure 22: percentage of failures from timeouts while reader processes increase and writers are fixed.....	67
Figure 23: Effect on % of fast read operations, on the vertical axis is the percentage and on the horizontal axis there are two categories, (1)80 readers and 10 writers, (2) 10 readers and 80 writers.....	68
Figure 24: Effect on % of fast write operations, on the vertical axis is the percentage and on the horizontal axis there are two categories, (1) 80 readers and 10 writers, (2) 10 readers and 80 writers.....	69
Figure 25: The percentage of fast write operations with 40 readers and 40 writers, while n increases.	71
Figure 26: The percentage of fast read operations with 40 readers and 40 writers, while n increases	72
Figure 27: 40 reader and 40 writer time of execution in respect to n	72
Figure 28: Percentage of fail operations due to timeouts in communication in respect to n	73
Figure 29: Read operations CPU time comparison of the SIMPLE with the SFW algorithm.....	75
Figure 30: Write operations CPU time comparison of the SIMPLE with the SFW algorithm.....	75
Figure 31: Read operation latency comparison of the SIMPLE with the SFW algorithm	76
Figure 32: Read operations fail% and fast% for the SFW algorithm	76
Figure 33: Write operations operation latency comparison of the SIMPLE with the SFW algorithm.....	77
Figure 34: Write operations %fail and %fast for the SFW algorithm	78

Chapter 1

Introduction

1.1 Motivation and Related Work

A distributed system is a collection of autonomous processes which interact by sending and receiving messages, but appears to its users as one compact logical system. Sharing data in distributed systems is not merely natural system functionality but a core requirement by its users. Processes can share data in a reliable way since data are replicated over multiple locations on inexpensive basic storage units (e.g., hard disks, servers, tapes).

Survivability of data is crucial in systems and applications. Distributed systems offer distributed storage of data on geographically diverse locations providing more robustness and fault-tolerance than single box servers. On the other hand how can a distributed system efficiently maintain data consistency among the data replicas despite system asynchrony and failures? System component failures of hardware such as hard disks, network links, routers and software are frequent. It is a great challenge for a distributed storage system to be able to continue sharing data in an unpredictable environment.

A common approach to ensure data survivability on single box server machines is data replication using redundant array of inexpensive disks (RAID) [1]. Consider that server machines can also fail if any of its hardware fails (e.g., network interface) and hence the services it provides will not be available to clients. Single box servers are single point of failures. RAID may avoid data loss from common disk failures but it still resides on a physical location exposed to natural disasters.

Distributed storage systems may overcome the problems of single box server systems by exploiting redundancy. Still, each of the servers in the system is exposed to the same failures as a single box system but not to catastrophic site failures. The more servers the distributed storage system has the more robust, fault-tolerant and reliable it is but with added cost.

Researchers have been addressing the survivability issue by constructing efficient read and write operations to access atomic registers. Atomic registers represent replicated data objects on distributed nodes. Any data object is perceived by the system and its users as a single data object with sequential access (linearizability) to it, regardless of the multiple replicas of the object existing in the system. Have in mind that atomic registers [2] [3] [4] are different from atomic operations commonly found in concurrency control of database systems, transaction processing (serializability properties [5] [6]).

The efficiency of read and write operations is measured as the number of communication rounds between the system processes, which are classified as reader, writer and server processes. A communication round starts when a client sends an operation request to all the servers and ends when the client has received “enough” of the server responses.

Faster distributed algorithms that efficiently maintain data consistency among the data replicas despite system asynchrony and failures have a broad range of applications. Pioneers in the message-passing model in [7] implemented an atomic Single Writer Multiple Reader (SWMR) register in which write operations need one (1) communication round (fast) and read operations need two (2) communication rounds (slow) to complete. In the SWMR model any client process may fail, while only a minority of servers may fail.

Continuing on the work of [7], the authors of [8] [9] presented a Multiple Writer Multiple Reader (MWMR) register in which read and write operations are slow and generalized majorities to quorums. The servers are organized into a quorum system: a collection of server sets (quorum) in which every two intersect with each other. A variation of the algorithm of [8] is referred to as algorithm SIMPLE in the context of this thesis.

Further research [10] concluded that fast read and write operations are possible in the SWMR but set a bound on the number of reader processes in the system. A bound that was later removed in [11], allowing an unbound number of readers but with the overhead of one (1) slow read operation per write operation. The register implementations in which fast and slow operations coexist are called semi-fast.

Fast or semi-fast operations were shown as not possible in the MWMR model [11]. In [12] two new implementations (algorithms CwFr and SFW) show that under certain constraints both read and write operations can be fast while atomicity is preserved in the presence of asynchrony and crashes. The write operations in algorithm CwFr need two communication rounds but it optimizes on read operations by taking advantage of quorum views. Quorum views [13] are a tool used to analyze the tag participation in the quorum. A tag is a tuple that essentially consists of a timestamp, a process identifier and a value. The SFW algorithm exploits a new technique called Server Side Ordering (SSO) which allows for fast read and write operations in certain cases.

1.2 Contribution

The goal of this thesis is to evaluate the efficiency and practicality of algorithms SIMPLE and SFW.

Algorithm SFW uses predicates at the client side to decide if operations need to proceed to a second communication round or not. These predicates search a huge solution space in order to decide. A heuristic method is used to implement the predicate. Our experiments demonstrate that the solution space the method searches can contain a valid answer for the predicate. However, it is possible that if the heuristic method does not find any answer it is not necessarily the case that a valid answer does not exist. The precise accuracy of the heuristic method is beyond the scope of this thesis and is left for future work.

An empirical evaluation of algorithms SFW and SIMPLE is conducted on PlanetLab [14], which is a global network for testing distributed services. The algorithms' performance is compared against their average operation latency (the total time it takes for an operation to complete) and the percentage of fast operations (for algorithm SFW).

The SFW algorithm promise of fast operations requires a high intersection degree on the underlying quorum system which might compromise the SFW robustness. So, one wonders how would the algorithm actually perform in a realistic distributed setting where crashes, failures and asynchrony are inherent? PlanetLab provides such arbitrary network conditions [15], and hence it is suitable to assess the practicality of algorithm SFW.

The first of the experiments performed for the empirical evaluation of the efficiency of algorithm SFW, examines the effect of the number of writers and readers in the system and their operation intervals. The results from this experiment are then

used to subsequent experiments, where the effect of the quorum system intersection degree is investigated. Finally a comparison of the average operation latency of algorithms SFW and SIMPLE is given.

The empirical evaluation of algorithm SFW shows that its implementation is practically feasible on unreliable distributed systems (such as PlanetLab) and its performance is reasonable (in the scenarios run) under the extreme conditions of PlanetLab.

1.3 Chapter breakdown

In the next chapter, atomic registers and quorum systems are discussed and an overview of related work is given. In Chapter 3, algorithms SIMPLE and SFW are described and in Chapter 4 the implementations of the algorithms are explained. In Chapter 5, the configuration and setup of running experiments on PlanetLab is presented and in Chapter 6 the results of the empirical evaluation of the algorithms are illustrated. Finally, in Chapter 7 conclusions and possible future work is presented on the subject.

Chapter 2

Background

In this chapter the notion of atomicity and quorum systems are defined. Related work is summarized and a description of PlanetLab is provided.

2.1 Atomic Read/Write Object

An atomic register is an abstract data structure that is defined by a set of possible values and a set of primitive operations, such as read and write. A process performs one operation at a time by sending a request to all the servers holding a replica of the register. To perform a read or write operation on the atomic register two steps are necessary. The *invocation* step includes either a read or a write request. Similarly, the corresponding *response* step includes either a read or a write acknowledgement [13]. The operation is considered *complete* if both steps are performed [4].

An operation α *precedes* an operation β if α completes before β 's invocation. Any operations α and β are considered *concurrent* if and only if α does not precede β and β does not precede α . In other words, two operations are *concurrent* if neither of them precedes the other [16]. If two operations are complete, not concurrent and are invoked by two distinct processes then they are called *consecutive* [12].

A register guarantees that once a processor reads a particular value, then, unless the value of this register is changed by a write, every future read of this

register is always available, regardless of processors slow-down or failures. Atomic registers guarantee the *atomicity* (linearizability) [4] of operations by satisfying the following properties:

1. A read operation δ returns
 - a. the value written by the most recent preceding write, or
 - b. a value written by a write operation that is concurrent with δ
2. If a read operation δ_1 reads a value from a write operation γ_1 and a read operations δ_2 reads a value from a write operation γ_2 and δ_1 precedes δ_2 , then γ_2 does not precedes γ_1 .
3. All write operations are totally ordered.

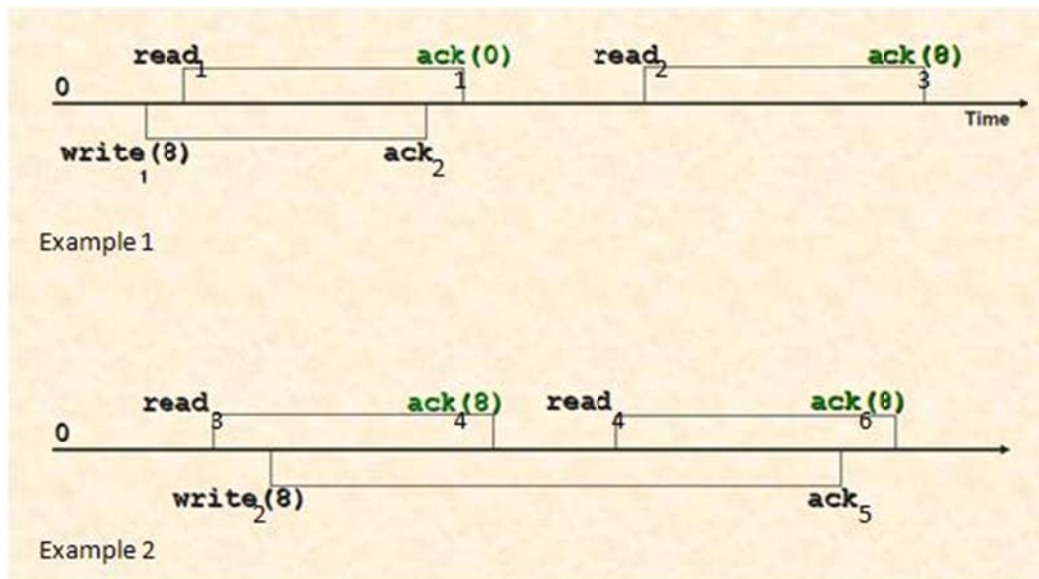


Figure 1: Atomicity Example [17]

Consider the examples in Figure 1 where initially the atomic register value is zero (0). The first example shows a write_1 operation that writes the value 8 to the atomic register. A read_1 operation is invoked after the write_1 operation but before the acknowledgement for the write_1 operation is received. Since the two operations are concurrent it is possible that the value returned by the read_1 operation will not be the

value written by the write operation despite the fact that it was invoked after it. On the other hand $read_2$ must read the value 8 since it is invoked after the $write_1$ completes. The second example shows a $write_2$ operation that writes the value 8 and is concurrent with $read_3$ and $read_4$. It is possible for $read_3$ to read the value 8 or the previous value (0) but $read_4$ must read the same value as $read_3$ since its invocation starts after $read_3$ completes, otherwise atomicity is violated.

Finally, the atomic register must be *wait-free* [2], which guarantees that if a non-faulty process invokes an operation then the operation completes in a finite number of steps, regardless of the status (execution speeds or failures) of the other processes.

2.2 Quorum Systems

A quorum is a group from a set of distributed nodes, typically servers [18]. A quorum system is a collection of quorums, in which any two quorums intersect with each other. Since any two quorums intersect, the quorum system is characterized as a pairwise (2-wise) quorum system. Formally a quorum system \mathbb{Q} is defined as, $\mathbb{Q} = \{ Q : Q \subseteq S \} s. t. \forall Q_i, Q_j \in \mathbb{Q} : Q_i \cap Q_j \neq \emptyset$, where $S = \{s_1, s_2 \dots s_n\}$ ($n \geq 1$) is the set of servers.

There are different types of quorum systems [19], some examples can be seen in Figure 2: Examples Of Quorum System Types:

- (a) A matrix type where servers form a grid and a combination of a row with a column defines a quorum. All quorums have the same size of $2\sqrt{|S|} - 1$, where S is the set of servers.
- (b) A majority type quorum system where each quorum size must be at least $\lceil (|S| + 1)/2 \rceil$, where S is the set of servers.

(c) A crumbling wall quorum system type where a quorum is defined by any one complete row and one server from each row below it.

Quorum systems can also be classified as static and dynamic [8]. A quorum system is static when it is pre-computed and it does not change after the algorithm execution starts. A dynamic quorum system may reconfigure its initial quorum deployment, which means that process (server) participation in quorums can change dynamically during execution.

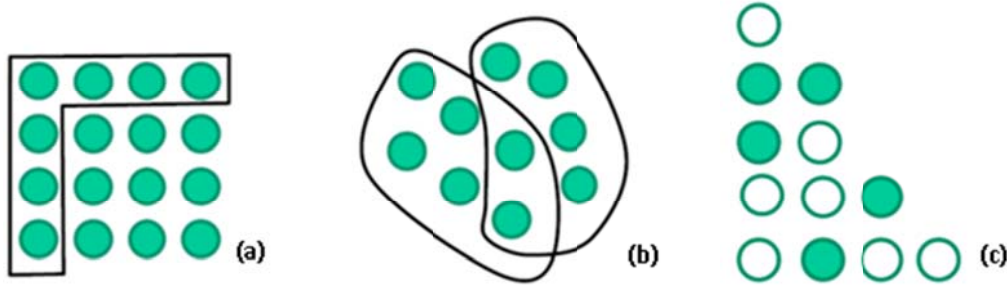


Figure 2: Examples of quorum system types. Circles represent servers and the black line surrounds the servers that belong to a quorum.

Quorum systems specializations are the n -wise quorum systems, in which any quorum intersects the minimum with n other quorums. A quorum system \mathbb{Q} is called an n -wise quorum system [12] if for any $A \subseteq \mathbb{Q}$, s. t. $|A| = n$ we have $I_A \neq \emptyset$ holds. We call n the intersection degree of \mathbb{Q} , and for a set of quorums $A \subseteq \mathbb{Q}$, the intersection of the quorums in A is denoted by $I_A = \bigcap_{Q \in A} Q$.

Quorums are widely used for ensuring consistency in atomic register implementations. In particular the servers holding the register replica are partitioned into intersecting quorums. Hence it is not necessary for clients accessing the register to communicate with all the nodes but only with the nodes that belong to some quorum Q . Only the nodes of Q receive the load related with the data object in question. Thus nodes outside Q are more relaxed and it results to higher availability

of the service overall. Also the basic technique to ensure consistency of the data in distributed storage systems is to notify some quorum Q of the update made. When a client accessing the data contacts some quorum Q' , it is ensured that it learns about the earlier update since quorums intersect.

It is not obvious how to efficiently deploy a theoretically good quorum system in a real network system. By first designing the quorum system, and then determining a good deployment, it seems possible to obtain both good network performance as well as good quorum system properties.

The quorum deployment problem is studied in [20] as a new combinatorial optimization problem. There are two parts to solving this problem: mapping a quorum system to real nodes and mapping from nodes to quorums. The general quorum deployment problem is defined as: given a quorum Q , and a distributed network C , the goal is to determine a deployment that has optimal cost. It is shown [20] that the general deployment problem cannot be approximated and that majorities is the most simple deployable quorum system in all networks. The quorum system deployment used in this thesis is presented in Chapter 4.

2.3 Prior Work

In the message-passing model, the processes communicate via messages sent through communication links. Each process has a unique identifier and is located at one node of the network and can only send messages to processes located in directly neighboring nodes. We consider three sets of processes: R readers, W writers and S servers communicating through reliable TCP channels in the asynchronous message passing model.

The network setup considered is unpredictable; *processes may crash* and there is *asynchrony*. A process may stop executing at any point of the computation with no prior notification and slow processes cannot be differentiated from crashed ones. Any of the clients may crash or get disconnected; for S servers where T servers may fail by crashing, up to half of the servers may crash ($T < \frac{S}{2}$) when considering majorities. In the case of quorums at least one quorum must not crash. Asynchrony means that there are no guarantees in message delays and relative process speeds (some process may be slower than others).

The *Single-Writer, Multiple-Reader* (SWMR) register implementation is presented by [7] in the message-passing model. The clients include only a single process for write operations and multiple processes for read operations while all the servers hold a register replica. Clients do not communicate between them and neither do servers. Clients only communicate with servers through communication rounds.

A process p performs a *communication round* for an operation π if:

1. p sends a message m regarding π to a subset of processes
2. Any process that receives m , replies to p
3. Process p collects “enough” of such replies and proceeds accordingly.

A process collects “enough” replies when a quorum of servers reply. The SWMR model in [7] considers “enough” server replies when a majority of them reply.

A tag-value pair is introduced to impose an order on the read operations. The tag consists of a label which basically is a positive number of type integer used as a timestamp. To support asynchrony the timestamp has nothing to do with real time and logical clocks, it is just a number that is incremented only by the writer process each time it performs a write operation. Essentially this label is used to define the order of write operations and which write value is read by the read operations.

It takes one communication round to complete a write operation in which the writer increments the timestamp and sends the tag to a quorum. The writer communication round in the SWMR model is shown in Figure 3, where the writer process increments the timestamp in its tag and sends a message to all the servers, then it waits until the majority of the servers reply to complete the write operation.

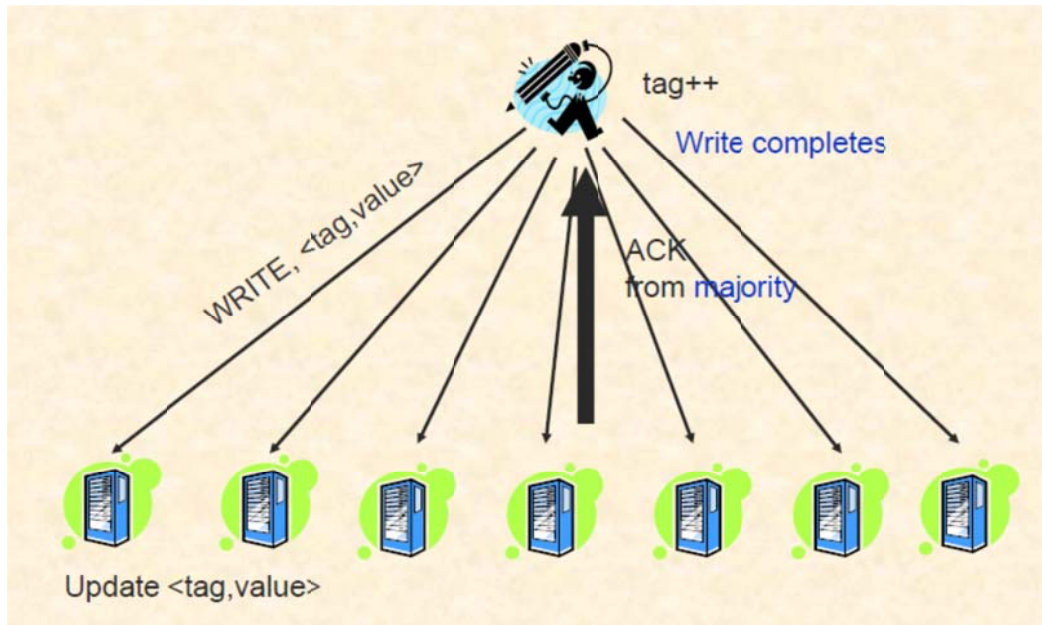


Figure 3: the writer processes communication round in the SWMR model [17]

The readers receive the latest tag from a majority of servers during the first communication round and they need a second communication round to actually write (on the register located on the servers) the value obtained in the first round.

An example of how atomicity can be violated if the reader does not perform a second communication round is shown in **Error! Reference source not found..** Initially the tag of the register is $\langle 1, V_0 \rangle$. A slow writer attempts a write operation on the register with tag $\langle 2, V_1 \rangle$. While the write operation is in progress, a reader reads the value of the register from a majority of servers that already has the tag $\langle 2, V_1 \rangle$. Since it does not do a second communication round to write this value it completes its operation having read the tag $\langle 2, V_1 \rangle$. A second reader reads the value of the

register from a majority of servers that the new tag has yet to be written, while the write operation is still in progress. The read operation is completed having read the tag $\langle 1, V_0 \rangle$. This violates the second atomicity property mentioned in Section 2.1.

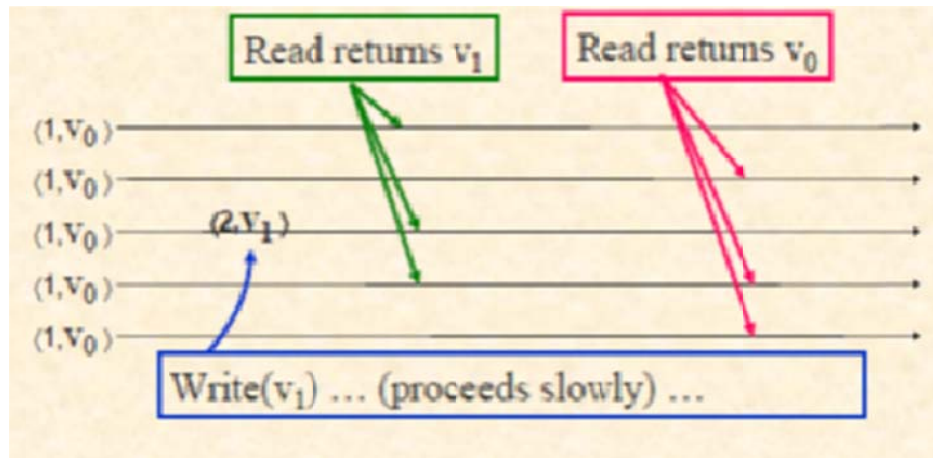


Figure 4: Atomicity violation in the absence of reader second communication round [17]

The *Multiple-Writer Multiple-Reader (MWMR)* register implementation is introduced in [8] [9]. It is similar to the SWMR with the exception that there are multiple writer processes as well. The authors implemented atomic registers using a quorum system (specifically a generalization of majority sets). Their implementation uses a dynamic quorum system where reconfigurations occur to accommodate changes on the server population. The *Reconfigurable Atomic Memory for Basic Objects (RAMBO)* [9], [21] implements a reconfiguration service. Even when there is no reconfiguration, read and write operations always take two communication rounds to complete and more otherwise. In the MWMR model, likewise to the SWMR model, the second communication round of the read operation is needed to actually write the value read in the first round. This led to the folklore belief that “atomic reads must write” [3] [22] [23].

The folklore belief is shown not always be true in [10], which introduced a *fast* wait-free atomic SWMR register implementation. In a *fast* implementation readers and writers perform only one communication round operations. To achieve fast

operations the authors of [10] bound the numbers of readers to be $R < S/T - 2$ (where R is the number of readers, S the number of servers and T the number of servers that may crash). Also they show that a fast implementation is impossible in the MWMR setting.

Observe that the limit on the number of readers shown in [10] for fast implementations is impractical. Later work [11] provides a non-straightforward extension of the work in [10] by implementing a *semifast* SWMR model while preserving atomicity. In a *semifast* implementation the writer operations take one communication round to complete whereas read operations take one or two communication rounds to complete. Formally the SWMR model implementation of an atomic object is semifast when the following are satisfied:

1. all write operations are fast and
2. all complete read operations can be either fast or slow iff $T < \frac{S}{2}$ and
3. If a read operation r_1 is slow, then all read operations that precede or succeed r_1 and return the same value as r_1 are fast (only a single complete read is slow per write operation).
4. There exists an execution of the implementation which contains only fast read and write operations (even if operations are concurrent).

The notion of *Virtual Node* is introduced, a group of reader processes that all share the same *Virtual Identifier*. Particularly a read operation must be fast if it precedes or succeeds a complete fast read operation, when both reads return the value written by the same write operation. Concurrent read operations with a slow read operation may or may not be fast. Furthermore it is also shown that no *semifast* implementation exists for the MWMR model even for $T = 1$. Simulations presented in [11] suggest that under reasonable execution conditions only a small percentage

(7.5% - 10%) of read operations are slow. In summary, fast or semi-fast operations in the MWMR model were shown as not possible.

A *Semifast Like Implementation for Quorum systems (SLIQ)* algorithm for the SWMR model is introduced in [13]. This implementation is *weak-semifast*, meaning that it enables fast reads but allows multiple slow reads per write; formally, a weak-semifast implementation is the same as a semifast implementation but without property 3. For this purpose a client-side prediction tool called *Quorum Views* is introduced. The *Quorum Views* are used to supply adequate data involving the distribution of the latest tag in the quorum being accessed. Read operations use the *Quorum Views* to make educated decisions locally whether a second round is needed. The *SLIQ* algorithm was simulated using the NS-2 network simulator [24]. The results showed that only about 13% of the read operations proceed to a second communication round, in common cases.

Constraints on the efficiency of the MWMR model are analyzed in [12] and two new algorithms are introduced. These algorithms support some fast operations while atomicity is preserved in the presence of asynchrony and crashes (Recall that it is impossible to have all operations to be fast [10] [11]).

The first is algorithm *CwFr*, which optimizes on read operations by taking advantage of *Quorum Views*. The write operations still need two communication rounds to complete. The second algorithm is the SFW algorithm, which exploits a new technique called Server Side Ordering (SSO). The SSO allows for both fast read and write operations in certain cases. When the intersection degree of the underlying quorum system is below 4, it is not clear which of the two algorithms performs better because all write operations of the SFW are slow as well.

This thesis focuses on algorithm SFW and explores its efficiency on PlanetLab when the intersection degree of the deployed quorum system is above 4.

2.4 PlanetLab

PlanetLab is built-up as a collaborative distributed system in which different organizations donate two or more computers adding up to a total of hundreds of nodes. Together these computers form a distributed overlay network for deployment and assessment of distributed planetary-scale network services [15] [25].

As of the writing of this thesis, PlanetLab is composed of 1089 nodes at 503 sites worldwide provided by academic and industry institutions. Its resources are divided into slices where each can be viewed as a network of virtual machines. 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. 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. Thus PlanetLab is a realistic deployment setting to test and evaluate SFW algorithm and compare it with a simpler, operation slow MWMM algorithm.

Chapter 3

MWMM Algorithms

In this Chapter the SIMPLE and SFM algorithms are defined and explained in more detail.

3.1 Algorithm SIMPLE

In algorithm SIMPLE the servers are arranged in a quorum system, using the message passing paradigm for communication in the presence of asynchrony and failures. Basically SIMPLE is the algorithm defined in [8] but the servers are arranged in a static quorum system. There are three set of processes, a set of servers $S = \{s_1, s_2 \dots s_n\}$, a set of readers $R = \{r_1, r_2 \dots r_\rho\}$ and a set of writers $W = \{w_1, w_2 \dots w_\omega\}$. Any reader or writer process may crash but at least a quorum must not crash. Algorithm SIMPLE only considers crash failures and not Byzantine failures [26] [27], that is, system components are assumed to work correctly and when they fail, they do so by crashing or stopping.

The algorithm uses $\langle tag, value \rangle$ pairs to order the values written to the register. The tag is a two field tuple consisting of $\langle ts, wid \rangle \in \mathbb{N} \times W$, where ts is the timestamp and wid the writer identifier of the writer that wrote the $value$. The writers are the only processes responsible for incrementing the ts . Initially the tag is set to $\langle 0, min(W) \rangle$ for every process. The $tags$ can be compared alphanumerically. Specifically, a tag t_1 is greater than a tag t_2 ($t_1 > t_2$) if $t_1.ts > t_2.ts$ or $t_1.ts > t_2.ts \wedge t_1.wid > t_2.wid$.

The servers keep the data replicas of the register object. Clients sent messages to servers. When a server receives a request it updates its (local) tag, if the received tag is more recent than its local tag, and then responds with an ACK message.

The writer protocol is shown in Figure 5: Writer protocol. The writer process sends a WRITE request, with its current $\langle \text{tag}, \text{value} \rangle$ pair to all the servers. Then the writer process waits until it receives an acknowledgement response from a quorum. After a quorum responds, the writer discovers the maximum tag (maxTag) received from the quorum, increments it and sends it to all the servers. When a quorum of servers responds the write operation is *complete*.

The reader protocol is shown in Figure 6 and it is similar to the writer protocol with the exception that the reader does not increment the maxTag .

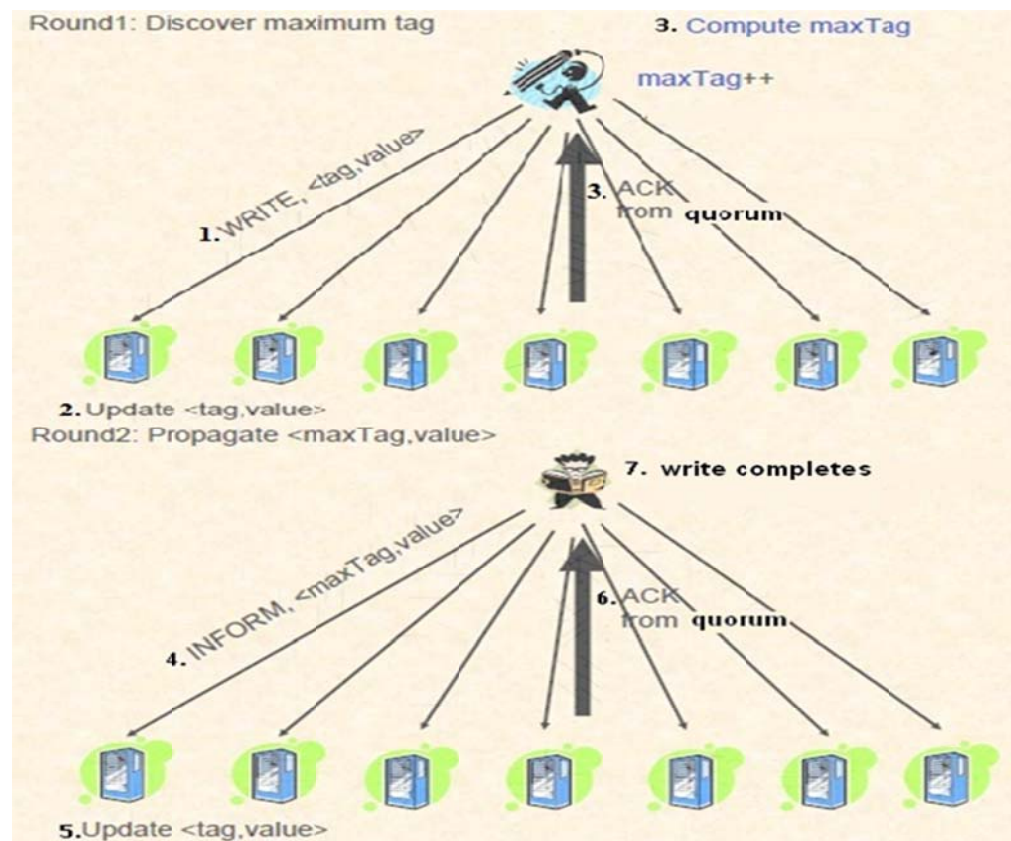


Figure 5: Writer protocol [17]

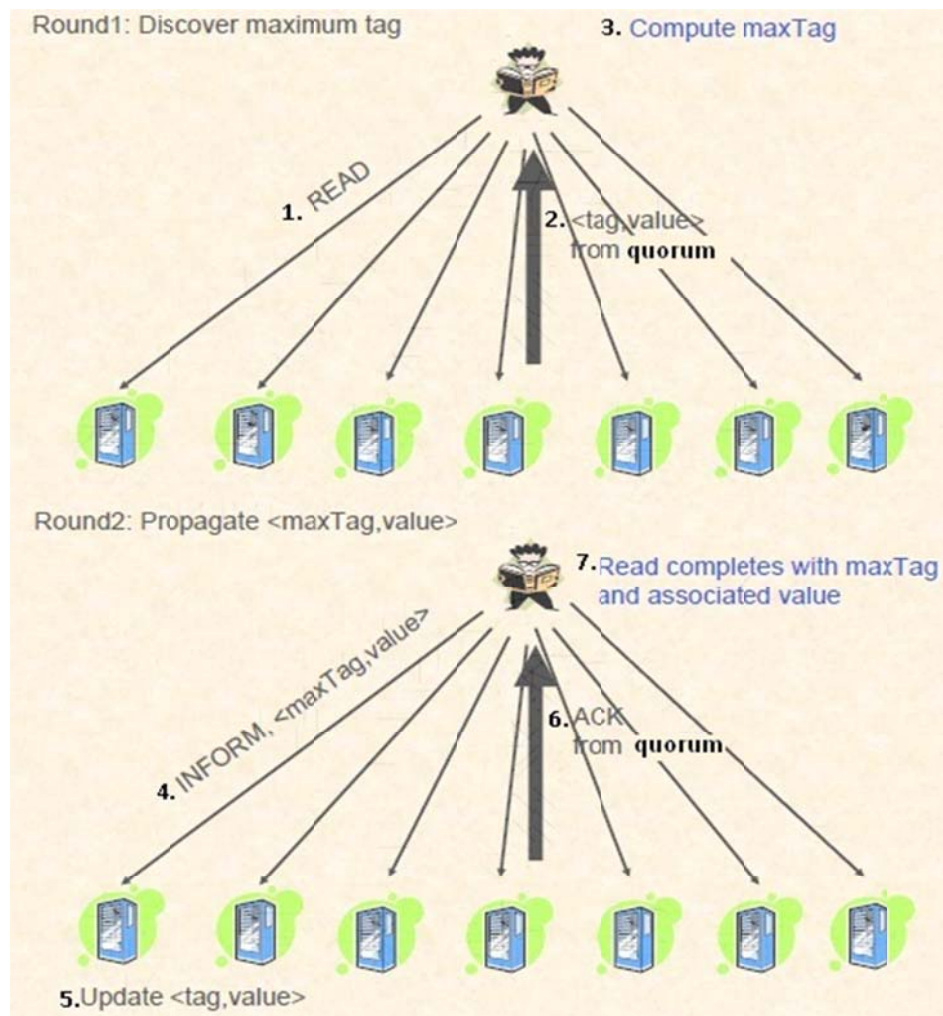


Figure 6: Reader Protocol, the *maxTag* denotes the maximum tag that a reader process receives from the quorum. [17]

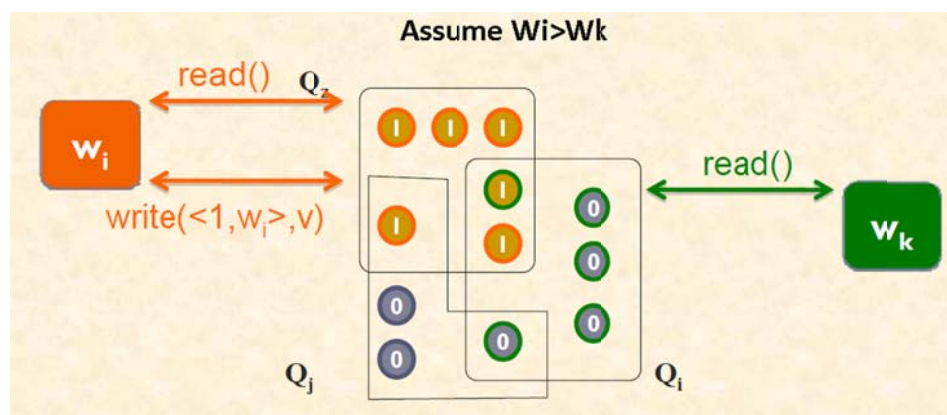


Figure 7: uniqueness of the SIMPLE tags [17]

3.2 Algorithm SFW

In algorithm SFW the servers are arranged in an n -wise quorum system, using the message passing paradigm for communication in the presence of asynchrony and failures. As with algorithm SIMPLE, there are three sets of processes and only crash failures are considered.

Algorithm SFW uses a $\langle tag, value \rangle$ to attain the required order on the values written to the register. The tag is different from the tag used by algorithm SIMPLE. The reason for this difference will be explained later in this section. Tag comparison is done alphanumerically as in the SIMPLE algorithm. The clients communicate with servers using communication rounds in which they sent their $\langle tag, value \rangle$ pair and their operation requests (READ / WRITE / PROPAGATE). The servers, when they receive a request they answer by sending their latest confirmed $\langle tag, value \rangle$ and an inprogress set which contains the ongoing write operations $\langle tag, value \rangle$ pairs.

The SFW algorithm uses a reader and a writer predicate. The predicates are used by each process to calculate the distribution of the latest tag in the responding quorum. If the reader (writer) predicate evaluates that the distribution of the tag is “good enough” such that a second communication round is not needed to ensure atomicity, then the read (write) operation completes in one communication round. The reader and the writer predicates are analyzed in Sections 3.2.2 and 3.3.3, respectively.

In algorithm SIMPLE the writers' need to proceed to second communication round to propagate the new tag-value pair of the write operation (Figure 10). Algorithm SFW is the first to introduce the possibility of one communication round (fast) write operations in the MWMM model. The predicate technique that enabled

reader operations in previous works [11] [13] to complete in one round operations, is extended [12] and applied to both the reader and the writer. In order for the writer predicate to be feasible, the responsibility of incrementing the tag timestamp has to be removed from the writers. This purpose was fulfilled by a new technique, introduced in [12], called Server Side Ordering (SSO).

The SSO technique created a new problem; generated tags by the servers may be different across servers, resulting to tag non-uniqueness. The SIMPLE algorithm does not have this problem, observe Figure 7; the tag is increment only by the writer which ensures that a quorum of server will have the same tag. To understand the problem, an example is given in Figure 8. Assume $W_i > W_k$, and Q_i, Q_j, Q_z quorums. A writer W_i communicates with Q_z to write, and W_k communicates with Q_i . Since the tag is incremented by the servers, all the servers in quorums Q_z, Q_i increment tag from 0 to 1 but due to asynchrony is possible that the intersection of $Q_z \cap Q_i$ to have its tag incremented twice resulting from 0 to 2. This leads to multiple tags for a single value and it violates atomicity.

To understand how atomicity is violated, take the following execution as an example [17]:

- W_i and W_k are two concurrent write operations that write values 3 and 4 with tags t_1 and t_2 respectively (without loss of generality let $t_1 < t_2$)
- r_1 and r_2 , succeed both write operations
 - r_1 witness t_1 for W_i and value 3, so t_2 for W_k . r_1 returns 3 since $t_1 < t_2$
 - r_2 witness t_2 for W_i and value 3, thus t_1 for W_k , r_2 returns 4 since $t_1 < t_2$

r_1 and r_2 , succeed both write operations but they do not agree on the latest written value.

For this purpose, the *tag* in the SFW is a tuple containing $\langle ts, wid, wc \rangle$, the *wc* is a writer counter, basically a number incremented by the writer at each write

operation. The wc field of the tag enables read/write processes to distinguish between write operations, since any processes can recognize that different tags have

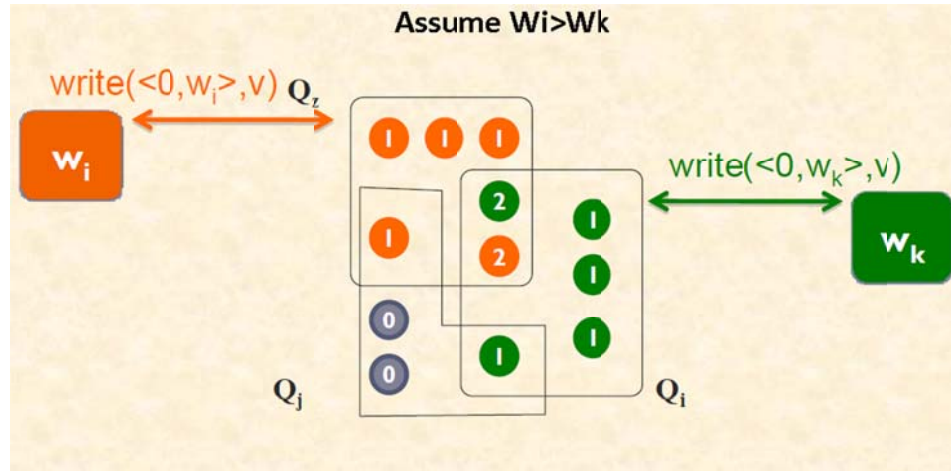


Figure 8: non-uniqueness of the SFW algorithm tags [17]

been assigned to a write operation. In [12] it is proved that algorithm SFW implements MWMR atomic read/write registers.

In [12], two operations, originating from two different processes, are regarded as *quorum shifting* operations if they are consecutive and they receive replies from two distinct quorums. If quorums, in an n -wise quorum system, do not have a common intersection then an atomic register implementation is impossible. An implementation with only fast write operations, cannot have more than $n - 1$ writer processes (n is the intersection degree of the underlying quorum system). Algorithm SFW can have up to $n/2$ fast consecutive write operations, while maintaining atomicity, thus it is nearly optimal.

We now proceed to describe the functionalities of the server, reader and writer processes in SFW.

3.2.1 Server

The server maintains the state of each register in the system and acts according to the message requests it receives. The state of the server for the register object is comprised of a *tag*, a *confirmed* tag and an *inprogress* set of tag-value pairs. The *confirmed* tag holds the latest confirmed tag seen by the server. The *inprogress* set is a set of tags that represent the ongoing write operations from each server perspective. The *inprogress* set holds a tag-value pair for the ongoing write operation of each writer process in the system.

Essentially when a server receives a request it first updates its local tag and confirmed tag, if the received tag is more recent. Additionally, if the request is for a write operation, the server increments its local timestamp and then assigns the writer id and the writer counter, of the write operation request, to its local tag. Next, the server removes any previously recorded tag-value pairs, of the writer, that reside in the server's *inprogress* set. The server generates a new t' tag-value pair where the tag is the current local tag of the server with its timestamp incremented by one (1) and the write operation value to be written. Finally the new t' tag is inserted into the *inprogress* set. A more formal definition of the server steps upon receiving a request follows:

- 1 Update (local) tag: The server adopts the request's tag if it is more recent than its local tag. Tag comparison is alphanumerical.
- 2 If it is a WRITE request from W_i then
 - 2.2 Create a new tag $t', < ts', w', wc' > = < ts, w_i, wc_i >$, assign to t' the local timestamp and the WRITE request's attributes (w_i, wc_i)

- 2.3 Remove any previous tag-value pairs from the specific writer and insert the newly generated t' tag along with the new value the writer wants to write. $Inprogress = (inprogress - \{<*, w_i, *>, val\}) \cup \{<ts', w_i, wc_i>, newVal\}$
- 3 Update confirmed tag: The server updates confirmed tag if the request's tag is more recent.

If server receives a READ request then steps 2, 2.2 and 2.3 are not executed.

3.2.2 Reader

The reader process sends a read request message to all servers containing the tag $<tag_r, value>$. When responses from a quorum Q are received, the reader creates and populates a new set, let that be called Is , with all tags from the *inprogress* sets of the responses and calculates the maximum confirmed tag ($maxConf$). So the reader has $w \times s$ (product) tags in the Is set, where w the number of writers in the system and s the number of servers in the responding quorum. The reader then compares each tag t from the Is set with $maxConf$.

If $maxConf \geq t$ then the reader adopts $maxConf$ tag along with its value. If $maxConf$ is received from an intersection between Q and $n - 1$ (where n the intersection degree of the quorum system) other quorums then the reader proceeds to a second communication round otherwise it completes (fast).

If $maxConf < t$ then the reader checks if t satisfies the reader predicate (Table 1: the SFW reader predicate). If the predicate is true for t then, the reader adopts t and its value.

Read predicate for a read ρ (PR):

$\exists \tau, B, MS, \text{ where: } \max(\tau) \in \bigcup_{s \in Q_i} \text{inprogress}_s(\rho),$

$B \subseteq \mathbb{Q},$

$0 \leq |B| \leq \frac{n}{2} - 2, \text{ and } MS = \{s: s \in Q_i \wedge \tau \in \text{inprogress}_s(\rho) \text{ s.t. either } |B| \neq$

$0 \text{ and } I_B \cap Q_i \subseteq MS \text{ or } |B| = 0 \text{ and } Q_i = MS. \}$

Table 1: the SFW reader predicate, where $|B|$ is rounded down to the nearest integer [12].

The reader proceeds to a second communication round if its predicate is true for t and t is propagated in an intersection of Q with exactly $n/2 - 2$ other quorums. In the case the predicate for t is false and I_s is empty, a second communication round is needed. While I_s is not empty the reader keeps comparing the tags in it with maxConf until it finds a tag smaller than maxConf or a tag that validates its predicate. In all other cases the reader is fast.

3.2.3 Writer

The writer process W_i sends a writer request to all the servers in the quorum system. The requests contain $\langle \text{tag}_w, \text{value} \rangle$ and the servers reply with the new tag for the write operation. The tags received from the server responses may differ. The writer needs a mechanism to select the latest tag and then judge based on the latest tag distribution in the quorum, if a second round of communication is needed to ensure atomicity. The write predicate provides this decision mechanism. Notice the differences of the writer process between the SFW and the SIMPLE algorithm in Figure 9 and Figure 10 respectively. The SFW writer does not need to use the first communication round to read the latest tag of the atomic register but rather it

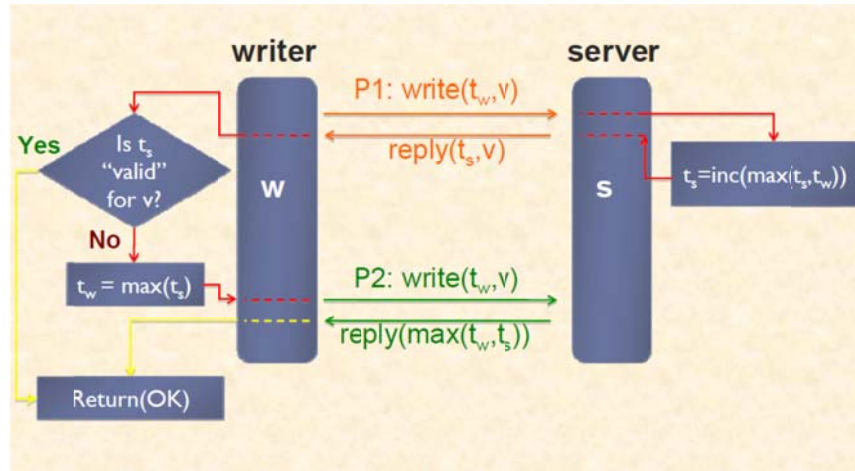


Figure 9: Writer communication messages and logic [17]

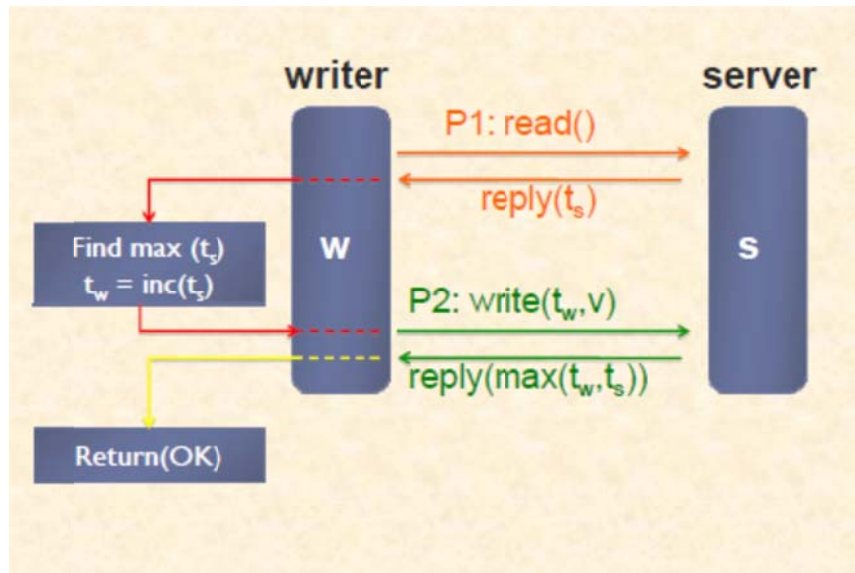


Figure 10: write operation of the SIMPLE algorithm. [17]

proceeds to directly write the new value to it. Like the writer in the SWMR model in Figure 3 but the difference is that the writer in the SFW does not increment the tag timestamp and it waits a reply from a quorum of servers.

The writer predicate (Table 2) is simpler than the reader predicate. Each server reply has an inprogress set which contains a tag for each writer in the system. The writer extracts only the tag referring to its unique identifier (*wid*), specifically every tag of the form $\langle *, wid, * \rangle$ where asterisk can be anything. Practically there is

Writer Predicate for a write ω (PW):

$\exists \tau, A, MS, \text{ where: } \tau \in \{\langle \cdot, \omega \rangle : \langle \cdot, \omega \rangle \in \text{inprogresss}(\omega) \wedge s \in Q\},$

$A \subseteq \mathbb{Q}, 0 \leq |A| \leq \frac{n}{2} - 1, \text{ and}$

$MS = \{s: s \in Q \wedge \tau \in \text{inprogresss}(\omega), s.t. \text{ either } |A| \neq 0 \text{ and } I_A \cap Q \subseteq MS \text{ or } |A| = 0 \text{ and } Q = MS. \}$

Table 2: The writer predicate for the SFW algorithm, where $|B|$ is rounded down to the nearest integer [12].

only one tag for each writer in the inprogress set. So the writer will extract $|Q|$ tags in total, which is the number of servers of the responding quorum Q , all these tags are inserted in a new set, let that be called I_s . The writer tries to find a tag t in I_s that validates the predicate (to TRUE). If a tag t , that validates the writer predicate exists then the writer adopts this tag along with its associated value.

Otherwise if such a tag t does not exist, the writer adopts the maximum tag in I_s and proceeds to a second communication round. The server may also proceed to a second communication round if the predicate is true but t is only propagated in an intersection of Q with more than $n/2 - 2$ other quorums. In any other case the write operation is fast and completes in one communication round.

Chapter 4

Implementation

In this chapter we present the design and implementation of the algorithms. Additionally, the tools created to execute the scenarios and retrieve the results are specified.

4.1 Design

An application for testing the MWMM algorithms on PlanetLab was implemented using the C programming language with Linux as Operating System. Specifically, it was compiled to be compatible with Fedora 8. The Client-Server model and TCP sockets were used for communication between servers, readers and writers. The server uses the paradigm of *serve one client with each server thread*, while clients use one thread per server for each communication round. Standard C libraries were used as the main building blocks of the implementation, with POSIX pthreads for threading.

The application consists of three major components: the server, reader and writer each of which executes as an independent process. The reader and writer processes require that the server processes are executed first and are listening to the designated ports in order to begin sending read/write operation requests. An arbitrary number of server, reader and writer processes are supported by the implemented system through a parameterized configuration file. The same is true for read and write operations.

Servers can only queue up to 256 requests on their accepting socket and can spawn up to a maximum of 200 threads for serving incoming requests. In practice the servers reach their maximum service capacity when a high number of readers and writers execute on PlanetLab, hence a preliminary experiment is executed to recognize these bounds.

The quorum system used in experiments is n -wise (where n the intersection degree of the quorum system). A static quorum system deployment is used for the servers. Server process participation in quorums is fixed and known before a scenario starts its execution and it remains the same until the end of its execution, despite the fact that network topology may dynamically change as links and network nodes fail. Also client processes assume that at least a single quorum is *correct*, that is it contains no faulty servers.

In the implementation of algorithm SIMPLE the quorum system type used is the majority. For the needs of algorithm SIMPLE we use n -wise quorum systems where n is the intersection degree of the quorum system, meaning that any n quorums of the system have a non-empty intersection. Specifically, in the implementation of SFW we define a quorum as a set of $S - T$, where S is the number of the servers and T the number of the servers that the system can afford to fail such that the total number of quorums in the system is $\frac{S!}{(S-T)! T!}$. Practically in the implementation we consider “enough” replies when the first $S - T$ responses from servers are received. It is not difficult to observe that this results to an n -wise Quorum System configuration.

Each thread spawned by the server receives the client request and examines the request's header to discover if a message body follows and needs to be received. Then the thread waits until it manages to acquire a *spin lock* on the critical section, which protects the server's state from concurrent access. A *spin lock* is a mechanism

to enforce mutual exclusion. It essentially causes the thread to wait in a loop until a variable (of type *pthread_spinlock_t*) is unlocked.

Functions that provide similar functionality to either of the processes share the same name. For example the *process* function is used by all processes to process an incoming messaging but the implementation in each case differs. This is similar to polymorphism used in object oriented languages.

An important requirement in design was that both the SIMPLE and the SFW algorithm share the same architecture and core code, such as shared libraries, communication procedures, same data structures and similar message exchange format. In the next sections a reference to the differences of the implementation of the two algorithms are given whenever it applies.

4.2 Communication

Clients communicate with Servers through TCP sockets by exchanging messages. The servers connection information such as IP or domain name are loaded from the configuration file. Each message has a dynamic size in bytes and contains a header and a body. The message header has a set of fields needed for the algorithm. For the needs of communication between clients and servers a protocol is created which is defined by the fields in the message header. Fields that can appear in the message header are summarized in table 3: message header . table 4 lists the message format used for communication between clients and servers.

The code that is responsible for generating the message header is located in the *message.c* code file. The definition of the message header creation functions is detailed in Table 6.

Message Header Field	Description
SentValue	Describes the Body value, zero(0) means body is empty and one(1) has value
Id	Process id of the sender
AlgorithmType	SIMPLE for always two round operations algorithm. SFW for operations that are predicate depended for 2 nd round of communication.
ObjectID	The unique object id of the register (atomic object).
MessageType	WRITE/ READ / INFO WRITEACK/ READACK/ INFOACK
Cnt	Request counter
Tag.ts	Tag Timestamp
Tag.wid	Tag writer id
Tag.wc	Tag write counter
ConfirmedTag(ts,wid,wc)	The same tag info for the confirmed tag
ConfirmedTag(value)	The confirmed object value, not sent for INFO message type requests
InprogressSet	An Inprogress [Tag(ts, wid, wc), Value] for each writer. Note that Value is not sent for INFO message type requests.

Table 3: Message header fields description

The configuration of the processes is setup using a config.ini file. This file contains all the global information shared among the implementation; essentially it defines the configuration of the system. The order of variable declaration in this file is

important. An example of a `config.ini` file with enough self-explanatory comments can be found in Appendix A.

SIMPLE server response message format
SentValue,Id,AlgorithmType,ObjectID,MessageType,Cnt,Tag.ts,Tag.wid,Tag.wc
SFW server response message format
SentValue,Id,AlgorithmType,ObjectID,MessageType,Cnt,Tag.ts,Tag.wid,Tag.wc,ConfirmedTag(ts,wid,wc),ConfirmedTag(value),InProgressSet
SIMPLE client request message format
SentValue,Id,AlgorithmType,ObjectID,MessageType,Tag.ts,Tag.wid,Tag.wc,Cnt
SFW client request message format
SentValue,Id,AlgorithmType,ObjectID,MessageType,Tag.ts,Tag.wid,Tag.wc,Cnt

Table 4: Messages format used in communication between clients and servers

On the client side, the *communicate* procedure (Table 5) is used as the communication primitive by which a complete communication round is performed. The *communicate* function implementation was inspired from [7], in which a function is described that handles communication. It takes as parameter the packet to send and returns an array of the acknowledging servers in `quorum_data` data structure along with its size.

Since clients need to communicate with all servers, the *communicate* procedure spawns a thread for each server to handle message interaction with the client. To manage these threads a controlling thread monitors the responses from the servers. When enough responses have been received, the controlling thread graciously notifies the threads which handle the servers that are yet to respond. The notified threads are responsible to stop any current action with their respective server and terminate.

The main Communication function used by clients	
<pre>data_t* communicate(pck_t *sMsg, data_t* quorum_data, int* quorum_size, bool_t *done);</pre>	<p>Communicate performs one communication round between client and servers.</p> <ul style="list-style-type: none"> -input parameter sMsg: a pointer to message to send. -output parameter quorum_data: the data of the responding acknowledging quorum. -output param quorum_size (the size of quorum_data).

Table 5: *communicate* function details

Message header creation functions	Description
<pre>char* messageToString(pck_t* msg);</pre>	<p>allocates memory for a new string and populate it with the fields as described in table 2. Memory allocated must be explicitly freed afterwards. Note: used by the client to send a request.</p>
<pre>pck_t* stateToMessage(pck_t *recv_msg);</pre>	<p>Creates a new package to send as response, using data from the received message and from current server process state. Memory allocated for return value must be explicitly freed afterwards. Note: this version is intended for use by the server.</p>
<pre>void stateToMessage(pck_t *msg, state_t *state);</pre>	<p>Includes the current client state in an existing package. Note: this version is intended for use by the client.</p>

Table 6: Message creation functions used in client and server implementation

4.3 Server

The servers' main functionality is implemented in the function detailed in Table 7. The basic steps of the server execution are:

- internal state is initialized,
- binds to a newly created socket,
- Initializes thread management data structures and listens for incoming connections.
- On each accepted connection to the listening socket the server spawns a thread to handle the new request on a new socket.

Due to the concept of one thread per client this can severely limit the number of concurrent threads a server can handle simultaneously. For 32-bit OS systems this limit is 512 threads. The `pthread_detach` function is called after a thread creation to let the OS release all the thread resources as soon as it finishes execution. Each thread executes the `serve_thread` function, the code of which is attached on Appendix C.

The `server_thread` function receives the client message and checks if the message body has a value (it may be empty) that needs to be received as well. Note that a value may be present but if it is the same as the server's current value, it is not necessary to actually receive and overwrite the object value. This has a great impact on performance when the message value is a large file.

Each request is processed by the *process* function; it examines the request tag and updates the server state according to the algorithm. Finally a response message is created and sent to the client.

<code>void initialize();</code>	Initialize server state
<code>int create_socket(&socketFd); //1. Create the socket</code>	Create a new socket
<code>bind_socket(&socketFd, &server, sys_conf.serverPort[pid]);</code>	Bind socket
<code>listen(socketFd, MAX_PENDING); //3. Set Socket to Listen</code>	Listen for connections
<code>while(1){ acceptReq(socketFd, &newSocket, &client, rem); pthread_create(&threads[i], &attr, (void*) pt2ProcessThread, (void*)&threads_data[i]); }</code>	Accept any incoming connection and create a new thread to serve it.
<code>void* serve_thread(void* thread_args);</code>	The main serving code of the server. Generally it calls: 1. recvReq() 2. recvMsgVa() 3. process() 4. sendRes
<code>pck_t* create_message(msg_t t, obj_t ot, alg_t alg);</code>	Allocates and initializes memory for a new message.
<code>int recvReq(int, pck_t*, int*);</code>	Receive request message
<code>int recvMsgVal(int, pck_t *,int, bool_t);</code>	Receive message value if needed
<code>void process(pck_t*);</code>	Process request: update server state and prepare response
<code>void sendRes(int, pck_t*);</code>	Send response

Table 7: Main server functions

4.4 Writer

The writer process calls the writeObject function whenever it needs to write a value to the atomic object. After each write operation the writer sleeps for a

preconfigured amount of time. The write interval between operations can be configured through the `confic.ini` file. The `writeObject` function code initializes a message to send, sets the writer process state to `WRITE`, increments the operation counters and then calls the `communicate` function to send the message.

Upon successful completion of the *communicate* function the responses received from the quorum are processed by the *process* function. The latter examines the responses from all the servers, updates the writer current state and proceeds to a second communication round if necessary.

In Table 8 the main functions of the writer process are summarized.

void writeObject(int objectId, state_t*objectState, int * intVal, obj_t objType, alg_t algType);	Performs a write operation, input parameters are the object id, the current writer state for the supplied object id, the integer value to write, the object type(file or integer) and the algorithm type(SIMPLE or SFW)
int compareTag(tag_t* a, tag_t* b);	Compares the two input parameters: If $a > b$ return 1, If $a == b$ returns 0 and -1 otherwise.
bool_t process(state_t *, data_t*, int);	Process the server responses and act according to the current executing algorithm.
int recvMsgVal(int , pck_t *, int);//(newSck, *msg,writeIt)	Receives message value pending on socket.
bool_t writerConditions(data_t* quorum_data, int quorum_size, tag_t** t, bool_t *isPropagated);	The predicate of the writer for the SFW algorithm returns True if predicate is valid, False otherwise.
void stateToMessage(pck_t *, state_t*);	Puts the current writer state into a message.

Table 8: Writer process main functions

4.5 Reader

The main function used by the reader is the *readObject* function, which is very similar to the *writeObject* function used by the writer. It creates a message to send, uses *communicate* function to send it and receives responses in *quorum_data* data structure which is passed to the *process* function for examination. The *Process* function updates the reader state as necessary, which is the most recent object value read and its tag. It also decides if a second communication round is needed in the case of algorithm SFW.

4.6 Code structure and Compilation

The code files are located under the *src* folder and they are organized in folders:

- reader: *reader.c*, *reader_main.c*
- server: *server.c*, *server_main.c*
- writer: *writer.c*, *writer_main.c*
- net: *sockets.c*, *sockets.h*
- utilities:
 - *communicate.c* *communicate.h*
 - *config.c*, *config.h*
 - *log.c*, *log.h*:
 - *message.c*, *message.h*
 - *utilities.c*, *utilities.h*
 - *quorum_gen.c*

- `test/reader/`, `test/writer/`, `test/server/`: all three folders need to exist in order for the make file to output the three executable files.

The Global header files are also located under the `src` folder:

- `data_structures.h`, all data structure definitions
- `mwmr.h`, all include files in one place
- `main.h`, include files, declarations of the functions, macros and global variables that are used by `reader_main.c`, `writer_main.c` and `server_main.c`
- `makefile`, has commands to build the whole source code or parts of it individually for unit testing. Executing `make` under `src` folder creates the executable files.

Code related to sockets such as binding, creating, receiving and sending is under `net/sockets.c`. Code related to communication between processes is in the `communicate.c` and `message.c` files.

Common resources and functionality that is not related to communication, such as management of log files, configuration files, comparison functions on various data structures and common logic are included in the utilities folder and are globally visible to all code in the implementation. An exception is the `quorum_gen.c` which is a separate individual program and is located directly under `src` folder. The `quorum_gen` program takes four (4) command line arguments: the quorum type, number of servers, number of failures and a seed number. The quorum type can be `-m` for majority or `-d` for `server number - failures number` quorum size generation. The output in the first case is `majorities.dat` file where in the latter is `majorities_x.dat`. The output file contains the total number of quorums generated in the first line and the number of failures the quorum system can sustain in the second line. Each line that follows defines a quorum.

Each process has a main file (*reader_main.c*, *writer_main.c*, *server_main.c*), which contains the main function, command argument management functions and signal handling functions. Basically the main files for each process call the *execute* function. The *execute* function loads the *confic.ini* file parameters into a global data structure (*config_t sys_conf*) so that is visible by the whole system, and setups timers for monitoring the operation latency of each read/write operation in the process. There are two operation latency timers that monitor the processor time and the actual real time it takes for a read/write operation to complete.

4.7 Read/write predicates of Algorithm SFW

As mentions in Section 3, the reader and writer algorithms use a predicate to decide when to proceed to a second communication round. The predicate implementation is a challenge due to the existential quantifier.

4.7.1 Reader Predicate

The idea behind the implementation of the predicate is to reduce the size of the solution space while searching for a tag that satisfies the predicate conditions. In order to avoid examining every possible case in the solution space, a heuristic method to move towards the solution is presented.

The algorithm implemented for the reader predicate is shown in Figure 11. Recall that the *inprogress* set contains the latest tag for each writer in the system. The concept of the heuristic algorithm is to find all the possible quorums which include all the servers that responded with the largest tag. The procedure that

evaluates the predicate and calculates the size of the intersection between the responding quorum and a subset of quorums from the quorum system is shown in Table 9.

In more detail, the heuristic implementation first sorts (descending) the unique tags return from the servers into T_k . If many servers responded with the largest tag then only a “few” quorums Q_s (comparing with the total number of quorums $\frac{S!}{(S-T)! T!}$) will exist that include all these servers. Thus the intersection size of Q_s with Q_k (responding quorum) will be big and for this reason is less probable for $I_s \subseteq S_t$, where S_t is a set of servers that have in their inprogress set the t tag being currently examined from T_k .

If the latest tag failed to validate the predicate, then the heuristic implementation examines older tags. As older tags are examined the $|S_t|$ decreases, reasonably fewer servers will have older tags, but $|Q_s|$ increases because more quorums exist that include all servers in S_t . This causes the size of the intersection of quorums in Q_s ($|I_{Q_s}|$) to decrease, resulting in $|I_s|$ ($I_s = \{s: s \in I_{Q_s} \cap Q_k\}$) to also decrease. Thus it is more probable to find a $I_s \subseteq S_t$; this depends on the size of S_t as well. If a tag satisfying the predicate does not exist it will be faster to find and stop earlier in the computation hence saving on average operation latency.

The procedure shown in Table 9:

1. Calculates, by reference parameter output r , the size of the quorums in the intersection of Q_k with Q_s .
2. Returns true or false if the predicate found a valid tag

The pseudocode in Table 9 assumes that is looking for a tag in the inprogress set and not the set of confirmed tags, which is needed in the case an inprogress tag is not found (which is the case shown in Figure 11 at lines 19-24). The reader

```

1: at each reader  $r$ 
2: procedure initialization:
3:  $tag \leftarrow (0,0,0)$ ,  $rCounter \leftarrow 0$ 
4: procedure read()
5:  $rCounter \leftarrow rCounter + 1$ 
6: send  $(R, tag, tag.wc, rCounter)$  to all servers
7: wait until receive  $(RACK, inprogress, confirmed, rCounter)$  from some quorum  $Q \in \mathbb{Q}$ 
8:  $rcvM \leftarrow \{(s, m) : m = (RACK, inprogress, confirmed, rCounter) \wedge s \text{ sent } m \wedge s \in Q\}$ 
9:  $maxC = \max_{m \in rcvM} (m.confirmed)$  /* find the maximum confirmed tag */
10:  $inP = \{(ts, wid, wc) : (ts, wid, wc) \in \bigcup_{(s,m) \in rcvM} m.inprogress\}$ 
11: if  $\exists \tau, MS, B : (\tau \in inP \wedge \tau > maxC) \wedge MS = \{s : (s, m) \in rcvM \wedge \tau \in m.inprogress\} \wedge$   

 $B \subseteq \mathbb{Q} \text{ s.t. } 0 \leq |B| \leq \frac{n}{2} - 2 \wedge I_{B \cup \{Q\}} \subseteq MS$  then
12:    $tag \leftarrow \tau$ 
13:   if  $|B| \geq \max(0, \frac{n}{2} - 2)$  then
14:      $rCounter \leftarrow rCounter + 1$ 
15:     send  $(RP, tag, tag.wc, rCounter)$  to all servers
16:     wait until receive  $(RPACK, inprogress, confirmed, rCounter)$  from some quorum  $Q \in \mathbb{Q}$ 
17:   end if
18: else
19:    $tag \leftarrow maxC$ ;  $MC \leftarrow \{s : ((s, m) \in rcvM) \wedge (m.confirmed = maxC)\}$ 
20:   if  $\nexists C : C \subseteq \mathbb{Q} \wedge |C| \leq m - 2 \wedge I_{C \cup Q} \subseteq MC$  then
21:      $rCounter \leftarrow rCounter + 1$ 
22:     send  $(RP, tag, tag.wc, rCounter)$  to all servers
23:     wait until receive  $(RPACK, inprogress, confirmed, rCounter)$  from some quorum  $Q \in \mathbb{Q}$ 
24:   end if
25: end if
26: return( $tag$ )

```

Figure 11: pseudo code for the reader predicate [12]

predicate must search the confirmed tags if no tag in the inprogress is found that evaluates it.

The purpose of the heuristic is to sometimes find a solution when a solution exists and always find no solution when none exists; this is clearly deduced from the description and the pseudo code presented in Table 9 since the heuristic works on sets that contain tags returned by the responding quorum. A study of the heuristic accuracy and a formal proof of its correctness are beyond the purpose of this thesis and are left for future work.

4.7.2 Writer Predicate

The writer predicate is simpler than the reader predicate. The implementation's pseudocode for the writer is shown in Figure 12. The writer uses the pseudocode as presented in Table 9 to calculate the intersection degree of the

responding quorum with a set of quorums from the quorum system and validate the predicate. The only exception in this case is wherever *wid* appears it refers to the specific writer id which initiates the write request.

```

procedure predicate_and_num_of_quorums_in_intersection( input  $Q_k$ , output  $r$  ):
/*  $Q_k$  the responding Quorum */
rcvMsg  $\leftarrow \{ \langle s, m \rangle : m = (RACK, inprogress, confirmed, rCounter) \wedge s \text{ send } m \wedge s \in Q_k \}$ 
/*find max confirmed tag*/
maxConf =  $\{ \langle ts, wid, wc \rangle : \langle ts, wid, wc \rangle \in m.confirmed \wedge \langle s, m \rangle \in rcvMsg \}$ 
/*find unique tags*/
 $T_k = \{ \langle ts, wid, wc \rangle : \langle ts, wid, wc \rangle \in m.inprogress \wedge \langle s, m \rangle \in rcvMsg \}$ 
sort_descending( $T_k$ ); /*sort  $T_k$  in descending order*/
For each  $t$  in  $T_k$  /*starting from the largest tag*/
if  $t < maxConf$  then
/* put all servers that have in their in progress set the  $t$  tag into  $S_t$  set*/
 $S_t = \{ s : s \in Q \wedge t \in m.inprogress \wedge \langle s, m \rangle \in rcvMsg \}$ 
/*while there is a combination  $S'_t$  from  $S_t$ , initially  $S'_t = S_t$ ,
comb( $cmb, k, n$ ) generates the next combination of  $n$  elements as  $k$  after  $cmb$ ,
where  $k$  is also the size of the subsets to generate */
while comb( $S'_t, k, |S_t|$ )  $\wedge S'_t \neq \{ \}$ 
/*get all Quorums that include all servers from  $S'_t$  and put them in  $Q_s$  */
 $Q_s = \{ s : s \in S'_t \wedge S'_t \subseteq S_t \wedge |S'_t| \neq 0 \}$ 
/*servers in intersection of: all quorums in  $Q_s$  with  $Q_k$ ,
put them in  $I_s$  */
 $I_s = \{ s : s \in I_{Q_s} \cap Q_k \}$  /*finish: if intersection is a subset of  $S_t$  */
If  $I_s \subseteq S_t$  then Return  $t$  /*and predicate is TRUE*/
If not a valid combination exists then  $k--$ ; /*comb( $S'_t, k, |S_t|$ ) is valid? */
else if  $maxConf \geq t$  then
/* If none of  $t$  in  $T_k$  satisfies the predicate then  $t = maxConf$ , predicate is FALSE for
all  $t$  in  $T_k$ .*/
return maxConf /* predicate is FALSE */

```

Table 9: Pseudo code for the reader predicate

```

1: at each writer  $w$ 
2: procedure initialization:
3:  $tag \leftarrow \langle 0, wid, 0 \rangle$ ,  $wc \leftarrow 0$ ,  $wid \leftarrow$  writer id,  $rCounter \leftarrow 0$ 
4: procedure write( $v$ )
5:  $wc \leftarrow wc + 1$ 
6: send ( $W, tag, wc, rCounter$ ) to all servers
7: wait until receive ( $WACK, inprogress, confirmed, rCounter$ ) from some quorum  $Q \in \mathbb{Q}$ 
8:  $rcvM \leftarrow \{ \langle s, m \rangle : m = (WACK, inprogress, confirmed, rCounter) \wedge s \text{ sent } m \wedge s \in Q \}$ 
9:  $T = \{ \langle ts, wid, wc \rangle : \langle ts, wid, wc \rangle \in m.inprogress \wedge \langle s, m \rangle \in rcvM \}$  /* find unique tags */
10: if  $\exists \tau, MS, A : \tau \in T \wedge MS = \{ s : \langle s, m \rangle \in rcvM \wedge \tau \in m.inprogress \} \wedge$   

    $A \subseteq \mathbb{Q} \text{ s.t. } 0 \leq |A| \leq \frac{n}{2} - 1 \wedge I_{A \cup \{Q\}} \subseteq MS$  then
11:    $tag = \tau$ 
12:   if  $|A| \geq \max(0, \frac{n}{2} - 2)$  then
13:      $rCounter \leftarrow rCounter + 1$ 
14:     send ( $RP, tag, tag.wc, rCounter$ ) to all servers
15:     wait until receive ( $RPACK, inprogress, confirmed, rCounter$ ) from some quorum  $Q \in \mathbb{Q}$ 
16:   end if
17: else
18:    $tag = \max_{\langle ts, wid, wc \rangle}(T)$ ;  $rCounter \leftarrow rCounter + 1$ 
19:   send ( $RP, tag, tag.wc, rCounter$ ) to all servers
20:   wait until receive ( $RPACK, inprogress, confirmed, rCounter$ ) from some quorum  $Q \in \mathbb{Q}$ 
21: end if
22: return(OK)
23:

```

Figure 12: writer predicate implementation pseudo code [12]

4.8 Correctness and Robustness

Local tests were frequently executed during development to validate that components are working as expected. Once the implementation was complete, manually targeted test cases were performed to validate the correctness and robustness of the implementation. For this purpose a log file is generated during the execution of test cases in which critical values are exported for later analyses in the event of errors. Specific robustness test cases (Table 10: Test cases) are performed. Test cases 1 and 2 verify that reader and writer processes execute correctly. Test case 3 has initially one reader, writer and server and at each execution the writer and server processes are increased. Test case 4 includes multiple reader and writers and one server, at each execution the server processes are increased. At all cases it is crucial that atomicity is not violated due to implementation errors. Test cases 3 and 4

are difficult to practically test due to multiple writers in the system, for this reason only a subset of operations were checked.

		One Server	Two Servers	...	Twenty Servers
1.	One reader(SR)	Test reader messages used in communications Test reader communication with multiple servers			
2.	One writer(SW)	Test writer messages validity with one, two and twenty servers.			
3.	SR (SW→MW)	Test that reader process reads the correct value while the writers and the servers in the system increase.			
4.	MWMR	Is atomicity violated?			

Table 10: Test cases

After the implementation described in Chapter 4 was tested locally for correctness, a robustness stress test was performed. It is important that server processes do not crash due to implementation errors, since general system errors caused in PlanetLab cannot be avoided. To ensure all processes robustness a stress test was performed locally with more aggressive settings than those used in experiment in PlanetLab. The stress setup included a hundred of readers and writers processes and starting from one (1) server multiple test were performed for up to twenty(20) servers.

Specifically, the tests attempted to either crash the servers due to memory leakage or reach system limits, such as max concurrent thread numbers, opened file descriptors, stack size, deadlocks and starvation. The stress test revealed a lot of areas that improvement could be made. Although most major server failures were

counteracted the problem of starvation of concurrent threads trying to acquire access to the atomic register is anticipated. The starvation issue is resolved on the client side with timeout on the server response. The downscale is an increase of fail operations from the client side but this problem only arises when the client number is very large and their operation interval is small. Another workaround to the starvation issue is to increase the timeout on the clients but this applies only on experiments since in real application examples functional requirements may restrict operation latency.

Chapter 5

Experimentation Setup

In this chapter the experimentation setup used during empirical evaluation is detailed. In Section 5.1 the methodology followed and the configuration used are given. Section 5.2 lists the procedure used to execute the scenarios on PlanetLab. Lastly in Section 5.3 the problems and limitations encountered during the configuration and execution of experiments are described.

5.1 Methodology and configuration

To evaluate the algorithm implementations, experiments are executed on PlanetLab and statistics are recorded for slow operations, operation latency and execution time. The quorum system deployment uses the same PlanetLab nodes for server processes in all experiments and scenarios.

Quorums are arranged to have $S - T$ size, where S is a set of servers and T the preconfigured maximum server failures which is configured based on the quorum intersection degree required in each experiment. The quorum intersection degree for each experiment is calculated using $n = \lceil S / T \rceil - 1$. All write operations in algorithm SFW are slow for $n \leq 4$ [12]. In other words, for S servers and intersection degree n the maximum number of server failures is $T = S / (n + 1)$. The system must be robust “enough” in order to avoid read/write operation failures when a few servers crash or are acting very slowly in PlanetLab. Furthermore, enough servers are

needed such that $T > 1$. For these reasons, in most of the experiments scenarios include twenty (20) servers where the quorum system intersection degree is $n = 6$ for $T = 3$. When the quorum intersection degree is different in an experiment, the respective value for T is given.

The algorithm implementations do not explicitly restrict the number of reader and writer processes but the system performance is expected to degrade when a lot of client processes are present in a scenario. To discover suitable configurations for the experiments, preliminary experiments are performed, where various scenarios examine the effect of a variable parameter as it is progressively increased. The parameters that are used as variables, only one at a time for each scenario, are: number of readers, number of writers, read operation interval, writer operation interval and quorum system intersection degree. In each scenario readers execute 200 read operations and writers 200 write operations. Due to the nodes' arbitrary slowness, most scenarios were executed two or three times in order to complete the 200 read/write operations assigned to each client involved in the scenario. An average is calculated when assembling the results from scenarios that executed more than once.

A list of all 20 nodes used for server processes is given in Table 11. In total, 100 PlanetLab nodes were included in the slice by randomly selecting from a list of more than 1000 nodes.

The apparently faster machines, based on their uptime and load during the last week, are selected for execution of the server processes. A list of all machines used for executing client processes can be found in Appendix F.

The PlanetLab machines use Fedora release 8 (code werewolf) kernel versions 2.6.x and their minimum hardware specification are shown in Table 12. The PlanetLab machines have adequate hardware specifications to execute the

experiments needed during empirical evaluation but usually about 1 GByte of memory is available, CPU usage arbitrary varies and disk space is for the most part sufficient.

Server Host Names	
freedom.informatik.rwth-aachen.de	flow.colgate.edu
chronos.disy.inf.uni-konstanz.de	jupiter.cs.brown.edu
dannan.disy.inf.uni-konstanz.de	ebb.colgate.edu
host2.planetlab.informatik.tu-darmstadt.de	pl1.ucs.indiana.edu
adrastea.mcs.suffolk.edu	pl2.planet.cs.kent.edu
75-130-96-12.static.oxfr.ma.charter.com	planetlab04.cs.washington.edu
pl1.grid.kiae.ru	fobos.cecalc.ula.ve
75-130-96-13.static.oxfr.ma.charter.com	ds-pl1.technion.ac.il
host3.planetlab.informatik.tu-darmstadt.de	cs-planetlab4.cs.surrey.sfu.ca
node-1.mcgillplanetlab.org	pl1.rcc.uottawa.ca

Table 11: PlanetLab nodes used for server processes

PCU	CPU	RAM	Disk
Built-in, remote-access power-reset capability, accessible from PLC, such as IntelAMT, HPiLO, DellRAC, IPMIv2, etc.	4x Intel cores @ 2.4Ghz (e.g., quad core or 2x dual core)	4 GByte	500 GB

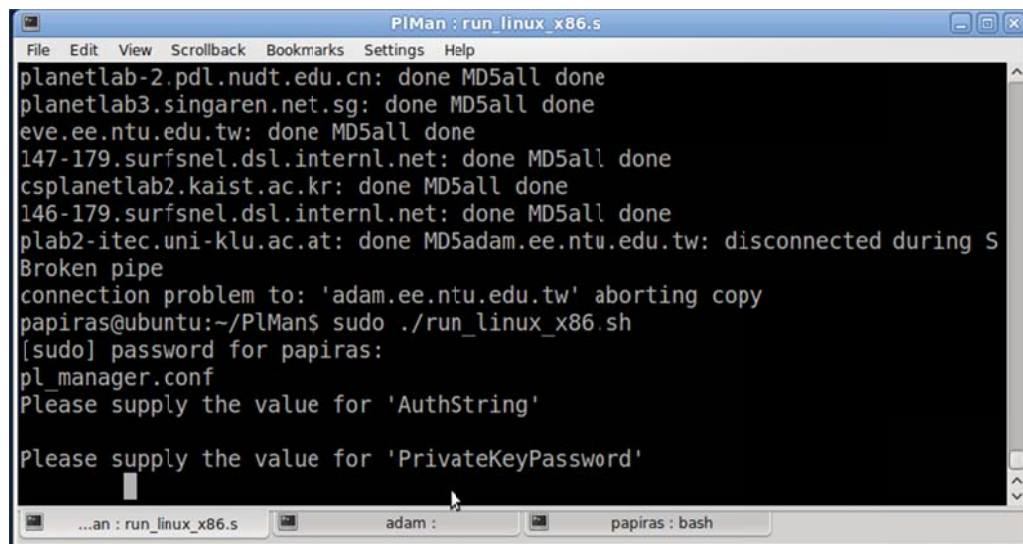
Table 12: PlanetLab machines minimum hardware specifications [28]

5.2 Executing on PlanetLab

To access PlanetLab network an account is needed on planet-lab.org website. Once an account is obtained, it can be used to login to PlanetLab and setup

a slice. After a slice is created nodes can be added to it through the web interface on the PlanetLab site [28]. However in order to securely connect to any node (through *ssh*) a public and a private key must be associated with the account. Step by step instructions for setting up keys can be found on PlanetLab website [28]. The tools used to assist execution of scenarios during empirical evaluation are *PIMan* and *pssh*.

The *PIMan* tool is developed in java and it provides a method to login to a PlanetLab slice (Figure 13), connect with a set of slice nodes (Figure 14), add/remove new nodes to the slice, upload the implementation executable(s) and download the results. While connected to the nodes, *PIMan* can direct any commands needed to the connected cost concurrently. To launch *PIMan* run `./run_linux_x86.sh` from the command line, complete the *AuthString* which is the same as the password used on planet-lab.org website and the *privatekeyPassword* which is the password used at the creation of the private key.



```

PIMan : run_linux_x86.s
File Edit View Scrollback Bookmarks Settings Help
planetlab-2.pdl.nudt.edu.cn: done MD5all done
planetlab3.singaren.net.sg: done MD5all done
eve.ee.ntu.edu.tw: done MD5all done
147-179.surfsnel.dsl.internl.net: done MD5all done
csplanetlab2.kaist.ac.kr: done MD5all done
146-179.surfsnel.dsl.internl.net: done MD5all done
plab2-itec.uni-klu.ac.at: done MD5adam.ee.ntu.edu.tw: disconnected during S
Broken pipe
connection problem to: 'adam.ee.ntu.edu.tw' aborting copy
papias@ubuntu:~/PIMan$ sudo ./run_linux_x86.sh
[sudo] password for papias:
pl_manager.conf
Please supply the value for 'AuthString'

Please supply the value for 'PrivateKeyPassword'

```

Figure 13: *PIMan* slice login, it request *AuthString* and *PrivateKeyPassword*

The *pssh* tool is a command executed through a terminal, for example, the command in Table 13 executes the *comExec* shell command to each of the hosts

listed in *host_list.txt* as user *slice_name*. The output of the command of each host is outputted into the *dirOut*. The *dirOut* directory contains a file with the name of each host which contains the output of the latest execute command.

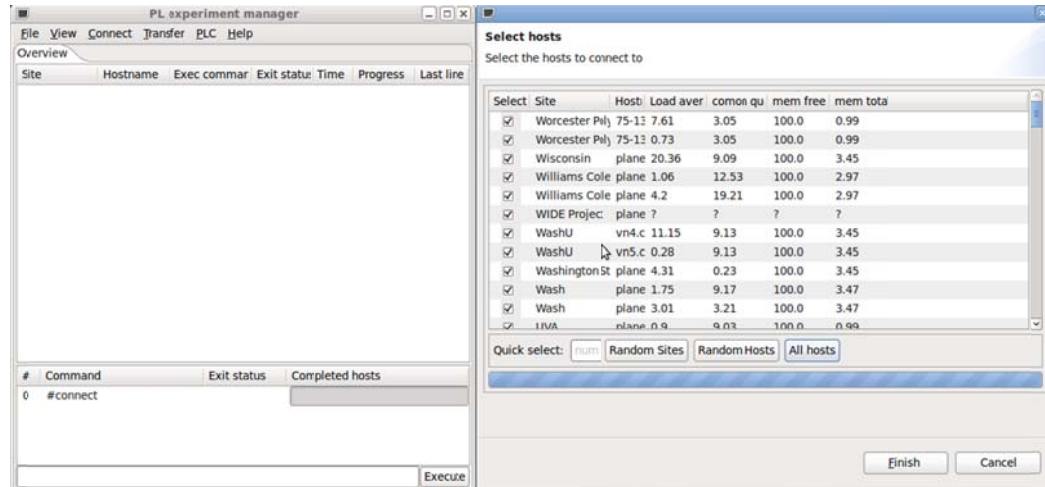


Figure 14: PIMan host selection screen on the right and Overview of connected hosts to the left.

```
pssh -h host_list.txt -l slice_name -o ./dirOut comExec
```

Table 13: example of pssh command

These are the tools used for executing in parallel commands on PlanetLab nodes during the empirical evaluation of the algorithms. The *PIMan* tool is user friendly but it displays only the first line return from each command executed on connected nodes. In contrast *pssh* can execute a command to all hosts and save all output lines from each node in local files for later processing. Practically *PIMan* tool is used to upload the directory tree needed by the scenario execution scripts and to download the whole directory tree that resides in each node when needed. On the other hand *pssh* tool is used for uploading configuration files, scenario startup and monitoring. Concluding the *pssh* tool proved more practical and useful than *PIMan* tool, even though *PIMan* is intuitive because of its GUI.

Executing in parallel hundreds of processes and monitoring them is not a trivial task even with the help of tools. For example, how multiple processes are launched concurrently to multiple nodes with different command line parameters for each process and for each node? For this reason a shell script is created to prepare each PlanetLab node for the execution of a scenario. The shell script code *runscenario.sh* is appended in Appendix B.

In brief, the script requires eight (8) command line parameters:

- total servers in the system,
- number of server failures,
- number of readers,
- number of writers,
- the algorithm type, (0 for SIMPLE, 1 for SFW)
- the register value type, (0 for integer, 1 for file)
- run test mode (1 executes in test mode, 0 executes normally)
- start mode, 1 executes only servers and 0 executes only clients,

The *runscenario.sh* script grants to all executable files permission to execute on the node it runs on and kills all currently executing server, reader and writer processes. The *runscenario.sh* script cleans all preexisting log files and parses a *server.ini* file. The *server.ini* file contains a line for each node required to participate in the scenario. Each line includes the node in the slice, the reader and writers processes it needs to execute in the following format:

The_PlanetLab_Node_ip_or_hostName:ReadersNum:WritersNum

The value of the first (number of servers) *runscenario.sh* script parameter, let it be *num*, also implies that the first *num* lines in *server.ini* file are configured to execute a server process. Finally the *runscenario.sh* shell script starts the execution of the client/server processes needed for the scenario.

The server and client processes need three (3) command line parameters:

- *id*, the unique identifier of the process of integer type that starts from zero and counts first the servers then the readers and finally the writers. The purpose of this id is to define a total order on the priority of processes in the algorithms and for internal book keeping of their state.
- the algorithm type, same as the parameter passed in shell script
- the register value type, same as the parameter passed in shell script

Before the shell script is executed *servers.ini*, *majorities_x.dat* and *config.ini* are uploaded to PlanetLab nodes. The shell script is executed twice with the *pssh* tool, firstly to start the servers execution and secondly to start the clients execution. An example of the steps to start a scenario is given in Table 14 and for downloading the results in Table 15.

5.3 Problems and Limitations

Various values for scenario parameters were tested during the preliminary experiments. Specifically, the number of reader and writer processes in the system, the operation intervals between sequential operations, the quorum intersection degree and the timeout of requests.

Test parameter values were driven by values used in local tests. The local tests were executed in order to provide an initial configuration for the preliminary experiments. Local test cases that include more than 80 processes, executed in a reasonable amount of time when run locally and under a specific configuration. This high number of processes in PlanetLab proved impractical due to the arbitrary slowness of machines and the implementation approaching full service capacity. The server processes reach full service capacity in PlanetLab when overwhelmed from

1	<code>pscp -h servers.txt -l cyprus_ATOMIC ./PlanetLab/scenarios/int.sfw/config.ini /home/cyprus_ATOMIC/PlanetLab/scenarios/int.sfw</code>
2	<code>pscp -h servers.txt -l cyprus_ATOMIC ./PlanetLab/scenarios/servers.ini /home/cyprus_ATOMIC/PlanetLab/scenarios/</code>
3	<code>pscp -h servers.txt -l cyprus_ATOMIC ./PlanetLab/scenarios/majorities_x.dat /home/cyprus_ATOMIC/PlanetLab/scenarios/</code>
4	<code>pnuke -h servers.txt -l cyprus_ATOMIC serverexe pnuke -h servers.txt -l cyprus_ATOMIC readerexe pnuke -h servers.txt -l cyprus_ATOMIC writerexe</code>
5	<code>pssh -h ./test_servers.txt -l cyprus_ATOMIC -o ./outs /home/cyprus_ATOMIC/PlanetLab/scenarios/int.sfw/runscenario.sh 20 4 80 80 0 0 0 1</code>
6	<code>pssh -h ./test_servers.txt -l cyprus_ATOMIC -o ./outs /home/cyprus_ATOMIC/PlanetLab/scenarios/int.sfw/runscenario.sh 20 4 80 80 0 0 0 0</code>

Table 14: Example of steps for starting a scenario

operation requests, causing operation cancellation (due to timeouts in communication).

Although increasing the timeout in communication may allow more processes to participate in the scenario, it leads to a high average of operation latency. During the preliminary experimentation phase it was discovered that the average latency in communication between PlanetLab nodes is 125ms. It is not practical to increase timeout beyond 10 seconds considering that with operation intervals at 1 second and timeouts at 10 seconds, it takes more than one hour to execute a scenario with 400

1	<p>pnuke -h servers.txt -l cyprus_ATOMIC serverexe</p> <p>pnuke -h servers.txt -l cyprus_ATOMIC readerexe</p> <p>pnuke -h servers.txt -l cyprus_ATOMIC writerexe</p>
2	<p>pssh -h ./servers.txt -l cyprus_ATOMIC -o ./results/sfw/writer cat</p> <p>/home/cyprus_ATOMIC/PlanetLab/scenarios/int.sfw/writer*.result</p>
3	<p>pssh -h ./servers.txt -l cyprus_ATOMIC -o ./results/sfw/reader cat</p> <p>/home/cyprus_ATOMIC/PlanetLab/scenarios/int.sfw/reader*.result</p>
4	<p>(a shell script to parse the results file and calculate averages and percentages of required fields, the script code is on Appendix G)</p> <p>results.sh ./results/sfw/reader</p> <p>results.sh ./results/sfw/writer</p> <p>(the script takes one parameter as input: the output directory location of step 2. and 3. Respectively)</p>

Table 15: Example of steps to download results

operations in total. Thus, there is no need to set the operation interval to a higher value than 1 second.

The main problem is not communication latency between PlanetLab machine nodes but the arbitrary slowness of nodes, especially the heavy loaded nodes, thus the execution time slots given to the slice are less. PlanetLab has a fairness resource allocation policy [14]. However, a process may execute slower than expected but it may still be killed if it uses too many resources during high pressure times.

The PlanetLab slice used for the experiments had 100 nodes from which only about 80 nodes were active while others were unavailable due to maintenance. This caused problems when some of the unavailable nodes were selected to execute server processes.

It is also important to note that PlanetLab nodes do not guarantee a static IP or message delays and have varying bandwidth limitations. Configuring the implementation of the algorithms to connect to servers using their IP address can fail, since their IP can change between (re)connections. To solve this issue the IP of the server must be resolved using the host name of the machine but this adds to the communication time. An alternative to PlanetLab DNS servers arbitrary slow behavior is CoDNS which basically gives benefit to anyone who wants more reliable name lookup service [29].

On some of the stress test scenarios where the timeout and capacity of processes is stressed, some node processes are killed and a notification email similar to the one shown in Table 16 is received.

Sometime before Thu Aug 12 16:28:24 2010 GMT, swap space was nearly exhausted on kc-sce-plab1.umkc.edu.

Slice cyprus_ATOMIC was killed since it was the largest consumer of physical memory at 244.4 MB (24.4%) (96.8 MB writable) after repeated restarts.

Please reply to this message explaining the nature of your experiment, and what you are doing to address the problem.

cyprus_ATOMIC processes prior to reset:

PID	VIRT	SZ	RES	%CPU	%MEM	COMMAND
23278	177.5 MB	44.4 MB	122.3 MB	12.1		/home/cyprus_ATOMIC/PlanetLab/writer/writerexe 149 0 1
23213	209.5 MB	52.4 MB	122.1 MB	12.1		/home/cyprus_ATOMIC/PlanetLab/reader/readerexe 49 0 1

Thu Aug 12 16:28:24 2010 GMT kc-sce-plab1.umkc.edu reset cyprus_ATOMIC

Table 16: PlanetLab kill auto send message example

When a scenario execution starts all client processes start sending requests concurrently, overwhelming the servers. This is also reflected in the results where the majority of operations latency at the beginning of the scenario execution is very high and during the execution it slowly drops to more expected values. This initial burst causes a high average of latency for operations and a high percentage of failures due to communication timeouts. For these reasons a small wait interval was added to each client process before scenario launch, in order to provide a less aggressive behaviour at the beginning of scenarios. The initial wait interval that each process waits before it starts its execution is fixed for all scenarios and it can be configured from the *confic.ini* file.

During the empirical evaluation of the algorithms a lot of problems appeared due to the nature of PlanetLab. Servers were arbitrarily going offline or were too slow to communicate with. During the file upload of execution files for the experiments some servers received the latest configurations while some others did not. This created a lot of confusion, making the file upload procedure a tedious task to accomplish.

Downloading the results of scenario executions faced the same problem causing even more delays to a scenario execution. To download the results from the PlanetLab nodes `cat reader*.result` unix command is used as the *execCom* parameter to pssh. As a result to pssh command all result files were downloaded to a local folder.

Chapter 6

Empirical Evaluation

In this chapter the experiments, scenarios and their parameters used during empirical evaluation are described. Subsequently for each experiment, the scenario results are represented in graphs and analysed.

6.1 Experiments and Scenarios

In this section the parameters for each experiment and their scenarios are detailed. The parameters that are the same for all scenarios except when explicitly stated otherwise are:

- Read operation interval: 1 second
- Writer operation interval: 1 second
- Quorum system intersection degree: $n = 6$, failures $T = 3$
- Communication timeout 10 seconds
- 20 servers

Using a large number of PlanetLab nodes for object replicas and expecting only $T = 3$ of them to not fail is impractical. In practice, the operation latency of operations will further increase as reader/writers must wait responses from a larger set of servers and the possibility some of them are performing arbitrarily slow is high. For these reasons, the number of servers was set to 20 and the server failures were varied from 1 to 3.

6.1.1 Experiment 1: Number of readers and writers effect

The purpose of this experiment is to study how the number of readers and writers affect the efficiency of algorithm SFW. The results obtained from this experiment are used to define a reasonable number of readers and writers to use in further experiments. The number of readers and writers differs in each scenario.

Scenario 1: Number of Writers

The efficiency of algorithm SFW is investigated when the number of writer processes in the system increases. Experiments are run with 10, 30, 40, 50, 80 writers while the number of readers remains 80.

Scenario 2: Number of Readers

In this scenario the efficiency of algorithm SFW is investigated when the number of readers in the system increases. Experiments are run with 10, 30, 50, 80 readers while the number of writers remains 80.

Scenario 3: Operation Interval

The purpose of this scenario is to study the impact of operation frequency on the efficiency of algorithm SFW. For this purpose the reader (rI) and writer intervals (wI) are varied as follows:

- $rI > wI, rI = 1, wI = 10$
- $rI < wI, rI = 10, wI = 1$
- $rI = wI = 1$

The value of 1 second is chosen based on the observed PlanetLab average latency in communication as explained in Section 5.3. The value of 10 seconds is chosen such that there is a great difference between rI and wI in order to get clear results. Experiments are run for 10 readers - 80 writers and 80 readers - 10 writers.

6.1.2 Experiment 2: Quorum Intersection Degree

In this experiment the effect of the quorum system intersection degree to the efficiency of algorithm SFW is investigated. The following scenario is considered.

Scenario: Effect of Quorum Intersection Degree

After observing the results of Scenarios 1 and 2 of Experiment 1, the number of writers and readers are set to 40 and the operation interval is set to 1 second.

Recall that algorithm SFW's write predicate allows fast write operations for quorum system intersection degree, $n > 4$ (note that n cannot be greater than $S - 1$). In this experiment the parameters used for servers $S = 20$ and server failures $T = 3$. In the case of 20 servers, the values for n used in the experiment is $\{6, 9, 19\}$, calculated as follows:

- $n = \lceil 20/3 \rceil - 1 = 6, T = 3$
- $n = \lceil 20/2 \rceil - 1 = 9, T = 2$
- $n = \lceil 20/1 \rceil - 1 = 19, T = 1$

6.1.3 Experiment 3: Comparison of the SFW with the SIMPLE algorithm

In this experiment the average operation latency of algorithm SFW is compared against the latency of algorithm SIMPLE. The following scenario is considered.

Scenario: Increasing readers and writers

The scenario is executed with reader and writer processes at 10, 20, 30 and 40 respectively. The two algorithms' performance is compared in respect to their average operation latency, average CPU cycles consumed (execution time), operation failure percentage due to timeouts and percentage of fast operations executed for algorithm SFW.

6.2 Results

The experiments and their results are depicted in graphs and analyzed separately for each scenario.

6.2.1 Experiment 1

Scenario 1: Number of Writers

The effect of changing the number of writers is investigated while the number of readers is fixed to 80. Different plots are presented that show how the amount of writers affects the percentage of fast operations, operation latency and failures. Note that the operations that timeout are not included in the calculation of the results for Figure 15, 16 and 17.

It is expected that the percentage of fast write operations will decrease while the number of writers increase. The readers are expected to perform a higher

percentage of fast read operations since it is more likely their predicate to be validated (return TRUE) on larger *inprogress* sets.

Before the turning point at $W \leq 40$ in Figure 15, the results observed are similar to the expected results. As writers increase they perform fewer fast write operations. This is expected behaviour since the writers proceed to a second

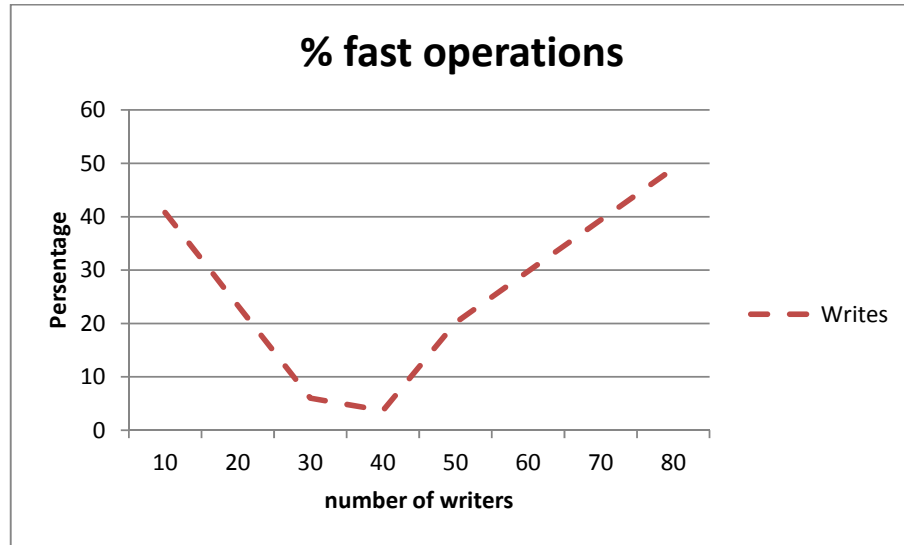


Figure 15: Percentage of fast write operations with 80 readers

communication round if there is a tag returned by the servers (of the quorum that replied) that is distributed among enough quorums. When there are more readers in the system it is more difficult to validate the predicate due to multiple concurrent write operations.

As mentioned, a turning point exists at 40 writers in Figure 15 (the average percentage of fast writes is 3.7% for $w = 40$). At the turning point the performance regarding the percentage of fast write operations is the worst, which indicates that the system approaches its capacity limit. This limit means that there are so many write and read processes in the system that the percentage of fast write operations is reduced. After the turning point the percentage of fast operations increases. Studying the log files generated during the execution of the scenario the following are

observed: some processes have crashed or are very slow, others suffer from long service starvation, while some other processes are serviced more frequently due to latency and the asynchrony of clients with the servers. Another reason could be that geographical proximity favours' some clients. The value of 80 writers is included in the results for completeness, to show that the system reaches its limits and is not used to draw conclusions on the efficiency of algorithm SFW.

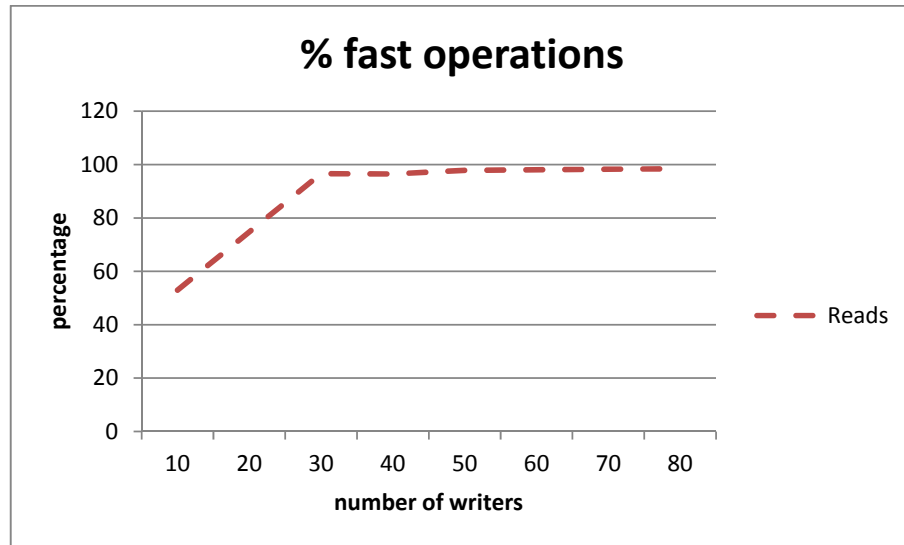


Figure 16: Percentage of fast read operations with 80 readers

In Figure 16 it is shown that the percentage of fast read operations increases, as the number of writers increase. This is in accordance to our expectation. The readers will first check if the max confirmed tag returned by the replying quorum is greater than any other tag in the *inprogress* set. With increasing number of readers, the read requests will be more frequent than write requests and as such it is more likely a fast read operation to occur. When the system has over 30 writers the percentage of fast read operations becomes stable, near 100%, hence this result gives a good indication for the efficiency of read operations of algorithm SFW.

In Figure 17, the operation latency of read operations is stable with increasing number of writers, which is expected, given the results in Figure 16. When $W = 40$,

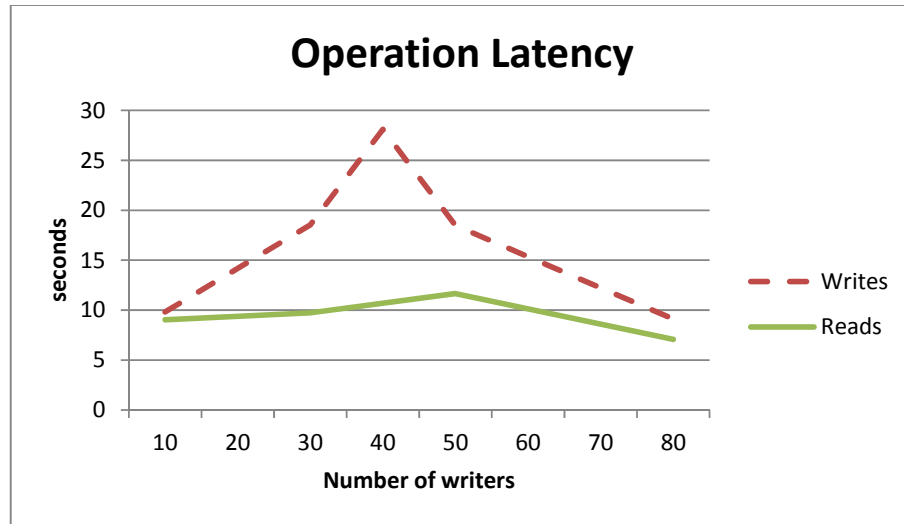


Figure 17: client performance: average operation latency

the operation latency of write operations is at its peak (28 seconds). Observe that the curve of Figure 17 (the write operations latency increases) is relatively the inverse of the curve shown in Figure 15 (the percentage of fast write operations decreases).

In Figure 18 the fail percentage caused by timeout of read operations increases as the number of writers increase until $W = 30$. Readers' fail percentage is at its peak when 30 writers are in the system since the readers in this case perform

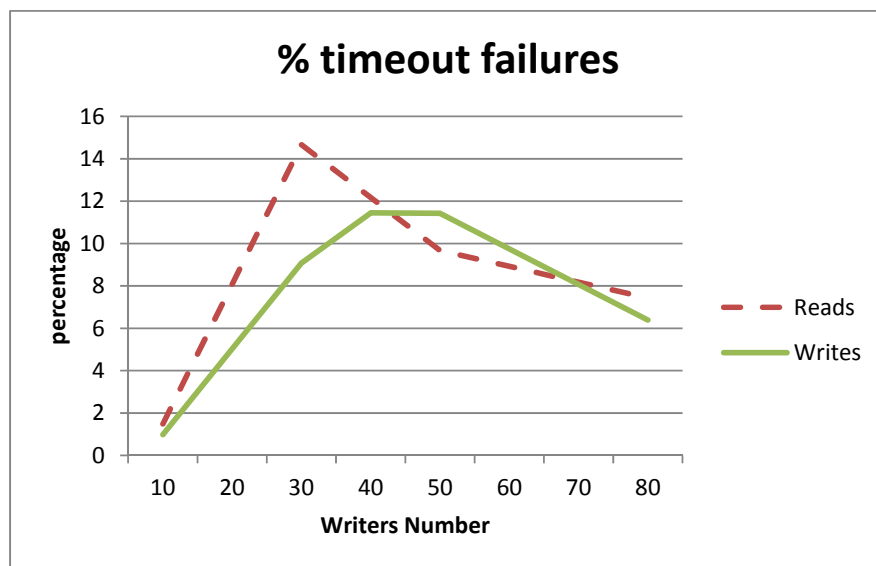


Figure 18: percentage of client timeout failures

mostly fast operations (96.6%) and they “compete for access” to the object’s replicas. Observe that the read operation latency is a bit higher for $W = 30$ than $W = 10$.

After $W = 30$, the failure timeout of read operations is unexpectedly decreased, because after $W = 30$ all read operations are fast and write operations are slow (up to $W = 40$) giving more access to the atomic register to readers. This also agrees with the fact that a lot of reader processes have crashed.

The write operation timeout percentage is increased until $W = 50$ since the number of writers is increased and there is more traffic towards the servers. As we approach 40 writers the percentage of fast write operations dramatically drops (Figure 15) and write operations latency radically increases (Figure 17), which causes more failures due to timeouts. After $W = 50$, as already mentioned, the system approaches its limit. The timeout percentage value after $W = 50$ is not representative since a lot of processes crash.

Scenario 2: Number of Readers

As an addition to the previous scenario, the effect of the number of readers is investigated while the number of writers is fixed. It is expected that the percentage of fast read and write operations will not be affected by the increase of the number of reader processes because their predicates are not influenced by the number of readers. The operation latency, of both read and write operations, is expected to increase when the number of readers increase since the load on the servers is increased.

As it can be seen in Figure 19, the percentage of fast write operations increases linearly from 10 to 30 readers. This behaviour is possible since more processes are executing concurrently competing for service, which intuitively reduces the number of writers gaining access to servers because they are being overrun by

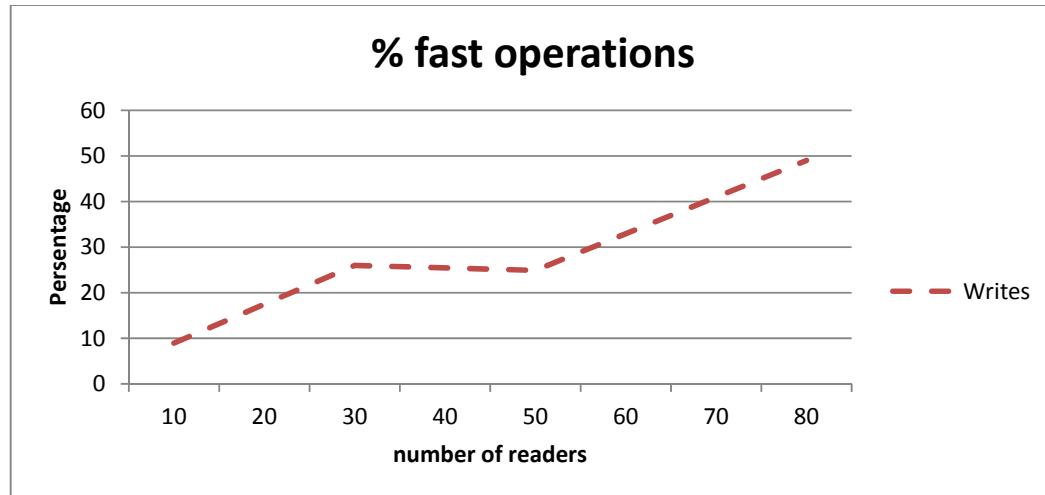


Figure 19: Percentage of fast write operations as the readers increase

the readers. In other words, it is more likely less writers to execute concurrently, since a portion of the servers capacity is been “attained” by readers. From 30 to 50 readers the percentage appears more stable (on average 26%). After the number of reader processes exceed 50 the fast write operation percentage is increased, which may be caused by the same reasons explained in Scenario 1. At 80 readers the system is unstable since, as observed from the log files, a lot of processes crash thus fewer read and write processes are actually executing in the system.

Essentially, the increased percentage of fast write operations at 80 readers is not representative but is included in the results for completeness.

In Figure 20, increasing the number of readers does not affect, by a noticeable degree, the percentage of fast read operations since it only drops by 2% (from 99% to 97%) from 30 to 50 readers. As mentioned, this is indeed our expectation.

The average operation latency of read operations, in Figure 21, is not significantly affected by the number of readers. On the other hand, the operation latency of write operations is substantially affected by the number of readers. Specifically, there is a notable decrease of the writer’s latency between 10 and 30

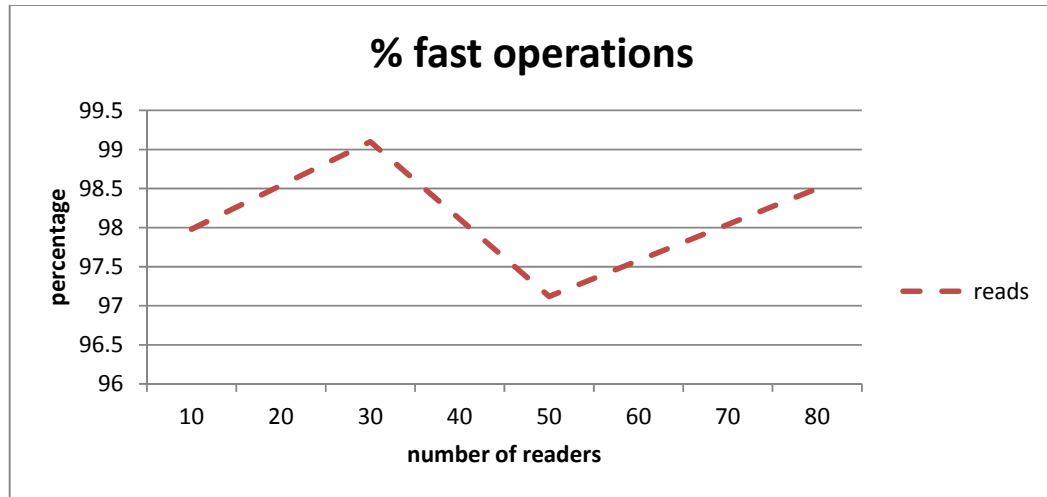


Figure 20: percentage of fast read operations as the readers increase

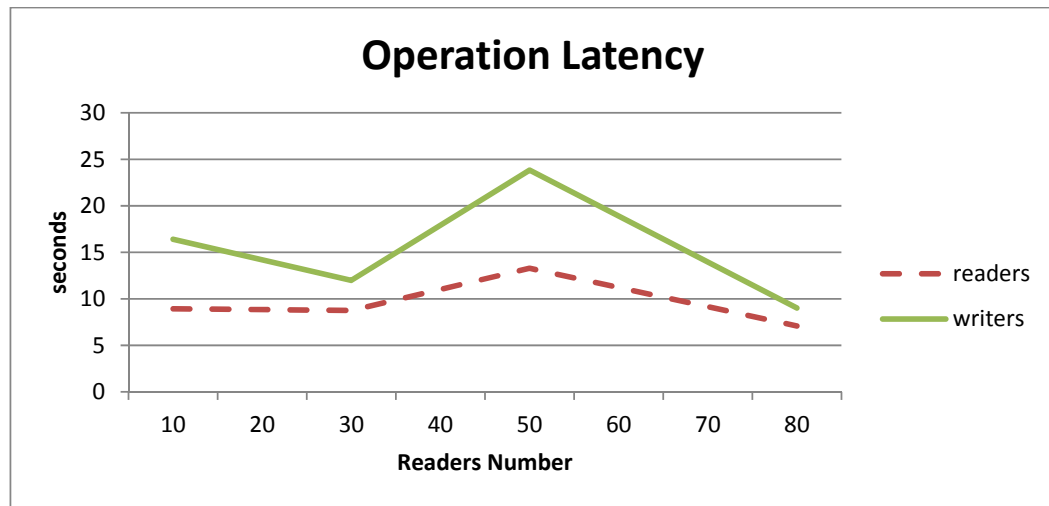


Figure 21: The average operation latency while reader processes increase and writers are fixed

since in Figure 19 for the same range there was an increase in the percentage of fast write operations. Between 30 and 50 readers a big increase in writer's operation latency is observed. Considering that for the same range the percentage of fast write operations was stable, it seems that the system approaches its capacity limit at 50 readers. At 80 readers the system is unstable since, as observed from the log files, a lot of processes crash.

The timeout failures of write operations (in Figure 22) are relatively stable as

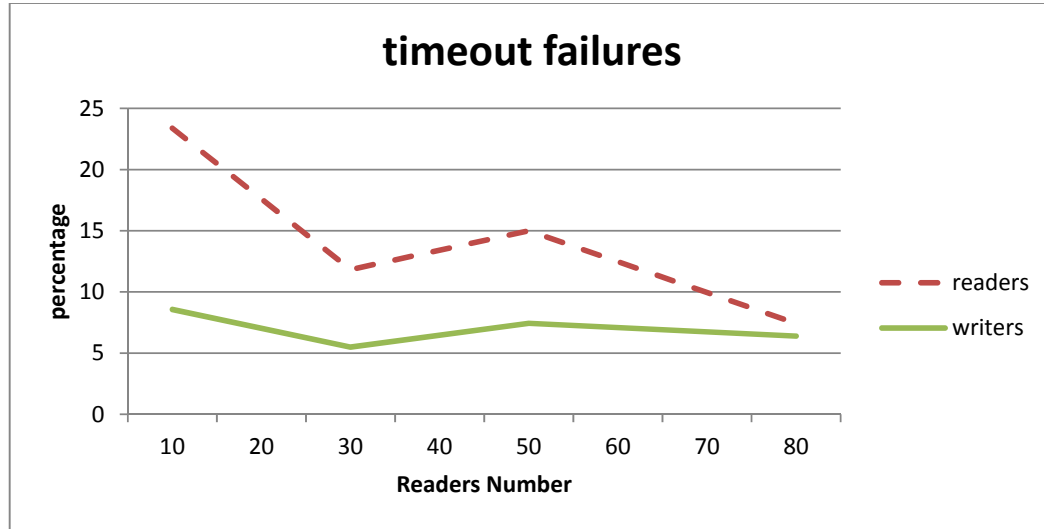


Figure 22: percentage of failures from timeouts while reader processes increase and writers are fixed

the number of readers increase. The read operation latency and percentage of slow read operations at 10 readers is not higher than at 30 readers but the fail operations when considering 10 readers (in Figure 22) is unreasonably high. Observing from the log files we see that there are a few readers which perform almost only fail operations due to timeouts. These PlanetLab machines are either very slow due to transient load or they are experiencing network congestion. When there are more readers these slow machines have less effect on the results.

Scenario 3: Operation Interval

Recall that read interval is denoted by rI and write interval by wI . For the read operations, when $rI < wI$ (1sec < 10 sec), it is expected that a high percentage of one round (fast) read operations will occur. In the case of write operations, a slower write frequency is expected to lead to one round (fast) write operations.

For $rI > wI$ (10sec > 1sec) it is expected to see a high percentage of fast read and write operations. Since the read operations are infrequent and do not

consume server resources, the percentage of fast write operations is expected to be high. Since the write operations are frequent and they finish earlier they stop consuming server resources leaving only readers to execute in the scenario. It is expected to result to a high percentage of fast read operations.

For $rI = wI$ (1sec), it is expected that both readers and writers perform fewer fast operations than in $rI < wI$ or $rI > wI$.

Results are close to the expected but in most of the cases there is no noticeable influence to the percentage of fast read operations (Figure 23). A noticeable effect of the read and write operation interval is observed when the number of readers is significantly more than the number of writers regarding write operations (Figure 24). The percentage of fast write operations in this case is increased when $rI = wI$ and $rI > wI$.

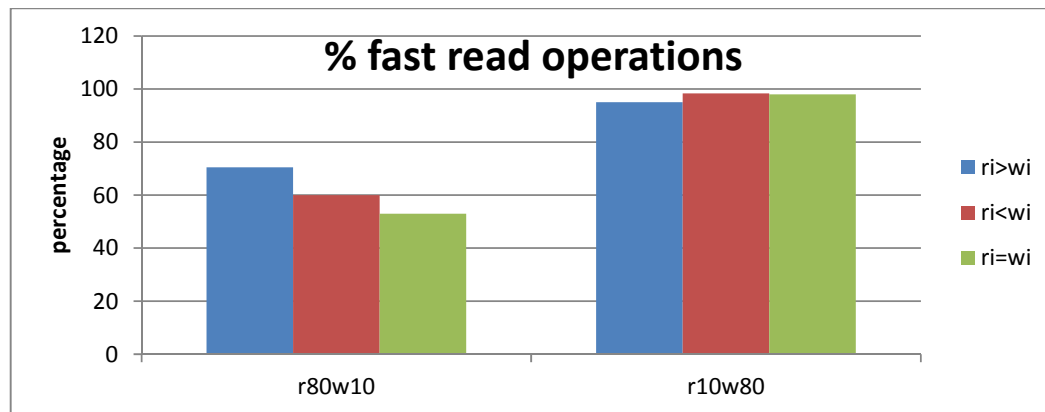


Figure 23: Effect on % of fast read operations, on the vertical axis is the percentage and on the horizontal axis there are two categories, (1)80 readers and 10 writers, (2) 10 readers and 80 writers

In the case of $rI = wI$ the percentage of fast write operations is the highest because there are few writers affecting the server tags. In the case of 80 writers and 10 readers, regardless of the chosen interval the percentage of fast write operations is low affected by the high number of writers in the system.

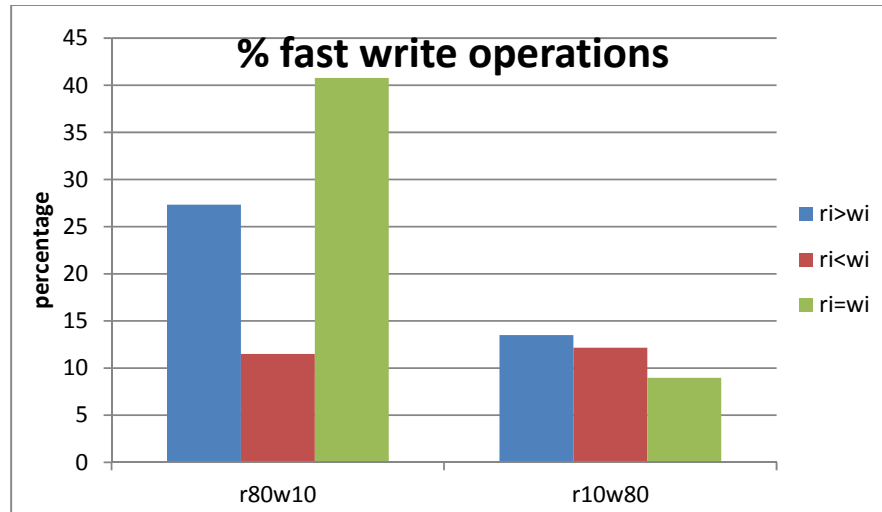


Figure 24: Effect on % of fast write operations, on the vertical axis is the percentage and on the horizontal axis there are two categories, (1) 80 readers and 10 writers, (2) 10 readers and 80 writers

Summary

Concluding from Scenario 1 and Scenario 2, it was indicated that the number of writers and readers in the scenarios should be low enough (less than 80) to provide stability of the system while scenarios are executed. In order to balance service congestion the number of processes could be leveraged around 40 readers and writers or by increasing the operation interval.

The system exhibits reasonable behaviour when the number of readers is equal with the number of writers in the system. In the case of fewer readers there are more read failures but nonetheless the readers perform fast read operations. Overall, it is desirable to keep the average operation latency low enough such that the scenarios in subsequent experiments complete in a reasonable amount of time.

From Scenario 3, it was observed that the read and write intervals do not greatly affect the efficiency of the algorithm. However, in the case of 80 readers and 10 writers a notable difference in the percentage of fast write operations was observed with the highest percentage being when $wI = rI$. Thus, it was decided to keep the read interval equal to the write interval in further experiments.

6.2.2 Experiment 2

The second experiment focuses on the quorum intersection degree of the underlying quorum system. Recall that a single scenario was considered here (effect of quorum intersection degree).

It is expected that for $n > 4$, the percentage of fast write operations increases due to the writer predicate that requires the writer tag to appear in an intersection with at most $n/2 - 1$ other quorums. As n increases so do the intersected quorums, thus their intersection size decreases and thus the predicate is easier to validate. Recall that n cannot be greater than $S - 1$. The results are close to the expected ones (Figure 25).

The percentage of fast read operations is expected to increase due to the reader predicate that examines all tags returned from servers and needs a tag to appear in the intersection with at most $n/2 - 2$ other quorums. As n increases so do the quorums that get intersected, thus their intersection size decreases and the predicate is easier to validate.

The percentage of fast read operations is high at all values for n (Figure 26). There is a slight drop on the percentage on high values of n (i.e., $n = 19$) that may be caused by the high percentage of operation timeouts (Figure 28) discussed later.

It is expected that the operation latency increases as the quorum intersection degree increases, since an operation waits for more servers to respond. In Figure 27 it can be observed that the operation latency of the read and write operations increases for values $n = 6$ and $n = 9$. Even though the percentage of fast write operations is the same when n is 6 and 9, the average write operation latency doubles. The same is true for read operations that exhibit less fast operations. This increase in the operation latency can be explained by the fact that clients expect more servers to respond at each communication round.

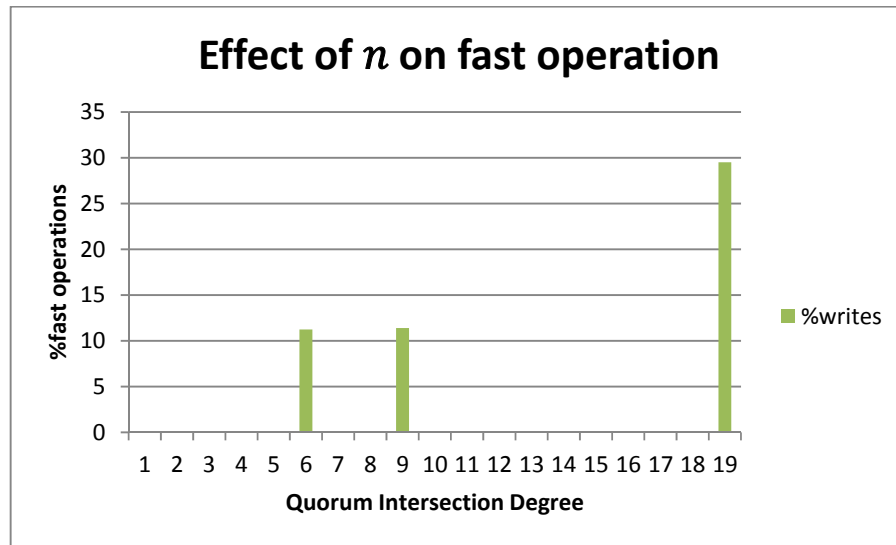


Figure 25: The percentage of fast write operations with 40 readers and 40 writers, while n increases.

In the case of $n = 19$, the operation latency is still high but less than $n = 9$ since the percentage of fast write operations is the highest for $n = 19$.

The percentage of fail operations in Figure 28 shows an increase of failures while the intersection degree increases, due to the fact that clients expect more servers to respond during a communication round.

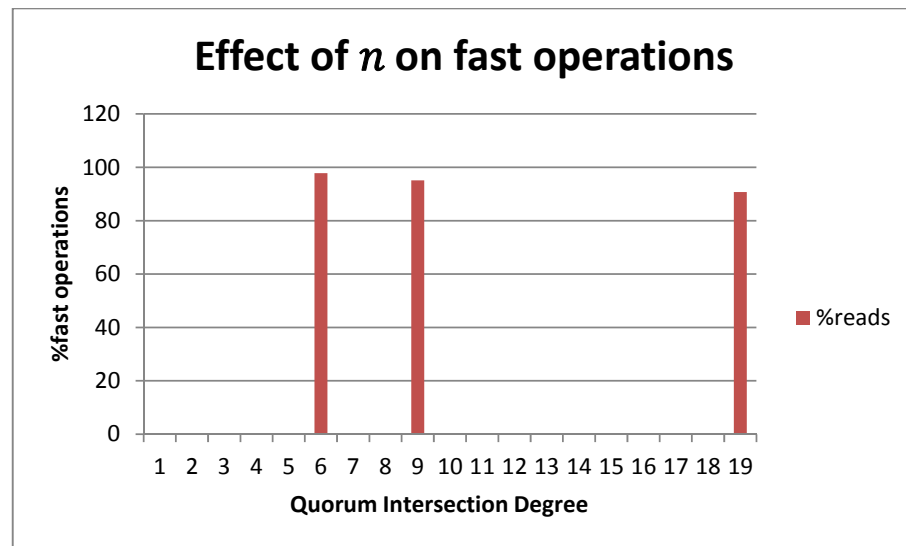


Figure 26: The percentage of fast read operations with 40 readers and 40 writers, while n increases

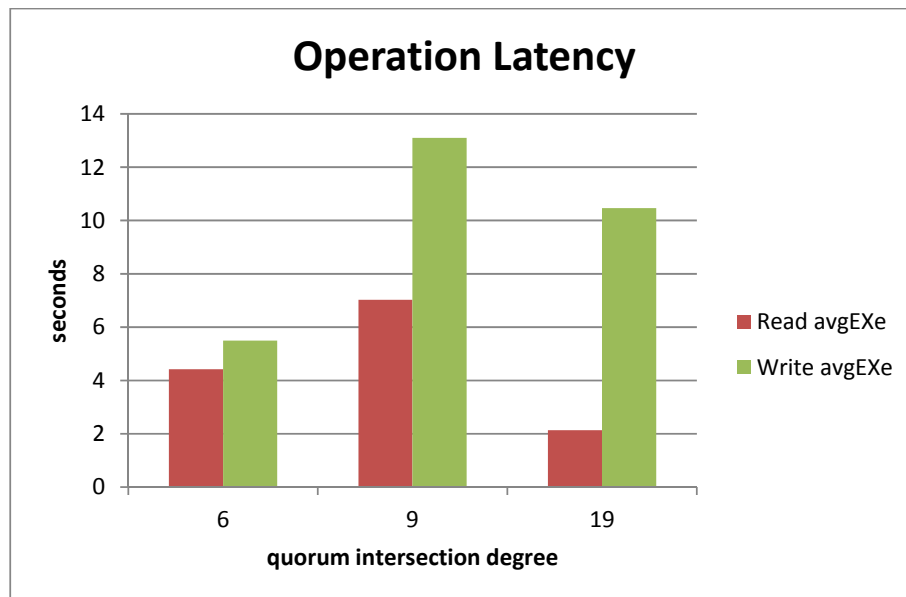


Figure 27: 40 reader and 40 writer time of execution in respect to n

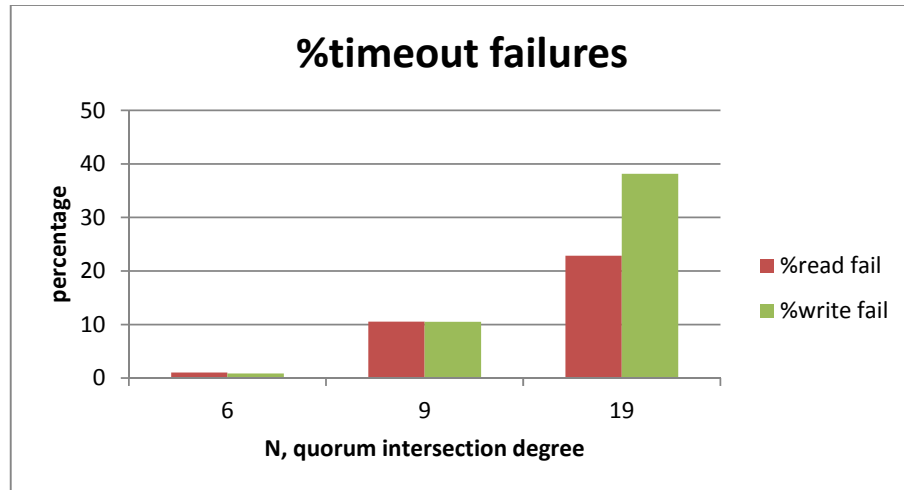


Figure 28: Percentage of fail operations due to timeouts in communication in respect to n

Summary

The results are close to the expected ones when examining the percentage of fast write operations. The percentage of fast write operations increases as the quorum intersection degree increases. On the other hand the percentage of fast read operations slightly drops as the intersection degree increases because there are a lot of failures due to timeouts. The operation latency increases as the quorum intersection degree increases. When a high percentage of fast operations exists then the operations latency is balanced off. From our experiments we observe that the quorum intersection degree is an important parameter, since in most cases it greatly affects the efficiency of algorithm SFW.

6.2.3 Experiment 3

In the last experiment the efficiency of algorithm SFW and is compared with the efficiency of algorithm SIMPLE. Recall that a single scenario was considered for this experiment as well.

It is expected that algorithm SIMPLE's operations will consume on average less CPU time than of the algorithm SFW, because SFW must also validate its operation's predicates. On the other hand, algorithm SFW's operation latency for read and write operations is expected to be less than of SIMPLE, given that algorithm SFW allows fast operations. In respect to failures, due to timeouts, it is expected that SFW will generally have a higher percentage of failures than SIMPLE, since SFW must wait for more server acknowledgements at every round of communication. This behavior is expected to be more noticeable when SFW performs mostly slow read or write operations.

The results for CPU time consumption for the read operations are close to our expectations. In the case of read operations (Figure 29), algorithm SIMPLE's CPU time consumption appears to be constant while processes increase. Algorithm SFW's read operations always need more CPU time than of SIMPLE. Increasing the number of processes also increases the time needed to validate the reader predicate, increasing the gap between SFW and SIMPLE even more.

Algorithm SIMPLE's write operations CPU time is stable as the number of processes increase in the system as well. Instead, algorithm SFW's write operations CPU time increases linearly with the increase of processes in the system (Figure 30), as expected.

On average, a read operation takes more time to execute in algorithm SIMPLE than in algorithm SFW and increasing the number of processes also increases their difference (Figure 31), since SFW allows fast read operations as depicted in Figure 32. Given the percentage of fast read operations at 30 readers and writers (and below), the difference in operation latency between the two algorithms is disappointing. However the efficiency of algorithm SFW is clear at 40 reader and writers where there is around 7 seconds difference in operation latency.

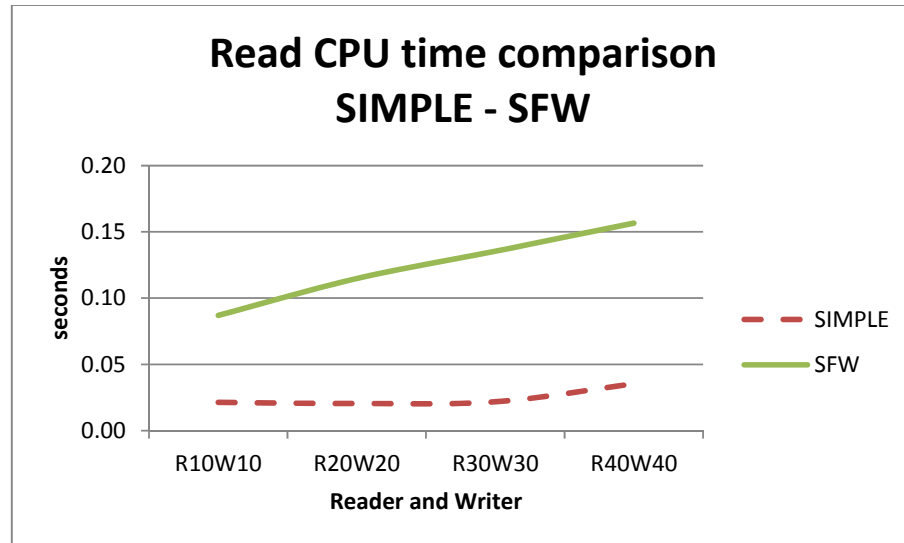


Figure 29: Read operations CPU time comparison of the SIMPLE with the SFW algorithm

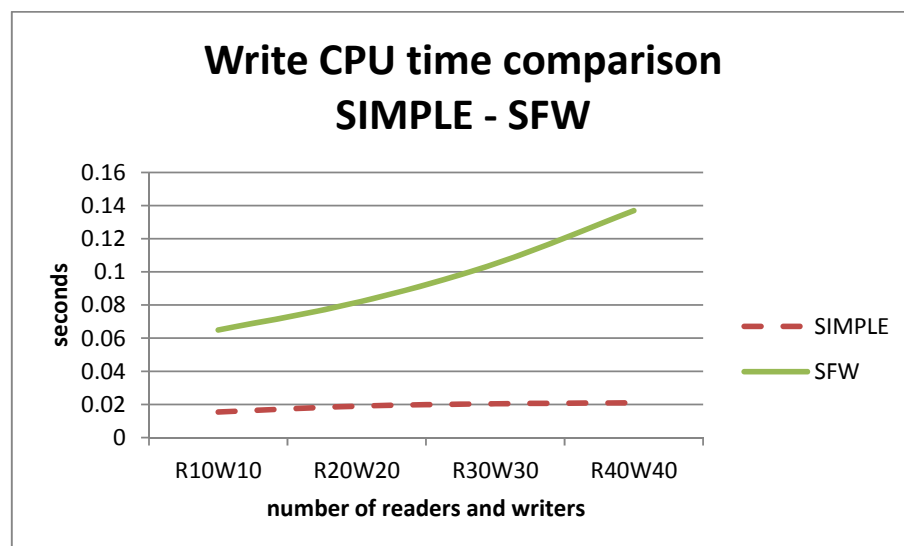


Figure 30: Write operations CPU time comparison of the SIMPLE with the SFW algorithm

This suggests that even larger difference will be witnessed at larger number of readers and writers.

In the case of write operations, per Figure 33, algorithm SIMPLE's operation latency increases linearly when the number of processes increases, which was

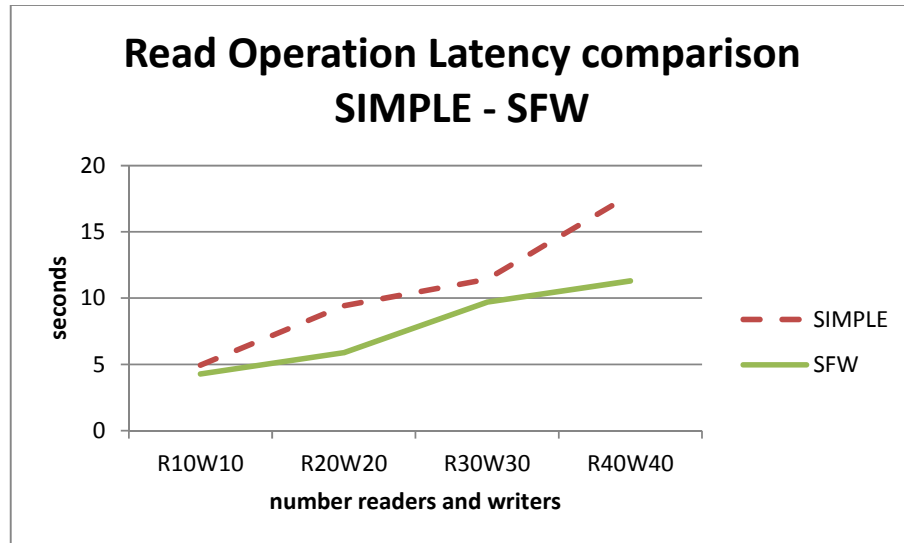


Figure 31: Read operation latency comparison of the SIMPLE with the SFW algorithm

expected since the bottleneck are the servers's capacity to serve (based on bandwidth, hardware specifications and load). In the range of 10-20 reader and 10-20

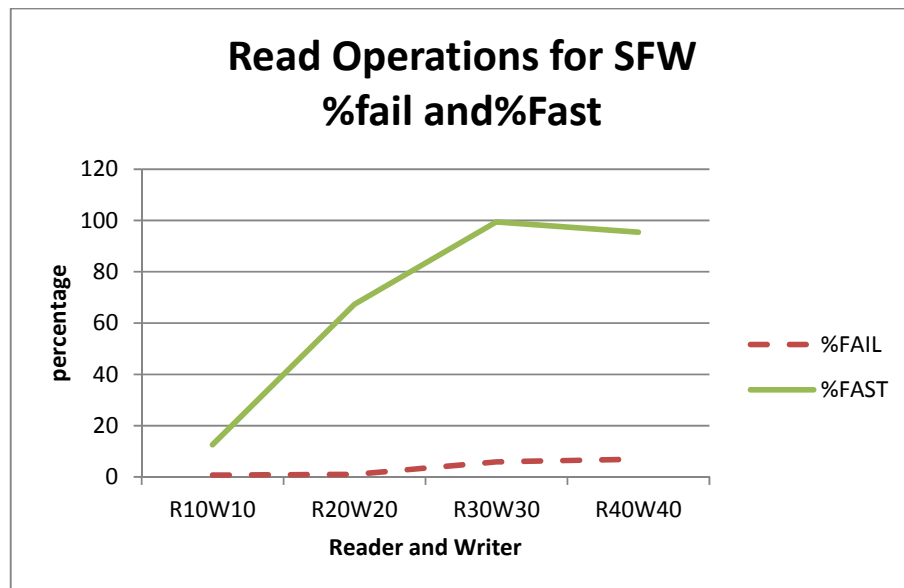


Figure 32: Read operations fail% and fast% for the SFW algorithm

writer processes in the system, algorithm SFW's write operations need slightly less time to execute because the percentage of one round write operations also increases

(Figure 34). In the range of 20-40 readers and writers algorithm SFW's write operations' execute slower than of SIMPLE with the peak of difference at the point where SFW performs 6.3% (Figure 34) of one round fast write operations. Algorithm SFW is slower in this case (30 reader and writers) due to the high write CPU execution time as seen in Figure 30. Again the difference between the two algorithms regarding operation latency is disappointing. Considering the high percentage of fast write operations at 20 readers and writers the operation latency of algorithm SFW is only 0.2 seconds better than the SIMPLE; we expected algorithm SFW to perform write operations in less time. One may conclude that for the chosen experiment values, the CPU time required by algorithm SFW voids the reduced time of fast writes.

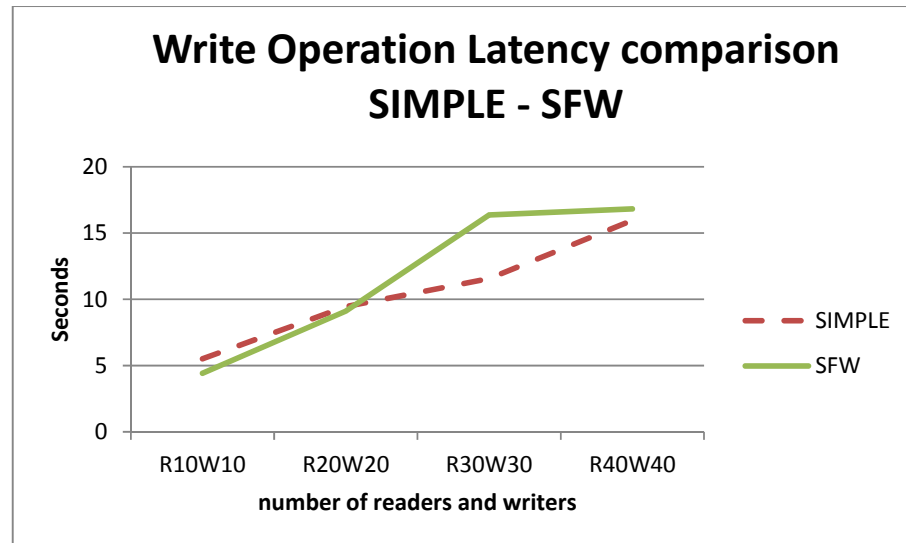


Figure 33: Write operations operation latency comparison of the SIMPLE with the SFW algorithm

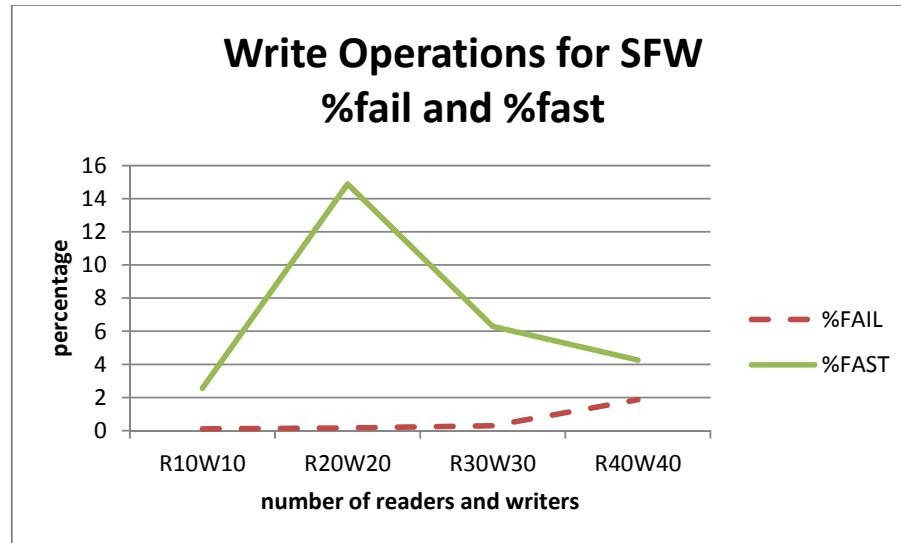


Figure 34: Write operations %fail and %fast for the SFW algorithm

Summary

Assuming no more servers than $T=3$ will fail and under a reasonable setup of readers and writers in the system, algorithm SFW's efficiency is better than of algorithm SIMPLE's efficiency, but by a small margin. The most important parameter is the operation latency and only for read operations algorithm SFW shows a small difference. The CPU time consumption is always higher for algorithm SFW due to the predicate computation and is increased with increasing clients. On the other hand, algorithm SIMPLE's CPU time consumption is stable while increasing the number of clients. This experiment suggests that algorithm SFW would be preferred over SIMPLE in settings where the communication delay dominates the time needed for the predicate computation (e.g., when the delay exceeds 0.2 secs).

6.3 Conclusions

In this section our conclusions from the scenarios are summarized. The first experiment provided indications for the number of readers and writers to use in further experiments. With very high numbers of readers and writers (specifically 80 readers, 80 writers and 20 servers) the system configuration (over PlanetLab) seems to reach a limit in which a lot of processes crash. Thus, it is important to choose carefully the number of servers and clients in order to avoid overwhelming the system.

Regarding the operation interval, it does not affect the efficiency of the system considerably. Consequently, it is appropriate to keep a low value in order for the execution to take a reasonable amount of time.

Will more scenarios with a larger amount of object replicas (servers) possibly show a larger effect of the quorum intersection degree (n)? Increasing the total number of servers to allow more servers to fail ("more" robustness), while maintaining desirable efficiency, do not change the fact that the maximum number of server failures "allowed" is a small percentage over the total number of servers.

Using a large number of PlanetLab nodes for object replicas and expecting almost all of them to be responsive is impractical. In practise the latency of operations further increases as reader/writers must wait for responses from a larger set of servers and the possibility that some of them might perform arbitrarily slow is high. Also, increasing the number of replicas may not be practical since it also increases the economical (total cost of ownership) aspect of maintaining the quorum system. The results indicate that n is an important parameter to the overall efficiency of algorithm SFW and in a real application setup it should be optimized according to the

application expected behaviour in order to obtain efficient operation latency and small percentage of failures.

Under reasonable conditions algorithm SFW can be efficient in comparison with algorithm SIMPLE. That is in the case when the number of reader and writers in the system is low. An indication for the number of writers is to be about the same as the number of servers (replicas) without restrictions on the number of the readers (besides the systems' load capacity). Algorithm SFW may not provide a substantial improvement on the operation latency when compared to SIMPLE, but it still reduces the communication overhead on the network links. The CPU time consumption is only a small issue for algorithm SFW because the processing of the predicate is performed on the client's side and thus server performance is not affected. Although algorithm SFW saves network bandwidth it does so by increasing CPU time consumption. If the predicate for any reason decides that a second communication round is needed then the CPU time spent is clearly an overhead. Thus it is important that the method used in the implementation to calculate the predicate be optimal. On the other hand, the large percentage of fast read and write operations suggest that algorithm SFW would perform much better than algorithm SIMPLE in settings where the communication delay dominates the time needed for the predicate computation.

Chapter 7

Epilogue

A lot of research has been conducted for studying efficient data survivability in distributed storage systems. Considering Multiple Writers and Multiple Readers (MWMMR) implementations where the atomic data object is replicated on a set of servers susceptible to failures. Researchers have attempted to answer the question of how efficient can a read/write operation be. Recent work introduced algorithm SFW which is the first algorithm in the MWMMR model to allow fast read and write operations.

In this thesis the practicality of algorithm SFW under real network conditions provided by PlanetLab is examined. Algorithm SFW uses read and write predicates to decide if a second communication round is needed by the read or write operation, respectively. These predicates try to discover the distribution of the tag in a large solution space. For this reason a heuristic method is proposed that reduces the solution space. An empirical evaluation of algorithm SFW is performed on PlanetLab using as metrics the percentage of fast operations, the operation latency, the quorum intersection degree, the CPU consumption and the percentage of failures from timeouts. The results are compared with a robust, reliable algorithm (SIMPLE) that always performs slow operations. Careful design is needed if implementations are to maximize the efficiency of algorithm SFW, although the restrictions imposed on the quorum system due to high intersection degree may be too much for most application specifications.

Overall, the algorithm mostly behaves as expected in the experiments we conducted. An essential decision that needs to be made when building a system

using algorithm SFW is the number of servers and clients to use. It is imperative to choose the appropriate number of servers for the wanted number of clients to avoid overwhelming the system. According to the experiments the operation interval does not notably affect the efficiency of the algorithm.

When the quorum intersection degree increases, the percentage of fast write operations and the operations latency increase as well. When a high percentage of fast operations exists then the operations latency is balanced off. The percentage of fast read operations remains generally high for all experimental values of quorum intersection degree.

Algorithm SFW can be efficient in comparison with algorithm SIMPLE but only by a small margin. That is in the case when the number of reader and writers in the system is low compared to the servers. Thus, the number of writers should approach the number of servers in the system. The minor improvement that SFW provides on the operation latency is disappointing. Regardless, considering the percentage of fast operations allowed by algorithm SFW in comparison to no fast operations offered by algorithm SIMPLE, it reduces the communication overhead on the network links. Although algorithm SFW saves network bandwidth it does so by increasing CPU time consumption used for the predicate computation. If the validation of the predicate procedure decides that a second communication round is needed, then the increased CPU time is an overhead compared to SIMPLE. It is critical that the technique chosen to implement the predicate validation be optimum.

A proof is still necessary for the accuracy of the proposed heuristic method and its efficiency. Additionally, an optimization of the algorithms' implementation can be done to reduce communication time by exploring the benefits of CoDNS [29] (reliable DNS on PlanetLab) to reduce host name address resolution. Reducing the communication delay can benefit both algorithms but it may give advantage to

SIMPLE, since algorithm SFW seems to perform better in settings where the communication delay overshadows the predicate computation.

Dynamic calculation of the operations timeout could be developed similar to the concept of TCP Vegas [30] congestion control algorithm, in which timeouts are set and round-trip delays are measured for every packet in the transmit buffer and additive increases in the congestion window are made dynamically. The concept is to dynamically calculate the timeout of an operation as a function of previous communication round-trips between servers. As a result the timeout can be dynamically readjusted to reflect both network congestion and client awareness of the servers performing arbitrarily slow. Finally, it remains to investigate the efficiency of algorithm SFW when the atomic object size increases (e.g. files of varying sizes are used). Despite the rather disappointing results regarding write operation latency in these experiments, it is expected that the efficiency of algorithm SFW will be clearer when files of varying sizes are used; in algorithm SIMPLE the writers send the file twice while algorithm SFW sometimes sends it only once.

Bibliography

- 1 Patterson, David A., Gibson, Garth, and Katz, H. Randy. A case of Redundant Arrays of Inexpensive Disks (RAID). *International Conference on Management of Data, Proceedings of the 1988 ACM SIGMOD international conference on Management of data* (1988), 109 - 116.
- 2 Herlihy, Maurice P. and Wing, Jeannette M. Axioms for Concurrent Objects. *Annual Symposium on Principles of Programming Languages Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1987), 13-26.
- 3 Lamport, Leslie. On Interprocess Communication. *Distributed Computing*, 1 (1985).
- 4 Herlihy, Maurice M. and Wing, Jannette. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12, 3 (1990), 463-492.
- 5 Bernstein, Philip A., Hadzilacos, Vassos, and Goodman, Nathan. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.
- 6 Weikum, Gerhard and Vossen, Gottfried. *Transactional Information Systems*. Elsevier, 2001.
- 7 Attiya, H., Bar-Noy, A., and Dolev, D. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42, 1 (Jan 1995), 124-142.
- 8 Lynch, Nancy A. and Shavartsman, Alexander A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *In Proceedings of Symposium on Fault-Tolerant Computing* (1997), 272-281.
- 9 Lynch, N. and Shvartsman, A. A. RAMBO: A reconfigurable atomic memory

- service for dynamic networks. *In Proceedings of 16th International Symposium on Distributed Computing (DISC)* (2002), 173-190.
- 10 Dutta, Partha, Guerraoui, Rachid, Levy, Ron R., and Chakraborty, Arindam. How fast can a Distributed Atomic Read Be? *In Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), 236-245.
 - 11 Georgiou, Chryssis, Nicolaou, Nicolas, and Shvartsman, Alexander A. Fault-tolerant semifast implementations for atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69, 1 (2009), 62-79. Preliminary version appeared in SPAA 2006.
 - 12 Englert, Burkhard, Georgiou, Chryssis, Musial, Peter M., Nicolaou, Nicolas, and Shvartsman, Alexander A. On the Efficiency of Atomic Multi-Reader, Multi-Writer Distributed Memory. *Proc. of the 13th International Conference on Principles of Distributed Systems (OPODIS 2009)* (2009), 240-254. Also as Technical Report at the University of Cyprus.
 - 13 Georgiou, Chryssis, Nicolaou, Nicolas C., and Shvartsman, Alexander A. On the Robustness of (Semi)Fast Quorum-Based Implementations of Atomic Shared Memory. *Proc. of the 22nd International Symposium on Distributed Computing (DISC 2008)* (May 2008), 289-304.
 - 14 Spring, Neil, Peterson, Larry, Bavier, Andy, and Vivek, Pai. Using PlanetLab for Network Research: Myths, Realities and Practices. *ACM SIGOPS Operating Systems Review*, 40, 1 (January 2006), 17-24.
 - 15 Peterson, L. L., Bavier, A. C., Fiuczynski, M. E., and Muir, S. Experiences building planetlab. *OSDI* (2006), 351-366.
 - 16 Lamport, Leslie. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, 7 (July 1978), 558-565.

- 17 Georgiou, Chryssis. Robust Implementations of Atomic Read/Write Objects in Message-Passing Systems. *Computer Science Seminar Presentation, University of Cyprus* (March 23, 2010).
- 18 Malkhi, D. and Reiter, M. Byzantine Quorum Systems. *Distributed Computing*, 11, 4 (1998), 203-213.
- 19 Peleg, D. and Wool, A. Crumbling walls: A class of high availability quorum system. In *Proceedings of 14th ACM Symposium on Principles of Distributed Computing (PODC)* (1995), 120-129.
- 20 Seth, Gilbert and Grzegorz, Malewicz. The Quorum Deployment Problem. *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS)* (December 2004).
- 21 Chockler, Gregory, Gilbert, Seth, Gramoli, Vincent, Peter, Musial M., and Shvartsman, Alexander A. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69, 1 (January 2009), 100-116.
- 22 Attiya, H, Chaudhuri, S, Friedman, R, and Welch, J.L. Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies. *SIAM Journal on Computing*, 27, 1 (February 1998), 65-89.
- 23 Fan, Rui and Lynch, Nancy. Efficient Replication of Large Data Objects. *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing* (July 2003), 335.
- 24 NS-2 Wiki Available at: http://nslam.isi.edu/nslam/index.php/Main_Page.
- 25 Bavier, A., Muir, S., Peterson, L. et al. Operating system support for planetary-scale network services. *Symposium on Networked Systems Design and*

- Implementation (NSDI '04)* (May 2004).
- 26 Lamport, Leslie, Shostak, Robert, and Pease, Marshall. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4, 3 (July 1982), 382-401.
- 27 Fisher, Michael J., Lynch, Nancy A., and Paterson, Michael S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32, 2 (April 1985), 374-382.
- 28 PlanetLab Home Page Available at: <http://www.planet-lab.org>.
- 29 KyoungSoo, Park, Vivek, Pai S., Peterson, Larry, and Zhe, Wang. CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups. *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation(OSDI '04)* (2004).
- 30 Low, Steven, Peterson, Larry, and Wang, Limin. Understanding TCP Vegas: A Duality Model. *Technical Report TR-616-00* (November 2000).
- 31 Garcia-Molina, Hector and Barbara, Daniel. How to assign votes in a distributed system. *Journal of the ACM*, 32, 4 (October 1985), 841-860.
- 32 Englert, Burkhard, Georgiou, Chryssis, Musial, M. Peter, Nicolaou, Nicolas, and Shvartsman, Alexander A. On the Efficiency of Atomic Multi-Reader Multi-Writer Distributed Memory. Technical Report, University Of Cyprus. *in Proc. of the 13th International Conference on Principles of Distributed Systems (OPODIS 2009)* (Nimes, France, 2009), 240-254.

Appendix A

The format of the configuration file `confic.ini` is shown Table 17: example of configuration file: `confic.ini`, with included comments that explain each field.

```
#IMPORTANT! AS A GENERAL RULE DO NOT CHANGE THE ORDER OF VARIABLE
#DEFINITION IN THIS FILE!!

#casing(upper/lower) of variables in this file does not matter
port=4709
serverNum=20
quorumNum=1140
readerNum=83
writerNum=83
alwaysSentValue=0
#for each server we define the IP address and the port
#maybe a port has been reserved at some point and all our servers crash
#.. this way only one server goes down due to port which we can accept

#freedom.informatik.rwth-aachen.de
serverAddr=137.226.138.154
serverPort=4709
#chronos.disy.inf.uni-konstanz.de
serverAddr=134.34.246.5
serverPort=4710
#dannan.disy.inf.uni-konstanz.de
serverAddr=134.34.246.4
serverPort=4711
#host2.planetlab.informatik.tu-darmstadt.de
serverAddr=130.83.166.199
serverPort=4708
#adrastea.mcs.suffolk.edu
serverAddr=192.138.213.236
serverPort=4713
#75-130-96-12.static.oxfr.ma.charter.com
serverAddr=75.130.96.12
serverPort=4714
#pl1.grid.kiae.ru
serverAddr=144.206.66.56
serverPort=4715
#75-130-96-13.static.oxfr.ma.charter.com
serverAddr=75.130.96.13
serverPort=4716
#host3.planetlab.informatik.tu-darmstadt.de
serverAddr=130.83.166.200
serverPort=4717
#node-1.mcgillplanetlab.org
serverAddr=192.197.121.2
serverPort=4718
#flow.colgate.edu
```

```

serverAddr=149.43.80.22
serverPort=4719
#jupiter.cs.brown.edu
serverAddr=198.7.242.41
serverPort=4720
#ebb.colgate.edu
serverAddr=149.43.80.20
serverPort=4721
#pl1.ucsf.indiana.edu
serverAddr=156.56.250.226
serverPort=4722
#pl2.planet.cs.kent.edu
serverAddr=131.123.34.36
serverPort=4723
#planetlab04.cs.washington.edu
serverAddr=128.208.4.99
serverPort=4724
#fobos.cecalc.ula.ve
serverAddr=150.189.2.101
serverPort=4725
#ds-pl1.technion.ac.il
serverAddr=132.68.237.34
serverPort=4726
#cs-planetlab4.cs.surrey.sfu.ca
serverAddr=206.12.16.155
serverPort=4727
#pl1.rcc.uottawa.ca
serverAddr=216.48.80.12
serverPort=4728

#####
# quorum
#votingMethod = 0, majority voting
#      1, majority - x
#      2, Grid type dynamically generated quorums
#quorum = list of servers in the quorum according with
#the order given in their definition(ip,port) above starting from 0
#note that if method=0 quorums will be ignored
#
#also note that quorum get auto assigned id starting from 0
#####
quorumFileName=../majorities_x.dat
votingMethod=1
#quorum=0,1
#quorum=0,2
#quorum=1,2
#####
#algorithm execution modifiers
#####
#scenarioNum, number of scenarios, according to the number of scenarios or algorithm execution
modifiers are redefined
#executionNum, how many times to execute the algorithm
#setting, 0 = stochastic(intervals can be randomly any of [ OpTime .. Interval ])

```

```

#           note that operationTime(Optime) is the average time between a node and a server.
#           and also that Interval can be any of read/write/fail
#           1 = fixed
#readNum, number of reader operations to execute
#readInterval & writeInterval, time between requests in millisecs
#will randomly try to fail every failInterval with failureProbability
#note that if is zero then a failure check for every server reply is performed.
#failProbability, fail percentage from 100%
scenarioNum=3

#scenario 1
executionNum=5
setting=1
readNum=200
writeNum=200
readInterval=5000000
writeInterval=5000000
startWait=200000
failInterval=60
failProbability=1000000
#scenario 2
executionNum=5
setting=1
readNum=200
writeNum=200
readInterval=430000
writeInterval=430000
startWait=200000
failInterval=60
failProbability=1000000
#scenario 3
executionNum=5
setting=1
readNum=200
writeNum=200
readInterval=430000
writeInterval=230000
startWait=200000
failInterval=60
failProbability=1000000
#no /n at end of file please

```

Table 17: example of configuration file: config.ini

Appendix B

```
#!/bin/bash
time=20m
totalServers=$1
fails=$2
totalReaders=$3
totalWriters=$4
alg=$5 #O SIMPLE, 1 SFW
type=$6 #0 INT, 1 FILE
testEcho=$7
#if is 1 start server only
#Else only reader and writers
startServerParam=$8

apath='/home/cyprus_ATOMIC/PlanetLab'

echo totalServers=$totalServers fails=$fails totalReaders=$totalReaders
totalWriters=$totalWriters alg=$alg type=$type testEcho=$testEcho
startServerParam=$startServerParam

counter=0
lines=0
ReadersSoFar=0
WritersSoFar=0
#--error-limit=no
valgrindParams='--tool=memcheck --leak-check=yes --show-reachable=yes -v'

> scripttest.ini #OVERWRITE TO NOTHING

s=serverexe
r=readerexe
w=writerexe
m=memcheck-x86-li

if [ $testEcho == 0 ]; then
    #chmod +x ../quorum_gen.exe
    #../quorum_gen.exe -d $totalServers $fails 832916
    chmod +x $apath/reader/readerexe
    chmod +x $apath/writer/writerexe
    chmod +x $apath/server/serverexe
    #the following lines kill any running process
    #ps axco pid,command | grep $s | awk '{ print "$s"; }' | xargs kill -9 &
    #ps axco pid,command | grep $m | sed
    's/^[:space:]]*\(.*)[:space:]]*$/\1/' | cut -d" " -f1 | xargs kill -9 &
    if [ $startServerParam == 1 ]; then
        ps axco pid,command | grep $s | sed
        's/^[:space:]]*\(.*)[:space:]]*$/\1/' | cut -d" " -f1 | xargs kill -9 &
        ps axco pid,command | grep $w | sed
        's/^[:space:]]*\(.*)[:space:]]*$/\1/' | cut -d" " -f1 | xargs kill -9 &
        ps axco pid,command | grep $r | sed
```

```

's/^[:space:]]*\(.*)[:space:]]*$/\1/' | cut -d" " -f1 | xargs kill -9 &
    #wait to kill all previously running instances
    echo "wait to kill all previously running instances..."
    sleep 1
fi

#rm *.log &
#rm *.txt &
#rm *.result &
#rm *.predicate &
cd $apath/scenarios/int.simple.S5.R5.W5
ls * | grep -vE "[0-9]\.[0-9]\.data$" | grep -vE ".*\.dat$" | grep -vE
".*\sh$" | grep -vE ".*\.exe$" | grep -vE ".*\.ini$" | xargs rm -f &
    #wait to delete all previously generated .data files
    echo "wait to delete all previously generated .data files"
    sleep 1
fi

startServer()
{
    echo serverID $1 $HOSTNAME >> scripttest.ini
    if [ $testEcho == 0 ]; then
        echo serverID $1 $HOSTNAME
        cd $apath/scenarios/int.simple.S5.R5.W5
        $apath/server/serverexe $1 $type $alg &>
        $apath/scenarios/int.simple.S5.R5.W5/server$1.txt.log &
    fi
}

startReader() #Two parameters p1=number p2=readersSoFar
{
    rp1=$1
    rp2=$2
    fromReader=0
    toReader=0

    fromReader=$(( totalServers + rp2 ))
    toReader=$(( fromReader + rp1 ))
    echo READER P1=$rp1 P2=$rp2 fromReader=$fromReader
    toReader=$toReader >> scripttest.ini
    cd $apath/scenarios/int.simple.S5.R5.W5
    for ((ir=fromReader;ir<toReader;ir++))
    do
        echo readerID $ir $HOSTNAME >> scripttest.ini
        if [ $testEcho == 0 ]; then
            echo readerID $ir $HOSTNAME
            $apath/reader/readerexe $ir $type $alg &>
            $apath/scenarios/int.simple.S5.R5.W5/reader$ir.txt.log &
        fi
    done
}

startWriter() #Two parameters p1=number p2=numbersSoFar
{

```

```

wp1=$1
wp2=$2
fromWriter=0
toWriter=0

    fromWriter=$(( totalServers + totalReaders ))
    fromWriter=$(( fromWriter + wp2 ))
    toWriter=$(( fromWriter + wp1 ))
    echo WRITER P1=$wp1 P2=$wp2 fromWriter=$fromWriter
toWriter=$toWriter >> scripttest.ini
cd $apath/scenarios/int.simple.S5.R5.W5
for ((iw=fromWriter;iw<toWriter;iw++))
do
    echo writerID $iw $HOSTNAME >> scripttest.ini
    if [ $testEcho == 0 ]; then
        echo writerID $iw $HOSTNAME
        $apath/writer/writerexe $iw $type $alg &>
$apath/scenarios/int.simple.S5.R5.W5/writer$iw.txt.log &
        fi
    done
}

if [ $testEcho == 0 ]; then
    sleep 1
fi

for i in $(cat $apath/scenarios/servers.ini)
do
    ar=$(echo "$i" | tr -s ':' ' ')
    counter=0
    param1=0
    param2=0
    for k in $ar; do
        case "$counter" in
            0)
                host=$k
                ;;
            1)
                param1=$k
                ;;
            2)
                param2=$k
                ;;
        esac
        #counter=$(echo "$counter+1" | bc -lq)
        counter=$(( counter + 1 ))
    done
    if [ $host == $HOSTNAME ]; then
        echo $host == $HOSTNAME
        echo $host == $HOSTNAME >> scripttest.ini
        if [ $lines -lt $totalServers ]; then
            if [ $startServerParam == 1 ]; then
                startServer $lines
            fi
        fi
    fi
done

```

```

        fi
    fi;
    if [ $startServerParam == 0 ]; then
        startReader $param1 $ReadersSoFar
        sleep 1
        startWriter $param2 $WritersSoFar
    fi
fi
ReadersSoFar=$(( ReadersSoFar + param1 ))
#WritersSoFar=$(echo "$WritersSoFar+$param2"|bc -lq)
WritersSoFar=$(( WritersSoFar + param2 ))
#lines=$(echo "$lines+1"|bc -lq)
lines=$(( lines + 1 ))
done

#sleep $time
#termination="\nAttempting to terminate scenario after 20m...!!\n"
#echo -e $termination

#ps axco pid,command | grep $s | sed 's/^[[:space:]]*\. *\[[:space:]]*$\1/' |
#cut -d" " -f1 | xargs kill -9 &
#ps axco pid,command | grep $w | sed 's/^[[:space:]]*\. *\[[:space:]]*$\1/' |
#cut -d" " -f1 | xargs kill -9 &
#ps axco pid,command | grep $r | sed 's/^[[:space:]]*\. *\[[:space:]]*$\1/' |
#cut -d" " -f1 | xargs kill -9 &

#we cant use command wait here because server are propably up and running
#so give some time for kill command to act
#sleep 5

terminated="\nStarted Process Successfull!!"
echo -e $terminated

```


Appendix C

```
void* serve_thread( void* thread_args ) {
    int serve_threaderr=0;
    data_t * my_data;
    char startingWith[ FILENAME_SIZE ],tempBuf[ FILENAME_SIZE ];
    pck_t *recvMsg = NULL;
    int cm=-2,writeln=0,processIndex=0,len=0,objId = sys_conf.objId;

    writeLog("Starting serve_thread");
    my_data = (data_t*) thread_args;
    //Read request Message from newSocket
    recvMsg = (pck_t*) create_message( -1 , sys_conf.objType, sys_conf.algType
); //init a temp msg
    if(recvMsg!=NULL)recvMsg->ptd = &(threads[ my_data->index ]);
    objId=recvMsg->objId;
    //delete any inprogress file we have from this writer so we can receive the new one
    if(sys_conf.algType == SFW && sys_conf.objType==FILE_TYPE){
        len = sprintf( startingWith,"INPROGRESS.%d.%d", recvMsg->pid,
sys_conf.id );
        writeLog("going to remove files startingWith (%s)",startingWith);
    }
    //remove_files( startingWith, len );
    //recv the value(FILE|INT) asap

    if( recvReq( my_data->newSckt, recvMsg, &serve_threaderr ) == 1
){ //==1,check if need to recv val also
        processIndex = recvMsg->pid - sys_conf.serverNum;
        //get message value
        cm = compareTag_s( recvMsg->pid, recvMsg->typ, &( recvMsg->tag ), &(
state[ recvMsg->objId ].tag ) );
        if( cm > 0 )writeLog("msgTag>tag");
        writeLog("( %d )>=( %d ) -- recvMsg->cnt[ ] >= state[ recvMsg->objId ].cnt[
processIndex ]",recvMsg->cnt, state[ recvMsg->objId ].cnt[ processIndex ]);
        if( recvMsg->cnt >= state[ recvMsg->objId ].cnt[ processIndex ]){
            if( sys_conf.algType == SIMPLE && (recvMsg->typ == INFO) ){
                //readers and writers always sent value on 2nd round communication(INFO) for the
simple algorithm
                if( cm > 0 ){
                    writeln = 1;
                    recvMsgVal( my_data->newSckt, recvMsg, writeln, FALSE );
                }
            }else if( sys_conf.algType == SFW ){ //handle SFW FILE type values
                //cm = compareTag_s( recvMsg->pid, recvMsg->typ, &(recvMsg->tag), &( state[
recvMsg->objId ].tag ) );

                if( sys_conf.objType == FILE_TYPE ){

                    bzero(tempBuf, FILENAME_SIZE );

                    //if( ( ( char * ) recvMsg->val ) != NULL ) sprintf( tempBuf, "%s", ( ( char * )
```

```

recvMsg->val) );
    //else{
        //recvMsg->val = malloc( FILENAME_SIZE * sizeof( char ) );
        //if( recvMsg->val == NULL ){ writeLog( "malloc return NULL" );exit(-1); }
    //}

    if( recvMsg->typ == WRITE ){ //save the FILE VALUE AS INPROGRESS
        int writerIndex = recvMsg->pid - sys_conf.serverNum -
sys_conf.readerNum;
        writelt=1;
        snprintf(          recvMsg->val,          FILENAME_SIZE,
"INPROGRESS.%d.%d.%d.(%d.%d.%d).data",
recvMsg->pid,sys_conf.id,recvMsg->objId,recvMsg->tag.ts    +
1,recvMsg->tag.wid,recvMsg->tag.wc );
        writeLog("WRITE recvMsgVal %s", recvMsg->val );
        //add new tag+value in the inprogress set(this is also done in the process
function)
        state[ recvMsg->objId ].inprogress[ writerIndex ].tag.ts =
recvMsg->tag.ts;
        state[ recvMsg->objId ].inprogress[ writerIndex ].tag.wid =
recvMsg->tag.wid;
        state[ recvMsg->objId ].inprogress[ writerIndex ].tag.wc =
recvMsg->tag.wc;
        recvMsgVal( my_data->newSckt, recvMsg, writelt, TRUE );

    }else if( recvMsg->typ == INFO || recvMsg->typ == READ ){
        //writelt=1;
        if( cm > 0 ){
            writelt = 1;
            snprintf(  recvMsg->val,  FILENAME_SIZE,  "%d.%d.data",
sys_conf.id, recvMsg->objId );
        }else writelt = 0;

        writeLog("READ recvMsgVal %s, writelt=%d", recvMsg->val,
writelt );
        recvMsgVal( my_data->newSckt, recvMsg, writelt, FALSE );
    }
    snprintf( recvMsg->val, FILENAME_SIZE, "%s", tempBuf );
    //means that msg->tag > state->tag, update value
}
}
}
}
//else if( recvMsg->typ == INFO ){
//shutdown the connection so no further reading from server on sck
//shutdown( my_data->newSckt, SHUT_RD );
//}
//process buf and output msg
/*, &state, servers, quorumSystem, &sys_conf, &sendMsg*/
//always update the count var that indicates the message freshness
//if (state[ recvMsg->objId].fail == FALSE && recvMsg->cnt >= state[ recvMsg->objId ].cnt[
recvMsg->pid ]){
    // state[recvMsg->objId].cnt[recvMsg->pid] = recvMsg->cnt;
//}

```

```

/*
if( pthread_mutex_lock(&state_mutex) != 0 ){
    printf("unable to lock state_mutex");
}
*/

//OPTIMIZATION?? maybe only WRITE request should lock
if( pthread_spin_trylock(&process_lock)!=0){
    printf("unable to lock state_mutex");
}
process(recvMsg);
if( pthread_spin_unlock(&process_lock)!=0){
    printf("unable to lock state_mutex");
}
/*
if(pthread_mutex_unlock(&state_mutex) != 0){
    printf("unable to unlock log_file_mutex");
}*/

//Send a response
sendRes( my_data->newSckt, recvMsg /*, &state, &sendMsg*/ );
free(recvMsg);
//==>sleep(1);
shutdown( my_data->newSckt, SHUT_RDWR );
close( my_data->newSckt );
writeLog( "close sckt:%d", my_data->newSckt );
/--?free( recvMsg );
//sleep(3);
writeLog("Closing serve_thread");
return( NULL );
}

```

Appendix D

Writer.c: WriteObject function

```
//returns intValue as integer value written
void writeObject( msg_t msgType, int objectId, state_t*objectState, int*
intVal, obj_t objType, alg_t algType) {
    data_t* quorum_data=NULL; //threads data
    int i=0, quorum_size=0, cnter = 0,m=0;
    pck_t* pckToSent=NULL;
    bool_t isComplete=FALSE;
    char*createFileName;
    pckToSent = ( pck_t * ) create_message( msgType, objType, algType );
    //////////////////////////////////////
    //effect of write
    if( objectState->fail == FALSE && objectState->status == IDLE ) {
        //status<--active
        objectState->status = ACTIVE;
        //phase<--W
        objectState->phase = WRITE;
        //opc <--opc + 1, also used by reader for
        //counting ops
        objectState->opCnt++;
        //the write operation counter
        objectState->tag.wc++;
        //pCount <-- pCount + 1
        //value<--v
        stateToMessage( pckToSent, objectState );
        if( sys_conf.objType == INT_TYPE ){
            pckToSent->ival = randomVal;
            pckToSent->ipval = objectState->tag.ival;
        }else{
            //no need to do anything here filename does not change.
            //init msg
            if( sys_conf.objType == FILE_TYPE ){
                createFileName = create_file( objectId, randomVal );
                bzero(pckToSent->val, FILENAME_SIZE);
                snprintf( pckToSent->val, FILENAME_SIZE, "%s", createFileName );
                free( createFileName );
            }else if( sys_conf.objType == INT_TYPE ){
                pckToSent->ival = randomVal;
                pckToSent->ipval = objectState->tag.ival;
            }
            write_cnt++;
            pckToSent->cnt = write_cnt;
        }
    }
    //2. send to all servers
    //communicate(WRITE,threads_data);
    quorum_data = communicate( pckToSent, quorum_data, &quorum_size,
&isComplete );
```

```

if( isComplete == FALSE ){
    for( i = 0 ; i < quorum_size; i++ )
        if( quorum_data[ i ].srvAck == FALSE )
            cnter++;
    writeLog( "1st Round Communicate FAILED ( servers Timeout/Offline = %d
)", cnter );
    writeResult("1 %d %d %d",
        objectState->tag.ts,
        objectState->tag.wid,
        objectState->tag.wc
    );
    objectState->status = IDLE;
    free( pckToSent );
    return;
}
//3. process buf and output msg
if( process(objectState, quorum_data, quorum_size ) == TRUE ){
    //objectState->ipval = objectState->tag.ipval = objectState->tag.ival;
    //objectState->ival = objectState->tag.ival = randomVal;
    if(sys_conf.objType==INT_TYPE)
        *intVal = objectState->ival;
    if (objType == INT_TYPE)//to do check wid
        writeLog("object WRITE: tag(%d,%d,%d)v: %d", objectState->tag.ts,
objectState->tag.wid, objectState->tag.wc,objectState->tag.ival);
    else
        writeLog("object WRITE: tag(%d,%d,%d)", objectState->tag.ts,
objectState->tag.wid, objectState->tag.wc);
    }else writeLog("WRITE:process FAILED");
    free(pckToSent);
    objectState->status = IDLE;
    //////////////////////////////////////
}

```

Appendix E

Reader.c: readObject function:

```
void readObject(msg_t msgType, int objectId, state_t*objectState, int* intVal,
obj_t objType, alg_t algType) {
    data_t* quorum_data = NULL; //threads data
    int i, quorum_size, cnter = 0, m = 0;
    pck_t* pckToSend;
    bool_t isComplete = FALSE;
    int serverResCount=0;
    pckToSend = (pck_t *) create_message(msgType, objType, algType);
    sys_conf.objId = objectId;
    //effect of read
    if (objectState->fail == FALSE && objectState->status == IDLE) {
        objectState->phase = READ;
        objectState->status = ACTIVE;
        objectState->opCnt++; //read operations counter
    }
    //2. send to all servers READ request and
    //get received acks tags in quorum_data,
    stateToMessage(pckToSend, objectState);
    quorum_data = communicate(pckToSend, quorum_data, &quorum_size,
&isComplete);
    if( isComplete == FALSE ){
        for (i = 0; i < quorum_size; i++) {
            if (quorum_data[ i ].srvAck == FALSE) {
                cnter++;
            }
        }
        writeLog("1st Round Communicate FAILED ( servers Timeout/Offline =
%d)", cnter );
        writeResult("1 %d %d %d",
            objectState->tag.ts,
            objectState->tag.wid,
            objectState->tag.wc
        );
        objectState->status = IDLE;
        free(pckToSend);
        return;
    }
    process(objectState, quorum_data, quorum_size);
    //3. process quorum_data
    //select valid value and received it
    if(sys_conf.objType==INT_TYPE)
        *intVal = objectState->ival;
    if (objType == INT_TYPE)
        writeLog("object READ:tag(%d,%d,%d) v:%d", objectState->tag.ts,
objectState->tag.wid, objectState->tag.wc,objectState->ival);
    else
        writeLog("object READ:tag(%d,%d,%d)", objectState->tag.ts, objectState-
>tag.wid, objectState->tag.wc);
}
```

```
free(pckToSent);  
objectState->status = IDLE;  
}
```

Appendix F

A list of all PlanetLab nodes used for client follows in Table 18.

Table 18: Client PlanetLab Nodes

ait05.us.es
aladdin.planetlab.extranet.uni-passau.de
deimos.cecalc.ula.ve
ds-pl3.technion.ac.il
dschinni.planetlab.extranet.uni-passau.de
orbpl1.rutgers.edu
146-179.surfsnel.dsl.internl.net
147-179.surfsnel.dsl.internl.net
cs-planetlab3.cs.surrey.sfu.ca
kc-sce-plab1.umkc.edu
lsirextpc01.epfl.ch
netapp7.cs.kookmin.ac.kr
node1.lbnl.nodes.planet-lab.org
node1.planetlab.albany.edu
nodeb.howard.edu
pl1.pku.edu.cn
plab-1.sinp.msu.ru
plab1-c703.uibk.ac.at
plab1.cs.ust.hk
plab2-itec.uni-klu.ac.at
plab2.cs.ust.hk

planetlab-1.cs.uh.edu:0:1
planetlab-1.imperial.ac.uk:0:1
planetlab-1.iscte.pt:0:1
planetlab-2.cs.auckland.ac.nz
planetlab-2.pdl.nudt.edu.cn
planetlab-4.EECS.CWRU.Edu
planetlab01.erin.utoronto.ca
planetlab01.sys.virginia.edu
planetlab03.cs.washington.edu
planetlab1.byu.edu
planetlab1.cs.purdue.edu
planetlab1.cs.uiuc.edu
planetlab1.csg.uzh.ch
planetlab1.eecs.wsu.edu
planetlab1.ifi.uio.no
planetlab1.informatik.uni-goettingen.de
planetlab1.jhu.edu
planetlab1.williams.edu
planetlab14.millennium.berkeley.edu
planetlab2.arizona-gigapop.net
planetlab2.cs.uoregon.edu
planetlab2.eecs.northwestern.edu
planetlab2.hiit.fi
planetlab2.itwm.fhg.de
planetlab2.sfc.wide.ad.jp

planetlab2.williams.edu
planetlab3.di.unito.it
planetlab3.singaren.net.sg
planetlab4.csres.utexas.edu
planetlab4.wail.wisc.edu
planetlab6.csres.utexas.edu
planetx.scs.cs.nyu.edu
pli1-pa-6.hpl.hp.com
plnode-03.gpolab.bbn.com
pnode1.pdcc-ntu.singaren.net.sg
ricepl-1.cs.rice.edu
ttu2-1.nodes.planet-lab.org
vicky.planetlab.ntua.gr
vn4.cse.wustl.edu

Appendix G

The script code that parses downloaded results and generates averages needed for the experiments graphs the SFW version follows in table [] (a similar script is used for the SIMPLE algorithm case).

```
#!/bin/bash

keyword="fail? ts wid wc fast? time(cpuTime) time(realTime)"
#
*****
**
# find_and_replace_in_files.sh
# This script does a recursive, case sensitive directory search and replace of files
# To make a case insensitive search replace, use the -i switch in the grep call
# uses a startdirectory parameter so that you can run it outside of specified directory - else this
script will modify itself!
#
*****
**

# ***** Change Variables Here *****
startdirectory=$1
searchterm="fail? ts wid wc fast? time(cpuTime) time(realTime)"
replaceterm=" "

if [[ $# -eq 0 ]]; then
    echo -e "Please provide a directory name in the current folder\n"
    exit
fi;

# *****

# Floating point number functions.

#####
#####
# Default scale used by float functions.

float_scale=6

#####
#####
# Evaluate a floating point number expression.

function float_eval()
{
    local stat=0
```

```

local result=0.0
if [[ $# -gt 0 ]]; then
    result=$(echo "scale=$float_scale; $" | bc -q 2>/dev/null)
    stat=$?
    if [[ $stat -eq 0 && -z "$result" ]]; then stat=1; fi
fi
echo $result
return $stat
}

#####
#####
# Evaluate a floating point number conditional expression.

function float_cond()
{
    local cond=0
    if [[ $# -gt 0 ]]; then
        cond=$(echo "$*" | bc -q 2>/dev/null)
        if [[ -z "$cond" ]]; then cond=0; fi
        if [[ "$cond" != 0 && "$cond" != 1 ]]; then cond=0; fi
    fi
    local stat=$((cond == 0))
    return $stat
}

echo "*****"
echo "* Search and Replace in Files Version .1 *"
echo "*****"
for file in $(grep -l -R $searchterm $startdirectory)
do
    sed -e "s/$searchterm/$replaceterm/ig" $file > /tmp/tempfile.tmp
    mv /tmp/tempfile.tmp $file
    echo "Modified: " $file
done
echo " *** Yay! All Done! *** "

operations=0
successfulOperations=0
fastOperations=0
column=0
percentageOfFast=0
totalExecTime=0
totalCpuTime=0
cat $1/* > r.txt
sed 's/$/ -1/' r.txt > results.txt
set -f
for i in $(cat results.txt)
do
    if [ $i == -1 ]; then
        column=$(( 0 - 1 ))

```

```

fi;
case "$column" in
0)
    operations=$(( operations + 1 ))
    isFail=$i
    if [ $isFail == 0 ]; then
        successfullOperations=$(( successfullOperations + 1 ))
    fi;
;;
1)
    ts=$i
;;
2)
    wid=$i
;;
3)
    wc=$i
;;
4)
    if [ $isFail == 0 ]; then
        isFast=$i
    fi;
    if [ $isFail == 1 ]; then
        isFast=2
    fi;
    if [ $isFast == 1 ]; then
        fastOperations=$(( fastOperations + 1 ))
    fi;
;;
5)
    if [ $isFail == 0 ]; then
        totalCpuTime=$(( float_eval "$totalCpuTime + $i" ))
    fi;
;;
6)
    if [ $isFail == 0 ]; then
        totalExecTime=$(( float_eval "$totalExecTime + $i" ))
    fi;
;;
esac
#counter=$(echo "$counter+1" | bc -lq)
column=$(( column + 1 ))
done
failedOperations=$(( operations - successfullOperations ))
avgCpuTime=$(( float_eval "$totalCpuTime / $successfullOperations" ))
avgExeTime=$(( float_eval "$totalExecTime / $successfullOperations" ))
percentageOfFast=$(( float_eval "$fastOperations / $successfullOperations * 100" ))
percentageOfFail=$(( float_eval "$failedOperations / $operations * 100" ))
echo -e "operations=$operations\n" >> summary_$1.txt
echo -e "successfullOperations=$successfullOperations\n" >>
summary_$1.txt
echo -e "failedOperations=$failedOperations\n" >> summary_$1.txt

```

```
echo -e "percentageOfFail=$percentageOfFail\n" >> summary_$1.txt
echo -e "fastOperations=$fastOperations\n" >> summary_$1.txt
echo -e "percentageOfFast=$percentageOfFast\n" >> summary_$1.txt
echo -e "totalCpuTime=$totalCpuTime\n" >> summary_$1.txt
echo -e "avgCpuTime=$avgCpuTime\n" >> summary_$1.txt
echo -e "totalExecTime=$totalExecTime\n" >> summary_$1.txt
echo -e "avgExeTime=$avgExeTime\n" >> summary_$1.txt

set +f
```