Evaluating Atomic MWMR Register Implementations on PlanetLab *

Chryssis Georgiou † Nicolas C. Nicolaou†

Abstract

Multiple-writer/multiple-reader (MWMR) read/write atomic register implementations provide precise consistency guarantees, in the asynchronous, crash-prone, message passing environment. Although there exists a number of theoretical solutions, their practicality is not typically investigated in a practical setting. The performance of atomic read/write register implementations is traditionally measured in terms of the latency of read and write operations due to (a) communication delays and (b) local computation. In this work we examine the practicality of three, theoretically-efficient, MWMR atomic register algorithms: Aprx-Sfw and CwFr from [10], and the generalization of the traditional algorithm of [3] in the MWMR environment, which we call Simple. To test the performance of the algorithms in a real-time environment we upload and run our implementations on the planetary network platform called PlanetLab [1]. Due to its simplistic nature, Simple requires two communication round-trips per read or write operation, but almost no local computation. The rest of the algorithms are (to this writing) the only to allow *single* round read and write operations but require *non-trivial* computation demands. Through our comparison we attempt to identify the trade-offs between the communication and the computation burdens for MWMR atomic register implementations in a real-time, adverse environment such as PlanetLab.

Technical Report TR-11-07
Department of Computer Science
University of Cyprus
September 2011

*This work is supported by the Cyprus Research Promotion Foundation's grant ΠΕΝΕΚ/0609/31 and the European Regional Development Fund.

†Department of Computer Science, University of Cyprus, Cyprus. Email: {chryssis,nicolasn}@cs.ucy.ac.cy.

1 Introduction

Emulating atomic registers in asynchronous, crash-prone, message-passing systems is one of the basic problems in distributed computing. In such settings the register is replicated among a set of replica hosts (or servers) to provide fault-tolerance and availability. Then read and write operations are implemented as communication protocols that ensure atomic consistency.

Efficiency of register implementations is normally measured in terms of the latency of read and write operations. Two factors affect operation latency: (a) computation, and (b) communication delays. An operation communicates with servers to read or write the register value. This involves at least a single communication round-trip, or *round*, i.e., messages from the invoking process to some servers and then the replies from these servers to the invoking process. Previous works focused on minimizing the number of rounds required by each operation. Dutta et al. [6] developed the first single-writer/multi-reader (SWMR) algorithm, where all operations complete in a single round. Such operations are called *fast*. The authors showed that fast operations are possible only if the number of readers in the system is constrained with respect to the number of servers. They also showed that it is impossible to have multi-writer/multi-reader (MWMR) implementations where *all* operations are fast. To remove the constraint on the number of readers, Georgiou et al. [15] introduced *semifast* implementations where at most one complete two-round read operation is allowed per write operation. They also showed that semifast MWMR implementations are impossible.

Algorithm Sfw, developed by Englert et al. [7], was the first to allow both reads and writes to be fast in the MWMR setting. The algorithm used quorum systems, sets of intersecting subsets of servers, to handle server failures. To decide whether an operation could terminate after its first round, the algorithm employed two *predicates*, one for the write and one for read operations.

A later work by Georgiou et al. [10] identified two weaknesses of algorithm Sfw with respect to its practicality: (1) the predicates used by the algorithm were NP-complete, and (2) fast operations were possible only when every *five* or more quorums had a non-empty intersection. To tackle these issues the authors introduced two new algorithms. The first algorithm, called Aprx-Sfw, proposed a polynomial log-approximation solution for the computation of the predicates in Sfw. This would allow faster computation of the predicates while potentially increasing the number of two round operations. However, algorithm Aprx-Sfw could not enable fast operations in any general quorum construction. For this reason, they presented algorithm CwFr that uses *Quorum Views* [14], client-side decision tools, to allow some fast *read* operations without additional constraints on the quorum system. Write operations in this implementation take two rounds.

Preliminary results on the operation latency of Aprx-Sfw and CwFr were gathered by Georgiou and Nicolaou in [13]. The authors simulated the two algorithms on the NS2 network simulator and their results suggested that the computation burden needed by the two algorithms was lower than the communication cost of a second communication round. The controlled variables of NS2 however cannot precisely describe the adverse network conditions that exist in a real-time environment. So although we may observe the trend of the algorithms we cannot make any claims on their practicality. In this work we try to evaluate the *practicality* of the aforementioned algorithms, by implementing and deploying them on a real-time planetary scale network platform, called PlanetLab [1].

Backround. Attiya et al. [3] developed a SWMR algorithm that achieves consistency by using intersecting majorities of servers in combination with $\langle timestamp, value\rangle$ value tags. A write operation increments the writer's local timestamp and delivers the new tag-value pair to a majority of servers, taking one round. A read operation obtains tag-value pairs from some majority, then propagates the pair corresponding to the highest timestamp to some majority of servers, thus taking two rounds.

The majority-based approach in [3] is readily generalized to quorum-based approaches in the MWMR setting (e.g., [19, 8, 18, 9, 16]). Such algorithms requires at least two communication rounds for each read and write operation. Both write and read operations query the servers for the latest value of the replica during the first round. In the second round the write operation generates a new tag and propagates the tag along with the new value to a quorum of servers. A read operation propagates to a quorum of servers the largest value it discovers during its first round. This algorithm is what we call Simple in the rest of this paper.

Dolev *et al.* [5] and Chockler *et al.* [4], provide MWMR implementations where some reads involve a single communication round when it is confirmed that the value read was already propagated to some quorum.

Dutta et al. [6] present the first *fast* atomic SWMR implementation where all operations take a *single* communication round. They show that fast behavior is achievable only when the number of reader processes R is inferior to $\frac{S}{t} - 2$, where S the number of servers, t of whom may crash. They also showed that fast MWMR implementations are impossible even in the presence of a single server failure. Georgiou et al. [15] introduced the notion of *virtual nodes* that enables an unbounded number of readers. They define the notion of *semifast* implementations where only a single read operation per write needs to be "slow" (take two rounds). They also show the impossibility of semifast MWMR implementations.

Georgiou et al. [14] showed that fast and semifast quorum-based SWMR implementations are possible if and only if a common intersection exists among all quorums. Hence a single point of failure exists in such solutions (i.e., any server in the common intersection), making such implementations not fault-tolerant. To trade efficiency for improved fault-tolerance, *weak-semifast* implementations in [14] require at least one single slow read per write operation, and where all writes are fast. To obtain a weak-semifast implementation they introduced a client-side decision tool called *Quorum Views* that enables fast read operations under read/write concurrency when *general quorum systems* are used.

Recently, Englert *et al.* [7] developed an atomic MWMR register implementation, called algorithm Sfw, that allows both reads and writes to complete in a *single round*. To handle server failures, their algorithm uses *n-wise quorum systems*: a set of subsets of servers, such that each n of these subsets intersect. The parameter n is called the *intersection degree* of the quorum system. The algorithm relies on $\langle tag, value \rangle$ pairs to totally order write operations. In contrast with traditional approaches, the algorithm uses the *server side ordering* (SSO) approach that transfers the responsibility of incrementing the tag from the writers to the servers. This way, the *query* round of write operations is eliminated. The authors proved that fast MWMR implementations are possible if and only if they allow not more than $n - 1$ successive write operations, where n is the intersection degree of the quorum system. If read operations are also allowed to modify the value of the register then from the provided bound it follows that a fast implementation can accommodate up to $n - 1$ readers and writers.

Contributions. Our goal is to provide empirical evidence on the practicality of MWMR atomic register implementations. We implement and deploy our algorithms on Planetlab, an overlay network infrastructure composed of machines that are located throughout the globe. In particular we implement and compare the following MWMR atomic register algorithms: Simple, Aprx-Sfw, and CwFr. Given the adverse and unpredictable conditions of the real-time system, we measure and compare the operation latency of the algorithms under two different families of service scenarios: (1) Variable number of readers/writers/servers, (2) Deployment of different quorum constructions. In our implementations communication was established via TCP/IP and the C/C++ programming language and sockets were used for interfacing with TCP/IP.

A general conclusion from our findings is that algorithms CwFr and Aprx-Sfw over-perform algorithm Simple in most of the scenarios. Also, computation costs do not have a great impact on the performance of read/write operations. This is partly due to the fact that in the tested real-time environment both CwFr and Aprx-Sfw may keep the percentage of slow operations under 20%. The intersection degree of the quorum system can be a decisive factor as it affects in a large degree the performance of Aprx-Sfw. Even though this is true, the average latency achieved by Aprx-Sfw in environments with small intersection degree is not much higher than the latency of the competition.

Paper organization. In Section 2 we briefly describe the model of computation that is assumed by the implemented algorithms. In Section 3 we provide a high level description of the algorithms we examine. In Section 4 we overview PlanetLab, we present our testbed we provide the scenarios we consider and briefly mention the implementation difficulties we encountered. Empirical results and comparisons of the algorithms are given in Section 5. We conclude in Section 6.

2 Model and Definitions

We consider the asynchronous message-passing model. There are three distinct finite sets of crash-prone processors: a set of readers $\mathcal{R}$, a set of writers $\mathcal{W}$, and a set of servers $\mathcal{S}$. The identifiers of all processors are unique and comparable. Communication among the processors is accomplished via reliable communication channels.

Servers and quorums. Servers are arranged into intersecting sets, or *quorums*, that together form a quorum system $\mathbb{Q}$. For a set of quorums $\mathcal{A} \subseteq \mathbb{Q}$ we denote the intersection of the quorums in $\mathcal{A}$ by $I_{\mathcal{A}} = \bigcap_{Q \in \mathcal{A}} Q$. A quorum system $\mathbb{Q}$ is called an *n-wise quorum system* if for any $\mathcal{A} \subseteq \mathbb{Q}$, s.t. $|\mathcal{A}| = n$ we have $I_{\mathcal{A}} \neq \emptyset$. We call n the *intersection degree* of $\mathbb{Q}$. Any quorum system is a *2-wise* (pairwise) quorum system because any two quorums intersect. At the other extreme, a $|\mathbb{Q}|$-*wise* quorum system has a common intersection among all quorums. From the definition it follows that an *n-wise* quorum system is also a *k-wise* quorum system, for $2 \leq k \leq n$.

Processes may fail by crashing. A process i is *faulty* in an execution if i crashes in the execution (once a process crashes, it does not recover); otherwise i is *correct*. A quorum $Q \in \mathbb{Q}$ is non-faulty if $\forall i \in Q$, i is correct; otherwise Q is faulty. We assume that at least one quorum in $\mathbb{Q}$ is non-faulty in any execution.

Atomicity. We study atomic read/write register implementations, where the register is replicated at servers. Reader p requests a read operation ρ on the register using action read_p. Similarly, a write operation is requested using action $\mathsf{write}(*)_p$ at writer p. The steps corresponding to such actions are called *invocation* steps. An operation terminates with the corresponding acknowledgment action; these steps are called *response* steps. An operation π is *incomplete* in an execution when the invocation step of π does not have the associated response step; otherwise π is *complete*. We assume that requests made by read and write processes are *well-formed*: a process does not request a new operation until it receives the response for a previously invoked operation.

In an execution, we say that an operation (read or write) π_1 *precedes* another operation π_2, or π_2 *succeeds* π_1, if the response step for π_1 precedes in real time the invocation step of π_2; this is denoted by $\pi_1 \rightarrow \pi_2$. Two operations are *concurrent* if neither precedes the other.

Correctness of an implementation of an atomic read/write object is defined in terms of the *atomicity* and *termination* properties. Assuming the failure model discussed earlier, the termination property requires that any operation invoked by a correct process eventually completes. Atomicity is defined as follows [17]. For any execution if all read and write operations that are invoked complete, then the operations can be partially ordered by an ordering $\prec$, so that the following properties are satisfied:

P1. The partial order is consistent with the external order of invocation and responses, that is, there do not exist operations π_1 and π_2, such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.

P2. All write operations are totally ordered and every read operation is ordered with respect to all the writes.

P3. Every read operation ordered after any writes returns the value of the last write preceding it in the partial order, and any read operation ordered before all writes returns the initial value of the register.

Efficiency and Fastness. We measure the efficiency of an atomic register implementation in terms of *computation* and *communication round-trips* (or simply rounds). A round is defined as follows [6, 15, 14]:

Definition 2.1 *Process p performs a communication round during operation π if all of the following hold: 1. p sends request messages that are a part of π to a set of processes, 2. any process q that receives a request message from p for operation π, replies without delay. 3. when process p receives enough replies it terminates the round (either completing π or starting new round).*

Operation π is *fast* [6] if it completes after its first communication round; an implementation is fast if in each execution all operations are fast. We use quorum systems and tags to maintain, and impose an ordering on, the values written to the register replicas. We say that a quorum $Q \in \mathbb{Q}$, *replies* to a process p for an operation π during a round, if $\forall s \in Q$, s receives a message during the round and replies to this message, and p receives all such replies.

3 Algorithm Description

Before proceeding to the description of our experiments we first present a high level description of the three algorithms we evaluate. To help the understanding of algorithm APRX-SFW we also describe algorithm SFW. We assume that the algorithms use quorum systems and follow the failure model presented in Section 2. Thus, termination is guaranteed if any read and write operation waits from the servers of a single quorum to reply. To order the written values the algorithms use (tag, value) pairs, where a tag contains a timestamp and the writer's identifier.

3.1 Algorithm SIMPLE

Algorithm Simple is a generalization of the algorithm developed by Attiya et al. [3] for the MWMR environment.

Server Protocol: Each replica receives read and write requests, and updates its local copy of the replica if the tag enclosed in the received message is greater than its local tag before replying with an acknowledgment and its local copy to the requester.

Write Protocol: The write operation performs two communication rounds. In the first round the writer sends query messages to all the servers and waits for a quorum of servers to reply. During the second round the writer performs the following three steps: (i) it discovers the pair with the maximum tag among the replies received in the first round, (ii) it generates a new tag by incrementing the timestamp inside the maximum discovered tag, and (iii) it propagates the new tag along with the value to be written to a quorum of servers.

Read Protocol: Similarly to the write operation every read operation performs two rounds to complete. The first round is identical as the first round of a write operation. During the second round the read operation performs the following two steps: (i) it discovers the pair with the maximum tag among the replies received in the first round, and (ii) it propagates the maximum tag-value pair to a quorum of servers.

3.2 Algorithm SFW

Algorithm SFW assumes that the servers are arranged in an *n-wise* quorum system. To enable fast writes the algorithm assigns partial responsibility to the servers for the ordering of the values written. Due to concurrency and asynchrony, however, two servers may receive messages originating from two different writers in different order. Thus, a read or write operation may witness different tags assigned to a single write operation. To deal with this problem, algorithm SFW uses two *predicates* to determine whether "enough" servers in the replying quorum assigned the same tag to a particular write operation.

Server Protocol: Servers wait for read and write requests. When a server receives a write request it generates a new tag, larger than any of the tags it witnessed, and assigns it to the value enclosed in the write message. The server records the generated tag, along with the write operation it was created for, in a set called *inprogress*. The set holds only a single tag (the latest generated by the server) for each writer.

Write Protocol: Each writer must communicate with a quorum of servers, say Q, during the first round of each write operation.At the end of the first round the writer evaluates a predicate to determine whether enough servers replied with the same tag. Let n be the intersection degree of the quorum system, and $\mathit{inprogress}_s(\omega)$ be the *inprogress* set that server s enclosed in the message it sent to the writer that invoked ω. The write predicate is:

PW: Writer predicate for a write ω**:** $\exists\, \tau, A, MS$ where: $\tau \in \{\langle ., \omega\rangle : \langle ., \omega\rangle \in \mathit{inprogress}_s(\omega) \wedge s \in Q\}$, $A \subseteq \mathbb{Q}, 0 \leq |A| \leq \frac{n}{2}-1$, and $MS = \{s : s \in Q \wedge \tau \in \mathit{inprogress}_s(\omega)\}$, s.t. either $|A| \neq 0$ and $I_A \cap Q \subseteq MS$ or $|A| = 0$ and $Q = MS$.

The predicate examines whether the same tag for the ongoing write operation is contained in the replies of all servers in the intersection among the replying quorum and $\frac{n}{2} - 1$ other quorums. Satisfaction of the predicate for a tag τ guarantees that any subsequent operation will also determine that the write operation is assigned tag τ. If the predicate **PW** holds then the write operation is fast. Otherwise the writer assigns the highest tag to the written value and proceeds to a second round to propagate the highest discovered tag to a quorum of servers.

Read Protocol: Read operations take one or two rounds. During its first round the read collects replies from a quorum of servers. Each of those servers reports a set of tags (one for each writer). The reader needs to decide which of those tags is assigned to the latest potentially completed write operation. For this purpose it uses a predicate similar to **PW**:

PR: Reader predicate for a read ρ**:** $\exists\, \tau, B, MS$, where: $\max(\tau) \in \bigcup_{s\in Q} \mathit{inprogress}_s(\rho)$, $B \subseteq \mathbb{Q}, 0 \leq |B| \leq \frac{n}{2} - 2$, and $MS = \{s : s \in Q \;\wedge\; \tau \in \mathit{inprogress}_s(\rho)\}$, s.t. either $|B| \neq 0$ and $I_B \cap Q \subseteq MS$ or $|B| = 0$ and $Q = MS$.

The predicate examines whether there is a tag for some write operation that is contained in the replies of all servers in the intersection among the replying quorum $\frac{n}{2} - 2$ other quorums. Satisfaction of the predicates for a tag τ assigned to some write operation, guarantees that any subsequent operation will also determine that the write operation is assigned tag τ. A read operations can be fast even if **PR** does not hold, but the read observed enough $confirmed$ tags with the same value. Confirmed tags are maintained in the servers and they indicate that either the write of the value with that tag is complete, or the tag was returned by some read operation.

The interested reader can see [7] for full details.

3.3 Algorithm Aprx-Sfw

The complexity of the predicates in Sfw raised the question whether they can be computed efficiently. In a recent work [10] (see also [11]) we have shown that both predicates are NP-Complete. To prove the NP-completeness of the predicates, we introduced a new combinatorial problem, called k-Set-Intersection, which captured both **PW** and **PR**. An approximate solution to the new problem could be obtained polynomially by using the approximation algorithm for the set cover. The steps of the approximation algorithm are:

By setting $U = \mathcal{S}$, M to contain all the servers that replied with a particular tag in the first round of a read or write operation, and k to be $\frac{n}{2} - 1$ for **PW** and $\frac{n}{2} - 2$ for **PR**, we obtain an approximate solution for Sfw. The new algorithm, called Aprx-Sfw, inherits the read, write, and serve protocols of Sfw and uses the above approximation algorithm for the evaluation of the **PW** and **PR** predicates. Aprx-Sfw promises to validate the predicates only when Sfw validates the predicates (preserving correctness), and

Given $(U, M, \mathbb{Q}, k)$:
Step 1: $\forall m \in M$
let $T_m = \{(U - M) - (Q_i - M) : m \in Q_i\}$
Step 2: Run SET-COVER greedy algorithm on
the instance $\{U - M, T_m, k\}$ for every $m \in M$:
Step 2a: Pick the set $R_i \in T_m$ with
the maximum uncovered elements
Step 2b: Take the union of every $R \in T_m$
picked in Step 2a (incl. R_i)
Step 2c: If the union equals $U - M$ go to Step 3;
else if there are more sets in T_m go to Step 2a
else repeat for another $m \in M$
Step 3: For any set $(U - M) - (Q_i - M)$ in the solution of set cover, add Q_i in the intersecting set.

Figure 1: Polynomial approximation algorithm for the k-SET-INTERSECTION.

yields a factor of $\log |\mathcal{S}|$ increase on the number of second communication rounds. This is a modest price to pay in exchange for substantial reduction in the computation overhead of algorithm SFW.

3.4 Algorithm CWFR

A second limitation of SFW is its reliance to specific constructions of quorums to enable fast read and write operations. Algorithm CWFR, presented in [10] (see also [12]), is designed to overcome this limitation, yet trying to allow single round read and write operations. While failing to enable single round writes, CWFR enables fast read operations by adopting the general idea of Quorum Views [14]. The algorithm employs two techniques:(i) the typical query and propagate approach (two rounds) for write operations, and (ii) analysis of Quorum Views [14] for potentially fast (single round) read operations.

Quorum Views are client side tools that, based on the distribution of a tag in a quorum, may determine the state of a write operation: completed or not. In particular, there are three different classes of quorum views. $qView(1)$ requires that all servers in some quorum reply with the same tag revealing that the write operation propagating this tag has potentially completed. $qView(3)$ requires that some servers in the quorum contain an older value, but there exists an intersection where all of its servers contain the new value. This creates uncertainty whether the write operation has completed in a neighboring quorum or not. Finally $qView(2)$ is the negation of the other two views and requires a quorum where the new value is neither distributed to the full quorum nor distributed fully in any of its intersections. This reveals that the write operation has certainly not completed.

Algorithm CWFR incorporates an iterative technique around quorum views that not only predicts the completion status of a write operation, but also detects the last potentially complete write operation. Below we provide a description of our algorithm and present the main idea behind our technique.

Write Protocol: The write protocol has two rounds. During the first round the writer discovers the maximum tag among the servers: it sends read messages to all servers and waits for replies from all members of some quorum. It then discovers the maximum tag among the replies and generates a new tag in which it encloses the incremented timestamp of the maximum tag, and the writer's identifier. In the second round, the writer associates the value to be written with the new tag, it propagates the pair to some quorum, and completes the write.

Read Protocol: The read protocol is more involved. The reader sends a read message to all servers and waits for some quorum to reply. Once a quorum replies, the reader determines $maxTag$. Then the reader analyzes the distribution of the tag within the responding quorum Q in an attempt to determine the latest, potentially complete, write operation. This is accomplished by determining the quorum view conditions. Detecting conditions of $qView(1)$ and $qView(3)$ are straightforward. When condition for $qView(1)$ is detected, the read completes and the value associated with the discovered $maxTag$ is returned. In the case of $qView(3)$ the reader continues to the second round, advertising the latest tag ($maxTag$) and its associated value. When a full quorum replies in the second round, the read returns the value associated with $maxTag$. Analysis of $qView(2)$ involves the discovery of the earliest completed write operation. This is done iteratively by (locally) removing the servers from Q that replied with the largest tags. After each iteration the reader determines the next largest tag in the remaining server set, and then re-examines the quorum views in the next iteration. This process eventually leads to either $qView(1)$ or $qView(3)$ being observed. If $qView(1)$ is observed, then the read completes in a single round by returning the value associated with the maximum tag among the servers that *remain* in Q. If $qView(3)$ is observed, then the reader proceeds to the second round as above, and upon completion it returns the value associated with the maximum tag $maxTag$ discovered among the original respondents in Q.

Server Protocol: The servers play a passive role. They receive read or write requests, update their object replica accordingly, and reply to the process that invoked the operation. Upon receipt of any message, the server compares its local tag with the tag included in the message. If the tag of the message is higher than its local tag, the server adopts the higher tag along with its corresponding value. Once this is done the server replies to the invoking process.

3.5 Algorithm Overview

Table 1 accumulates the theoretical communication and computation burdens of the four algorithms presented above. The name of the algorithm appears in the first column of the table. The second and third columns of the table shows how many rounds are required per write and read operation respectively. The next two columns present the computation required by each algorithm and the last column the technique the algorithm incorporates to decide on the values read/written on the atomic register.

7 **Algorithm**	**WR**	**RR**	**RC**	**WC**	**Decision Tool**
SIMPLE	2	2	$O(\lvert\mathcal{S}\rvert)$	$O(\lvert\mathcal{S}\rvert)$	Highest Tag
SFW	1 or 2	1 or 2	$O(2^{\lvert\mathbb{Q}\rvert-1})$	$O(2^{\lvert\mathbb{Q}\rvert-1})$	Predicates
APRX-SFW	1 or 2	1 or 2	$O(\lvert\mathcal{W}\rvert\lvert\mathcal{S}\rvert^2\lvert\mathbb{Q}\rvert)$	$O(\lvert\mathcal{S}\rvert^2\lvert\mathbb{Q}\rvert)$	Predicate Approximation
CWFR	2	1 or 2	$O(\lvert\mathcal{S}\rvert\lvert\mathbb{Q}\rvert)$	$O(\lvert\mathcal{S}\rvert)$	Quorum Views / Highest Tag

Table 1: Theoretical comparison of the four algorithms.

4 PlanetLab Implementation

In this section we present the construction details of our experiment. First we provide a description of the PlanetLab platform and then we present the parameters we considered and the scenarios we run for our implementations.

4.1 The PlanetLab Platform

PlanetLab is an overlay network infrastructure which is available as a testbed for computer networking and distributed systems research. As of September 2011, PlanetLab is composed of 1075 machines, provided by academic and industry institutions, at 525 locations worldwide. Malicious and buggy services can affect the communication infrastructure and the nodes performance; therefore, strict terms and conditions for providing security and stability in the PlanetLab are enforced. Nodes may be installed or rebooted at any time, turning their disk into a temporary form of storage, providing no guarantee regarding their reliability.

As oppose to NS2 simulator [2], there is almost no control over the components and execution sequence of the algorithms on PlanetLab, as it overheres the communication delays and congestion conditions of

Machine name	Country	# Cores	CPU Rate (Ghz)	RAM (GB)
nis-planet2.doshisha.ac.jp	Japan	2	2.9	4
chronos.disy.inf.uni-konstanz.de	Germany	2	2.4	4
planetlab2.cs.unc.edu	United States	4	2.6	4
peeramidion.irisa.fr	France	2	2.9	4
pl2.eng.monash.edu.au	Australia	4	2.6	4
planetlab2.ceid.upatras.gr	Greece	2	2.6	4
planetlab-01.bu.edu	United States	8	2.4	4
planetlab3.informatik.uni-erlangen.de	Germany	n/a	n/a	n/a
planetlab1.cs.uit.no	Norway	2	2.6	4
ple2.cesnet.cz	Czech Republic	4	2.8	8
planetlab01.tkn.tu-berlin.de	Germany	4	2.4	4
planetlab1.unineuchatel.ch	Switzerland	2	2.6	4
evghu14.colbud.hu	Hungary	n/a	n/a	n/a
zoi.di.uoa.gr	Greece	4	2.4	8
planetlab1.cs.vu.nl	Netherlands	8	2.3	4
pli1-pa-4.hpl.hp.com	United States	8	2.5	6
plab3.ple.silweb.pl	Poland	4	2.4	4
planetlab-2.imperial.ac.uk	United Kingdom	2	2.9	1
planet1.unipr.it	Italy	2	2.6	4
planetlab2.rd.tut.fi	Finland	8	2.4	16

Table 2: List of machines that hosted our experimet.

wide area networks (e.g. Internet cloud). Hence, some of the execution scenarios and environmental settings used for the simulation environment do not apply in this adverse and unpredictable, large-scale, real-time environment. Furthermore, since no global snapshot of the system may be acquired, the we relied on local decisions and readings of each individual participant in the system.

4.2 Experimentation Platform

Our test environment consists of a set of writers, readers, and servers. Communication between the nodes is established via TCP/IP. For our experiments we used the machines that appear in Table 2. Those were in total 20 machines. Our implementations were written in C++ programming language and C sockets were used for interfacing with TCP/IP.

Each of those machines was used to host one or more processes. The servers were hosted on the first 10 machines. If the number of servers exceeded the number of hosts then each host was running more then one server instance in a round robin fashion. For instance, if the number of servers were 15 then two servers were running in each of the first 5 machines, and one server instance on each of the remaining 5 machines. In other words the 1^{st} and 11^{th} servers were running on the first machine, the 2^{nd} and 12^{th} servers on the second machine and so on. Note that the choice of the first 10 machines was such as to split our servers throughout the world. The readers and the writers were started on all 20 machines starting from the 10^{th} machine and following a round robin technique. Thus, some of the first 10 machines could run a server and a client at the same time.

We have evaluated the algorithms with majority quorums. As discussed in [7], assuming $|\mathcal{S}|$ servers out of which f can crash, we can construct an $(\frac{|\mathcal{S}|}{f} - 1)$-wise quorum system $\mathbb{Q}$. Each quorum Q of $\mathbb{Q}$ has size $|Q| = |\mathcal{S}| - f$. The processes are not aware of f. The quorum system is generated *a priori* and is distributed to each machine as a file. So each participant can obtain the quorum construction be reading the appropriate file from the host machine.

We use the positive time parameters $rInt = 10sec$ and $wInt = 10sec$ to model operation frequency. Readers and writers pick a uniformly at random time between $[0 \ldots rInt]$ and $[0 \ldots wInt]$, respectively, to invoke their next read (resp. write) operation. Each reader and writer chooses a new timeout every time their latest operation is completed. This preserves the property of well-formedness discussed in Section 2.

Finally we specify the number of operations each participant should invoke. For our experiments we allow participants to perform up to 20 operations (this totals to 400-3000 operations).

4.3 Difficulties

Consistent storage protocols are not readily designed for the network infrastructure. For this reason our algorithms need to utilize the existing communication protocols like TCP/IP to achieve process communication. So we need to be especially careful that the use of these protocols does not affect the correctness of our algorithms. Below we present some hurdles we faced during the development of our implementations along with the workarounds we used to ensure the correctness of the algorithms.

Multiplexing. Client-Server architecture proved to be more challenging than expected for the implementations of consistent implementations. In traditional techniques the server listens for an incoming connection and spawns a child or a thread process to handle an incoming request from a client. By doing so, each client is being served independently from the rest of the clients, and thus does not have to wait for the other clients to terminate. At the same time each client gets the illusion that it communicates with a different server. This is acceptable if clients do not modify the local information of the server. It creates serious synchronization issues however when the two clients try to update the server's state.

To overcome this problem we chose not to use forking or threads to establish the communication between the server with the clients. Rather, we used *multiplexing*. This is a non-blocking technique were the server does not block to wait for incoming connections. Instead, the server allows the clients to connect to it and periodically checks of any new connection request. Once a new connection is established the server generates a new file descriptor and places the descriptor in a pool of connections. In a continuous loop the server checks if a client wants to transmit a message by checking the state of the descriptor associated with the connection of the particular client. When a message is received the server communicates with the sending process to satisfy its request. This is an explicit communication between the server and the client and circumvents any synchronization issues. On the other hand, the use of non-blocking communication protocols allows different clients to be connected at the same time with a particular server and wait their turn until they get served.

Resource Limits. Each slice on PlanetLab offers limited resources for each experiment. Furthermore the master machine that starts the experiments has limitations on the number of processes that can be initiated concurrently. For these reasons we bounded the number of readers and writes to 40. We still obtained four points by running our implementations with 10,20,30 and 40 read and write processes. We believe that this is an adequately large sample fro the extraction of consistent results.

Sampling. The PlanetLab environment is extremely adverse. Machines may go down at any time in the execution, the network latency may increase without notice and participant communication may be interrupted. Thus, to obtain some reasonable results we had to run our algorithms alternatively to maintain short time intervals between the algorithm's runs. Our first try was to run all the scenarios on a single algorithm and then move on the next algorithm. This technique produced large time intervals between the execution of two consecutive algorithms. The problem of that method was that the conditions (network delay, node failures etc.) of the network at the time of the run of one algorithm were changing dramatically by the time the next algorithm was running. Algorithm alternation proved to provide a solution to this problem. Even though we still observe network changes the plotting of the results provide less "noise" on the latency of the operations due to network delay differences.

Weighted Average Time. The last issue we faced during our development had to do with the hectic participation of the nodes in PlanetLab. Since the nodes in PlanetLab may be interrupted at any time during the execution of any algorithm, we observed that in most of the runs not all the readers were terminating. This is because the readers or writers hosted in a failed PlanetLab node were not able to complete all the necessary operations. As the operations completed by those failed readers and writers were not counted, getting the average of the read/write latency by dividing the total latency over the total number of terminated readers/writers per run did not produce a consistent result. So we decided to obtain the weighted average of the read/write latency by dividing the total operation latency over the total number of terminated processes. For example, assume a scenario were we initiated 20 readers out of which 10 terminated in the first run and 15 terminated in the second run. Lets assume for example that the total read latency of the first run was 100s and the total latency of the second run was 120s. With the non-weigthed method the average latency was

$$nonWeightedAvg = (\frac{100}{10} + \frac{120}{15})/2 = 9s$$

With the weigthed average we obtain the following:

$$WeightedAvg = \frac{100 + 120}{10 + 15} = 8s$$

The weighted average is the average latency we use for our plots.

4.4 Scenarios

The scenarios were designed to test (i) scalability of the algorithms as the number of readers, writers and servers increases, and (ii) the relation between quorum system deployment and operation latency. In particular we consider the following parameters:

1. **Number of Participants:** We run every test with 10, 20, 30, and 40 readers and writers. To test the scalability of the algorithms with respect to the number of replicas in the system we run all of the above tests with 10, 15, 20, and 25 servers. Such tests highlight whether an algorithm is affected by the number of participants in the system. Changing the number of readers and writers help us investigate how each algorithm handles an increasing number of concurrent read and write operations. The more the servers on the other hand, the more concurrent values may coexist in the service. So, algorithms like APRX-SFW and CWFR, who examine all the discovered values and do not rely on the maximum value, may suffer from local computation delays.

2. **Quorum System Construction:** As mentioned in Section 4.2 we use majority quorums as they can provide quorum systems with high intersection degree. So, assuming that f servers may crash we construct quorums of size $|\mathcal{S}| - f$. As the number of servers $|\mathcal{S}|$ varies between 10,15,20, and 25, we run the tests for two different failure values, i.e. $f = 1$ and $f = 2$. This affects our environment in two ways:

   (i) We get quorum systems with different quorum intersection degrees. According to [7] for every given $\mathcal{S}$ and f we obtain a $(\frac{|\mathcal{S}|}{f} - 1)$-wise quorum system.

   (ii) We obtain quorum systems with different number of quorum members. For example assuming 15 servers and 1 failure we construct 15 quorums, whereas assuming 15 servers and 2 failures we construct 105 different quorums.

Changes on the quorum constructions help us evaluate how the algorithms handle various intersection degrees and quorum systems of various memberships.

Another parameter we considered was operation frequency. Due to the invocation of operations in random times between the read and write intervals as explained in Section 4.2, operation frequency varies between each and every participant. Thus, fixing different initial operation frequencies does not have an impact of the overall performance of the algorithms. For this reason we avoided running our experiments over different operation frequencies.

5 Empirical Results

In this section we discuss our findings. We compare algorithms CwFr, Aprx-Sfw, and Simple to establish conclusions on the overall performance (including computation and communication) of the algorithms. All the plots of our experiments appear in the Appendix. Below we make clear references to those plots and we present parts of those plots where necessary.

To examine the impact of computation on the operation latency, we compare the performance of algorithms Aprx-Sfw and CwFr to the performance of algorithm Simple. Recall that algorithm Simple requires insignificant computation. Thus, the latency of an operation in Simple directly reflects four communication delays (i.e., two rounds).

In the next paragraphs we present how the read and write operation latency is affected by the scenarios we discussed in Section 4.4. A general conclusion that can be extracted from the experiments is that in most of the runs, algorithms Aprx-Sfw and CwFr perform better than algorithm Simple. This suggests that the additional computation incurred in these two algorithms does not exceed the delay associated with a second communication round.

As mentioned before, in general, PlanetLab machines often go offline unexpectedly preventing some scenarios to finish. Due to this fact, there are some cases in the plots where we do not obtain any data for a particular scenario. As those cases are not frequent we ignore them in our conclusions below.

5.1 Variable Participation

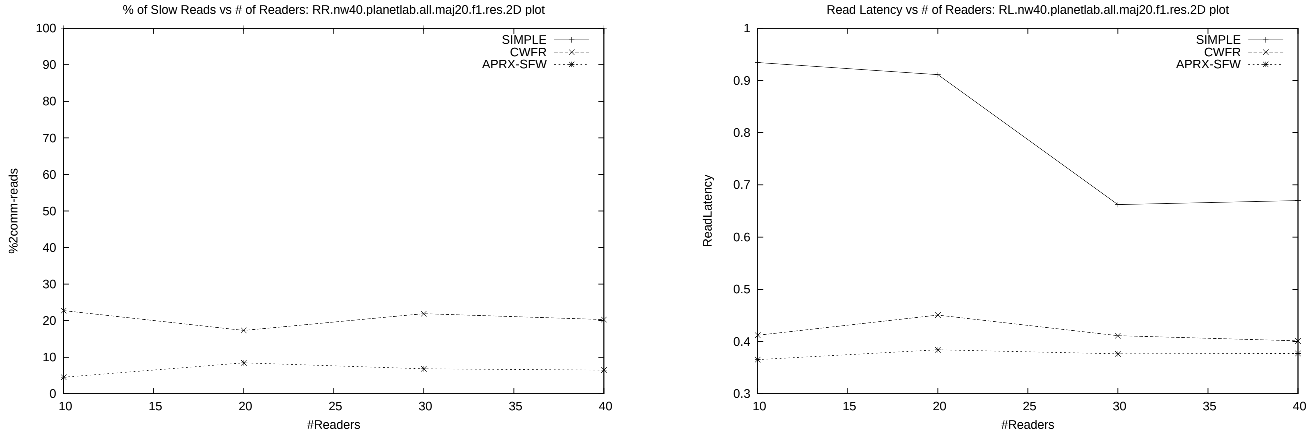
For this family of scenarios we tested the scalability of the algorithms when the number of readers, writers, and servers changes. The plots that appear in the Appendices A and B present the results for the read and write latency respectively.

The plots in Appendix A present how the performance of read operations is affected as we change the number of readers. To obtain a figure we fix the number of servers and server failures, and we vary the number of readers and writers. So, for instance, Figure 2 fixes the number of servers to $|\mathcal{S}| = 10$, and $f = 1$. Each row of the figure fixes the number of writers and varies the number of readers. Therefore we obtain four rows corresponding to $|\mathcal{W}| \in [10, 20, 30, 40]$. Each row in a figure contains a pair of plots that presents the percentage of slow reads (on the left) and the latency of each read (on the right) as we increase the number of readers.

From the plots we observe that, unlike the results we obtained on the controlled environment of NS2 in [13], Aprx-Sfw overperforms CwFrin most of the cases. This makes evident that the adverse environment of a real-time system diminishes the high concurrency between read and write operations. Notice that Aprx-Sfw and CwFr require higher computation demands when multiple writers try concurrently to change the value of the register. Computation is decreased when writes are consecutive. Thus, low concurrency favors algorithms with high computation and low communication demands.

The read performance of both Aprx-Sfw and CwFr is affected slightly by the number of readers in the system. On the other hand we observe that the number of writers seem to have a greater impact on the performance of algorithm CwFr. As the number of writers grows, both the number of slow read operations and the latency of read operations for CwFr increase. Unlike our findings in [13], the number

40 Writers:



40 Writers:

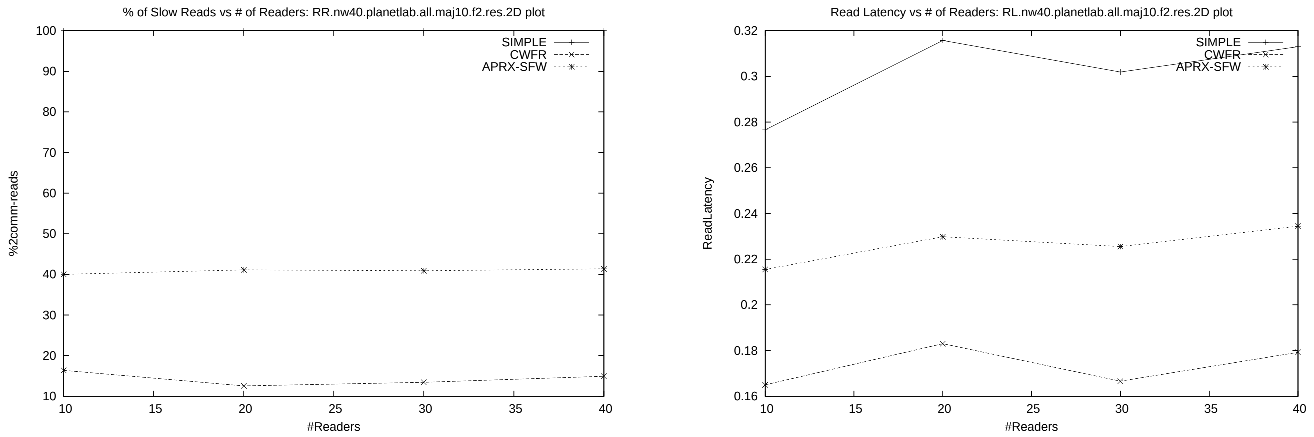
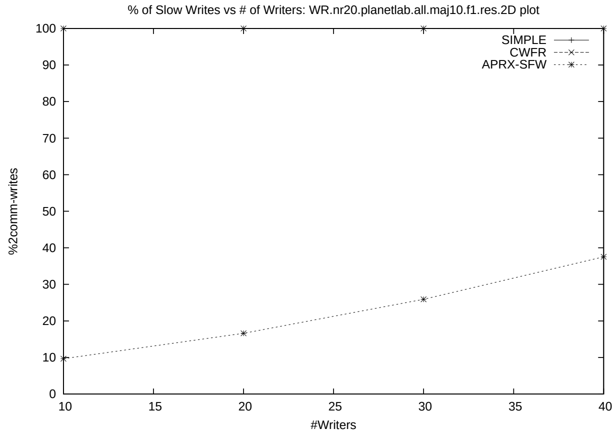


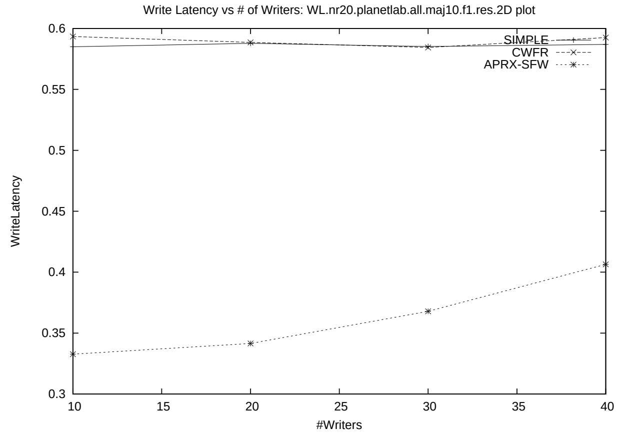
Figure 2: Plots from Figure 8 - 19-wise quorum system ($|\mathcal{S} = 20$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

of writers does not have a great impact on the performance of Aprx-Sfw. This is another evidence that write operations do not overlap due to the adverse environment, and thus Aprx-Sfw can discover quickly a valid tag without the need of examining all the candidate tags. Both CwFr and Aprx-Sfw require fewer than 20% of reads to be slow in all scenarios. The only scenario where the percentage of slow reads for Aprx-Sfw rise above 50% is when we deploy a 4-wise quorum system. This demonstrates the dependence of the predicates on the intersection degree of the underlying quorum system. We discuss this relation in the next section. The large percentage of fast read operations, keeps the overall latency of read operations for the two algorithms below the latency of read operations in Simple. That suggests that the computation burden does not exceed the latency added by a second communication round.

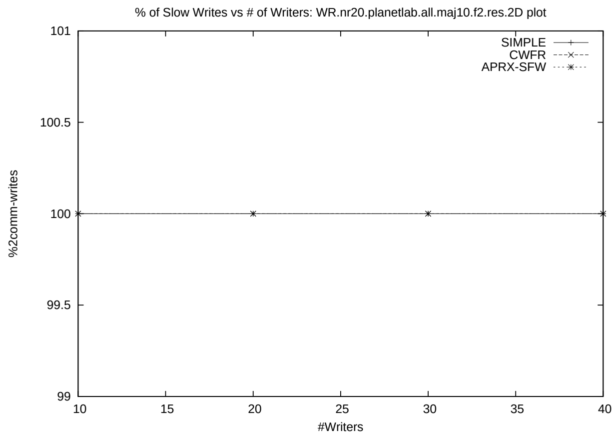
Our outcomes can be seen in Figure 2. The 20-wise intersection allows Aprx-Sfw to enable more single round read operations than CwFr even if there are 40 writers in the service. The 4-wise intersection, of the plots on the second row of Figure 2, on the other hand causes the Aprx-Sfw to introduce a higher percentage of slow reads. In this case the predicates could not get validated and thus many reads needed to perform two communication rounds. CwFr allows fewer slow reads than Aprx-Sfw. As expected, the read latency of CwFr in this scenario is less than the read latency of Aprx-Sfw. Despite the higher percentage of slow reads however, the read latency of both algorithms remains below the read latency of Simple.

20 Readers:





20 Readers:



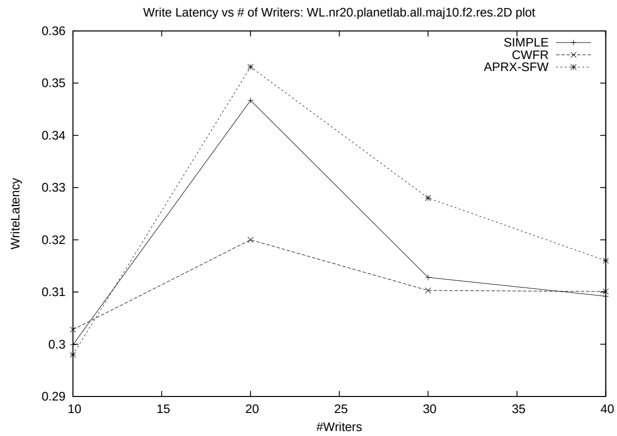


Figure 3: 9-wise and 4-wise quorum systems ($|\mathcal{S}| = 10$, $f = 1, 2$): **Left Column:** Percentage of slow writes, **Right Column:** Latency of write operations

Similar to the read operations, the performance of write operations is not affected by the number of reader participants. It is affected however by both the number of writers and servers in the system. The only algorithm that allows single round write operations is Aprx-Sfw. This is very distinct in our experiments and can be also seen in the first row of Figure 3 (part of Figure 14). When deploying a 9-wise quorum system Aprx-Sfw allows more than 60% of the writes to be fast. We observe that the percentage of slow writes increases as the number of writers grows. This is expected as more write operations may collide and thus more tags are going to be propagated in the system. It is a pleasant surprise that the latency of write operations of Aprx-Sfw is below the write latency of the algorithms that require two communication rounds. It can be taken as an evidence that the computation burden of Aprx-Sfw does not exceed the time of the second communication round. Note here that unlike the read operations, writes can be fast in Aprx-Sfw only when the write predicate holds. This characteristic can be depicted from the second row of Figure 3 (part of Figure 18) where the 4-wise intersection does not allow for the write predicate to hold. Thus, every write operation in Aprx-Sfw performs two communication rounds in this case. It is of great importance that the write latency of Aprx-Sfw is almost identical to the delay of the other two algorithms. That shows once more that the computation burden of Aprx-Sfw does not affect the latency of writes by a high margin. The spikes on the latency of the write operations in the same figure appear due to the small range of the values and the small time inconsistency that may be

Servers $\|\mathcal{S}\|$	Server Failures f	Int. Degree n	Quorums $\|\mathbb{Q}\|$
10	1	9	10
15	1	14	15
20	1	19	20
25	1	24	25
10	2	4	45
15	2	6	105
20	2	9	190
25	2	11	300

Figure 4: Quorum system parameters.

caused by the variable delays of the network.

5.2 Quorum Construction

We consider majority quorums due to the property they offer on their intersection degree [7]: if $|\mathcal{S}|$ the number of servers and up to f of them may crash then if every quorum has size $|\mathcal{S}|-f$ we can construct a quorum system with intersection degree $n = \frac{|\mathcal{S}|}{f} - 1$. Using that property we obtain the quorum systems presented on Table 4 by modifying the number of servers and the maximum number of server failures.

In Appendix C we plot the performance of read and write operations (communication rounds and latency) with respect to the number of quorum members in the quorum system. Each figure contains four pairs of plots. The first two pairs correspond to the quorum system that allows a single server failure whereas the bottom two pairs correspond to the quorum system that allows up to two server failures. For each type of quorum system the top pair describes the performance of read operations while the bottom pair the performance of write operations. Lastly, the left plot in each pair presents the percentage of slow read/write operations, and the right plot the latency of each operation respectively.

From the plots it is clear that Aprx-Sfw is affected by the number of quorums and in extent by the intersection degree of the quorum system. On the other hand CwFr does not show a clear affect. In particular we observe that the cardinality of the quorum system, reduces the percentage of slow reads for Aprx-Sfw. The distinction between Aprx-Sfw and CwFr is depicted nicely from the upper row of Figure 5. The line of the slow read percentage of Aprx-Sfw crosses below the line of CwFr when the intersection degree grows to $\frac{20}{2} - 1 = 9$, and remains lower for larger values of the intersection degree. Operation latency on the other hand is not proportional to the reduction on the amount of slow operations. Both algorithms Aprx-Sfw and CwFr experience an incrementing trend on the latency of the read operations as the number of servers and the quorum members increases. It is worth noting that the latency of read operations in Simple also follows an increasing trend even though every read operation requires two communication rounds to complete. This is evidence that the increase on the latency is partially caused by the communication between the readers and the servers: as the servers increase in number the readers need to send and wait for more messages. The latency of read operations in CwFr is not affected greatly as the number of servers changes. As a result, CwFr appears to maintain a read latency close to 0.4 sec in every scenario. With this read latency CwFr over-performs algorithm Simple in every scenario as the latter maintains a read latency between 0.6 and 1 sec. Algorithm Aprx-Sfw also experiences an increasing change on the latency of read operations. The read latency in Aprx-Sfw is affected by both the number of quorums in the system, and the number of writers in the system. As a result the latency of read operations in Aprx-Sfw in conditions with a large number of quorum members and writers may exceed the read latency of Simple up to 5 times. The reason for such performance is that every read operation in Aprx-Sfw the reader examines the tags assigned to every writer and for every tag runs the approximation algorithm on a number of quorums in the system. The more the writers and the quorums in the system, the more time the read operation takes to complete.

20 Readers, 30 Writers, f=2:

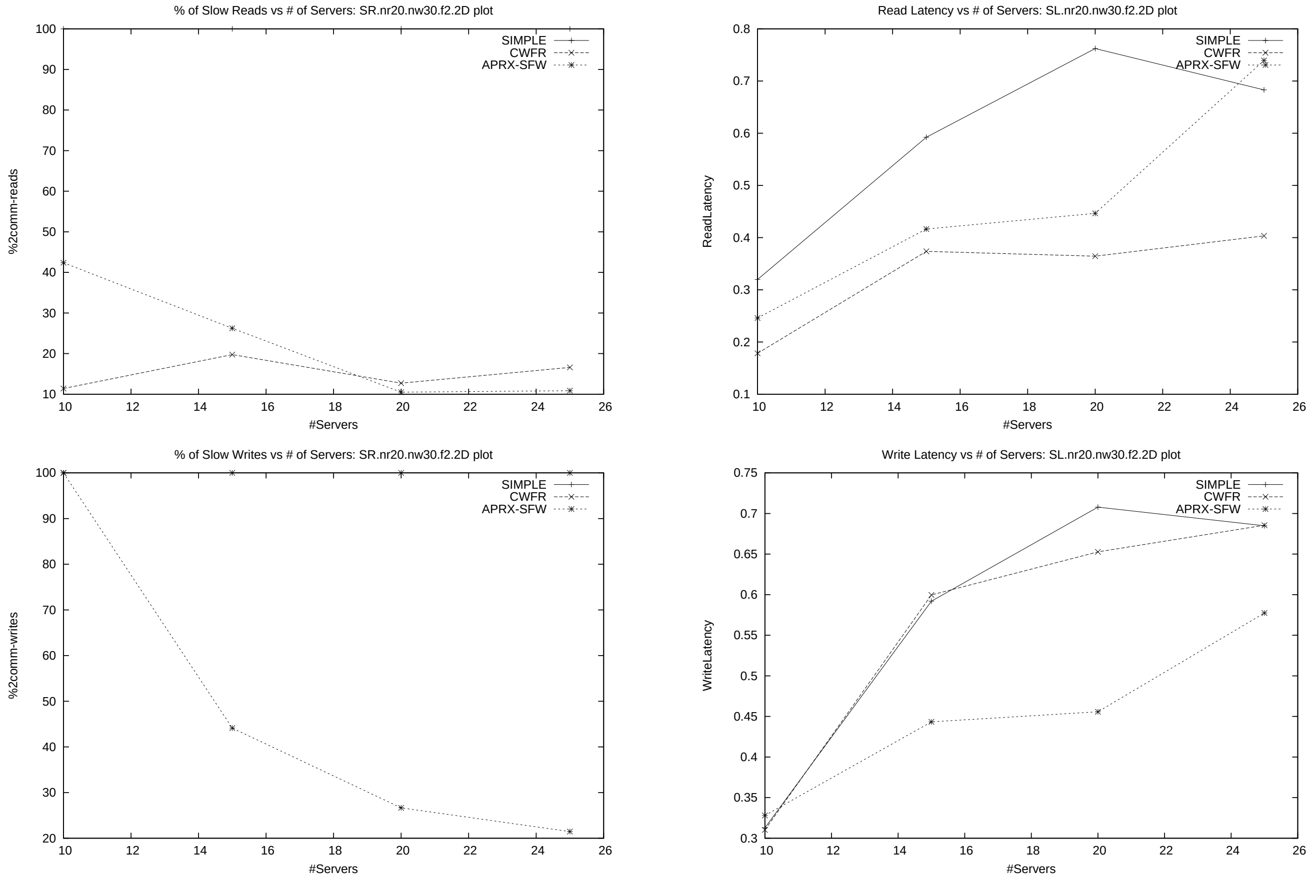


Figure 5: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

Similar observations can be made for the write operations. Observe that although both CwFr and Simple require two communication rounds per write operation, the write latency in these algorithms is affected negatively by the increase of the number of quorums in the system. As we said before this is evidence of the higher communication demands when we increase the number of servers. We also note that the latency of the write operations in Simple is almost identical to the latency of the read operations of the same algorithm. This proves the fact that the computation demands in either operation is also identical. As for Aprx-Sfw we observe that the increase on the number of servers reduces the amount of slow write operations (similar finding is noted in [13]). Also we observe that the average write latency of Aprx-Sfw increases as the number of quorums increases in the system. Interestingly however, unlike the findings in [13], the latency of writes does remain in most cases below the competition. Comparing with the latency of read operations it appears that although Aprx-Sfw may allow more fast read operations than writes under the same conditions, the average latency of each read is almost the same as the latency of write operations. An example of this behavior can be seen in Figure 5 (part of Figure 31). In this example slow reads can drop as low as 10% and the average read latency climbs to an average of 0.5 seconds (the 0.7 sec point appears to be an outlier). On the other no less than 20% slow writes are allowed and the average write latency climbs just above 0.5 seconds. The simple explanation for this behavior lies on the evaluation of the read and write predicates: each reader needs to examine the latest tags assigned to every writer in the system whereas each writer only examines the tags assigns to its own write operation.

6 Conclusions

This work investigates the practicality of three MWMR atomic register algorithms designed for the asynchronous, failure prone, message passing environment. The experiments involved the implementation and evaluation of the algorithms on the planetary-scale PlanetLab platform. We compared the performance of algorithms CwFr,Aprx-Sfw and Simple. Under the adverse real time environment offered by Planetlab we were able to observer the behavior of the algorithms in real-time network conditions. We studied the relative scalability of the algorithms using variable number of participants and the impact that the deployed quorum system may have on each algorithm's performance. In particular, we tested the scalability of the algorithms by varying the number of readers, writers and servers in the system. To introduce various types of quorum systems we manipulated the number of servers along with the number of maximum server failures in the system.

The results for algorithms CwFr, Aprx-Sfw and Simple suggest that both algorithms CwFr and Aprx-Sfw over-perform Simple. Unlike the simulation results performed in [13], we observed that in a real-time environment Aprx-Sfw overperforms CwFr in most scenarios were the intersection degree is higher than 6. That is, both the percentage of slow reads along with the average read latency is lower than the read performance of CwFr. So we observed that the quorum intersection degree plays a major role for the performance of Aprx-Sfw over the rest of the algorithms. However, it appears that a high intersection degree is not necessary for Aprx-Sfw since in cases with small intersection degree less than 50% of reads needed to be slow and the latency of both the write and read operations was almost identical or better than the latency of the competition. We also observed that the number of writers, servers and quorums in the system can have a negative impact on both the number of slow operations, and the operation latency for all three algorithms. This behavior agrees with our theoretical bounds presented in Table 1. Furthermore our results suggest that the computation burden placed on each participant by Aprx-Sfw and CwFr is less than the cost of a second communication round. The computation cost is not negligible, however, since we observe cases where the percentage of slow reads decreases when the average read latency increases. Thus, the example reveals that although we need less communication, the computation costs keeps an incrementing trend to the operation latency. Algorithm CwFr does not seem to be affected much by the number of servers and quorums in the underlying quorum system. The percentage of slow reads remains in the same levels while the latency increases due mainly to the increasing communication.

Overall it is safe to claim that algorithms CwFr and Aprx-Sfw should be preferable by applications in a real-time setting. The choice of the algorithm (CwFr or Aprx-Sfw) essentially depends on the environment in which the application is to be deployed. If quorum systems with large intersection degrees are possible/available then Aprx-Sfw is a clear winner. If a general quorum construction is to be used then CwFr should be considered.

Acknowledgments. We thank Alexander Russell and Alexander A. Shvartsman for helpful discussions.

References

[1] PlanetLab: A planetary-scale networked system, `http://www.planet-lab.org`.

[2] NS2 network simulator. http://www.isi.edu/nsnam/ns/.

[3] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing memory robustly in message passing systems. *Journal of the ACM 42(1)* (1996), 124–142.

[4] Chockler, G., Gilbert, S., Gramoli, V., Musial, P. M., and Shvartsman, A. A. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 100–116.

[5] Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., and Welch, J. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceedings of 17th International Symposium on Distributed Computing (DISC)* (2003).

[6] Dutta, P., Guerraoui, R., Levy, R. R., and Chakraborty, A. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), pp. 236–245.

[7] Englert, B., Georgiou, C., Musial, P. M., Nicolaou, N., and Shvartsman, A. A. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)* (2009), pp. 240–254.

[8] Englert, B., and Shvartsman, A. A. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)* (2000), pp. 454–463.

[9] Fan, R., and Lynch, N. Efficient replication of large data objects. In *Distributed algorithms* (Oct 2003), F. E. Fich, Ed., vol. 2848/2003 of *Lecture Notes in Computer Science*, pp. 75–91.

[10] Georgiou, C., Nicolaou, N., Russel, A., and Shvartsman, A. A. Towards feasible implementations of low-latency multi-writer atomic registers. In *10th Annual IEEE International Symposium on Network Computing and Applications* (August 2011).

[11] Georgiou, C., Nicolaou, N., Russel, A., and Shvartsman, A. A. Towards feasible implementations of low-latency multi-writer atomic registers. Tech. Rep. TR-11-03, Dept. of Computer Science, University of Cyprus, Cyprus, March 2011.

[12] Georgiou, C., and Nicolaou, N. C. Algorithm cwfr: Using quorum views for fast reads in the MWMR setting. Tech. Rep. TR-10-05, Dept. of Computer Science, University of Cyprus, Cyprus, December 2010.

[13] Georgiou, C., and Nicolaou, N. C. Simulating efficient MWMR atomic register implementations on the NS2 network simulator. Tech. Rep. TR-11-06, Dept. of Computer Science, University of Cyprus, Cyprus, May 2011.

[14] Georgiou, C., Nicolaou, N. C., and Shvartsman, A. A. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 289–304.

[15] Georgiou, C., Nicolaou, N. C., and Shvartsman, A. A. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 62–79. A preliminary version of this work appeared in the proceedings 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'06).

[16] Gramoli, V., Anceaume, E., and Virgillito, A. SQUARE: scalable quorum-based atomic memory with local reconfiguration. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), ACM, pp. 574–579.

[17] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[18] Lynch, N., and Shvartsman, A. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)* (2002), pp. 173–190.

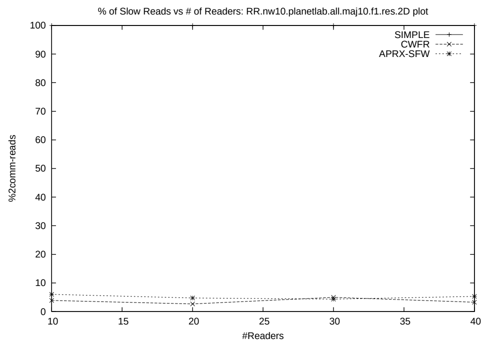
[19] Lynch, N. A., and Shvartsman, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.

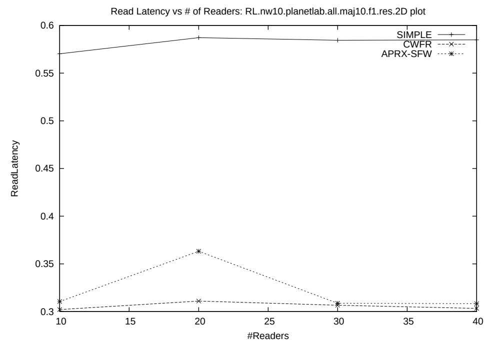
Appendix: Figures

A Read Performance

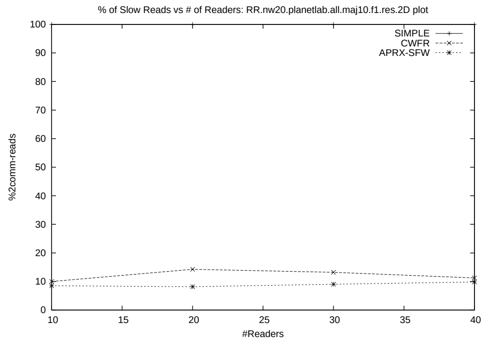
Below we present the plots regarding the read performance under variable number of writers and quorum constructions. In particular, we run the SIMPLE, CWFR, and APRX-SFW algorithms using quorum constructions with eight different intersection degrees by setting the number of servers to 10, 15, 20, and 25, and by tolerating 1 and 2 server failures. We assume majority quorums, where each quorum Q_i has size $|Q_i| = |\mathcal{S}| - f$, where f the maximum number of server failures. We test each quorum system, using 10,20, 40, and 80 readers and writers. By fixing the intersection degree and the number of writers a plot depicts the performance of read operations as we increase the number of readers in the system. Such plots help us determine the scalability of the algorithms in terms of reader participants.

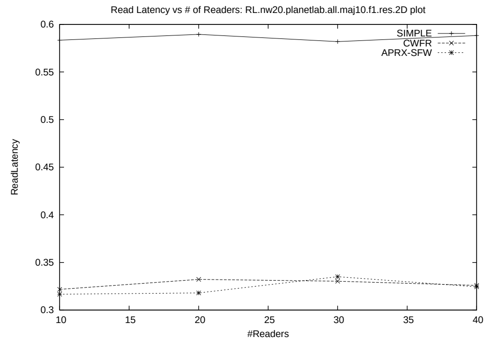
10 Writers:



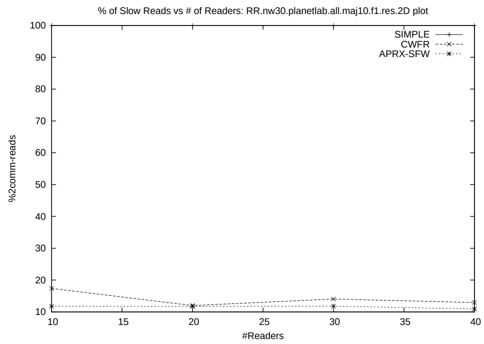


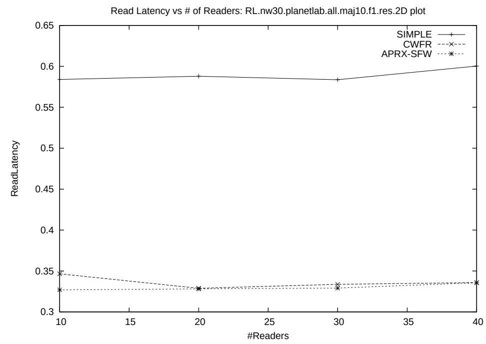
20 Writers:



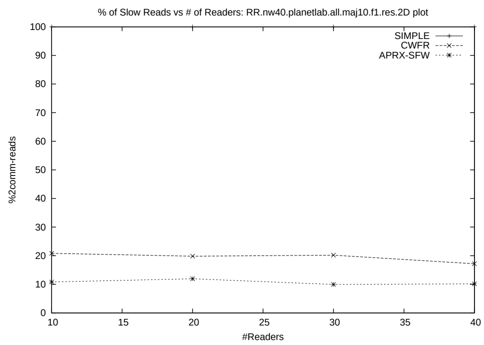


30 Writers:





40 Writers:



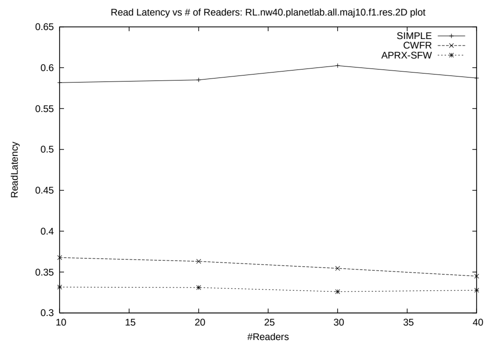
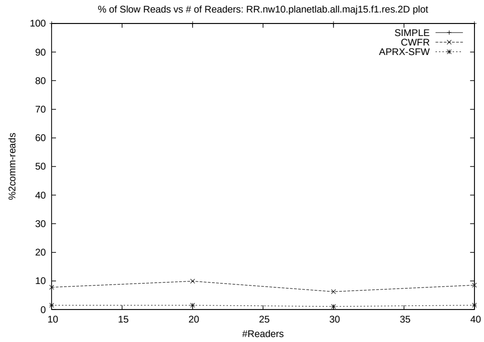
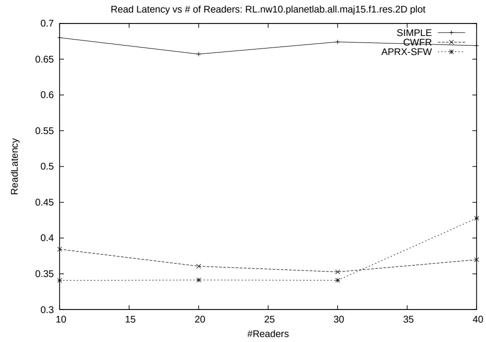


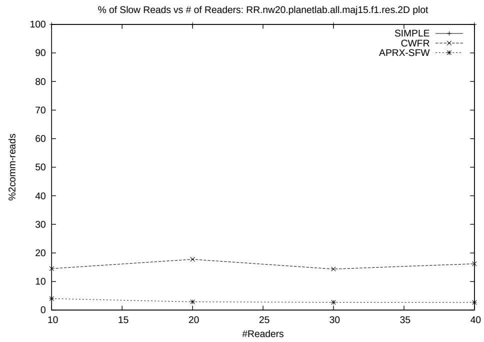
Figure 6: 9-wise quorum system ($|\mathcal{S}| = 10,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

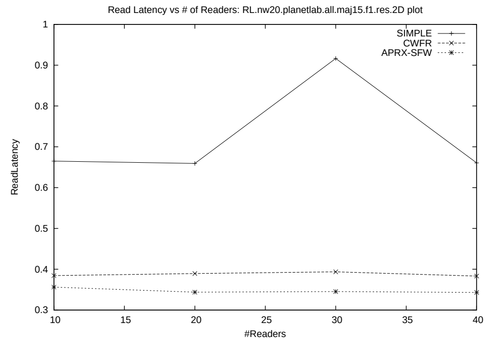
10 Writers:



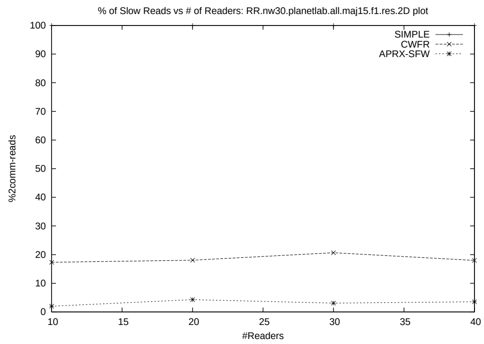


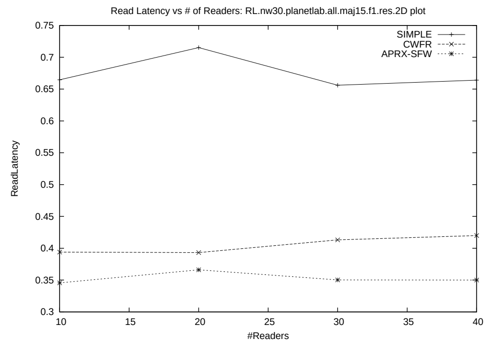
20 Writers:



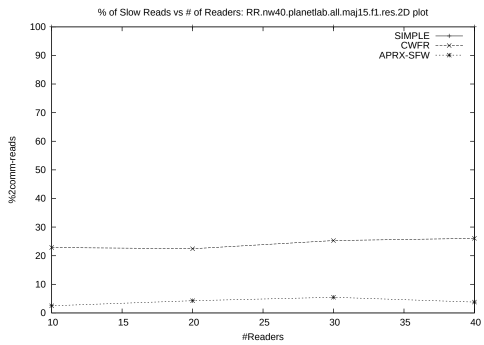


30 Writers:





40 Writers:



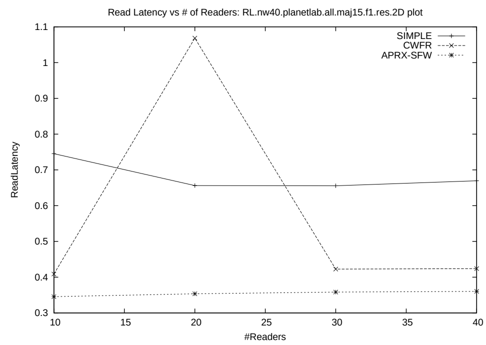
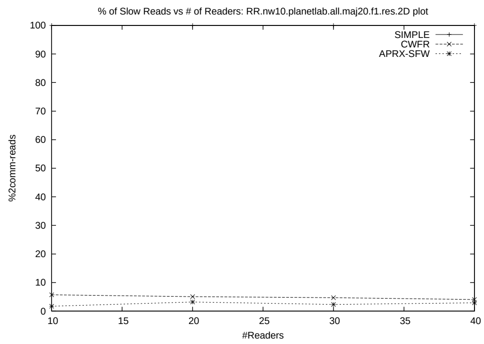
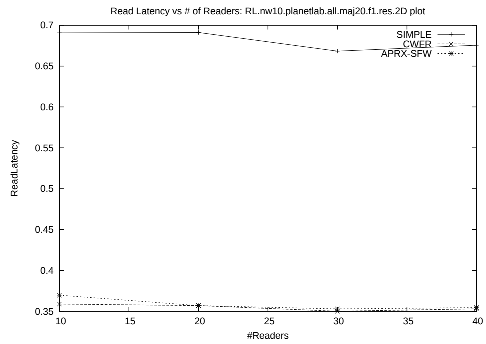


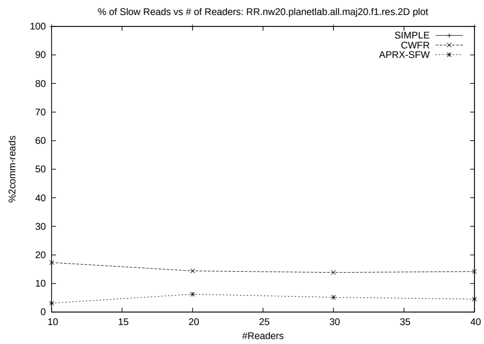
Figure 7: 14-wise quorum system ($|\mathcal{S}| = 15$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

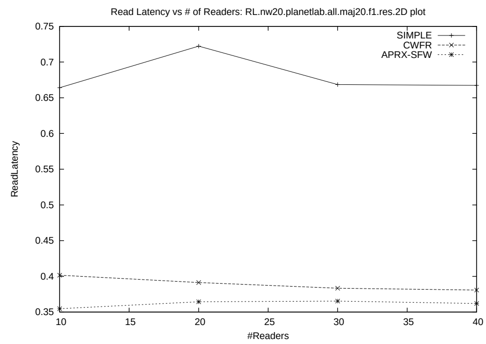
10 Writers:



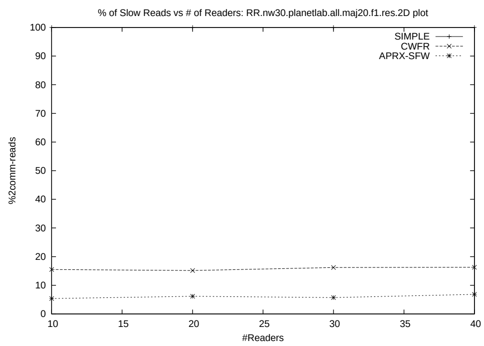


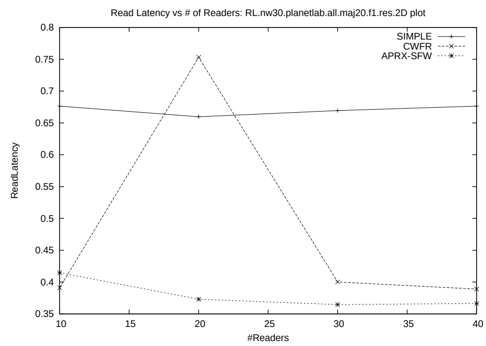
20 Writers:



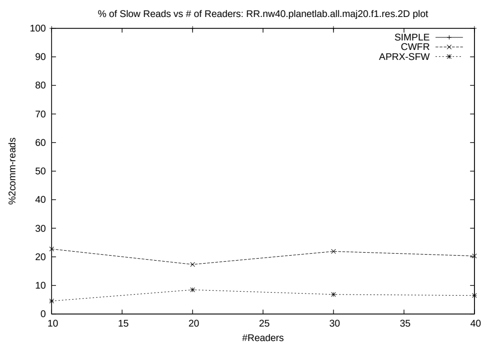


30 Writers:





40 Writers:



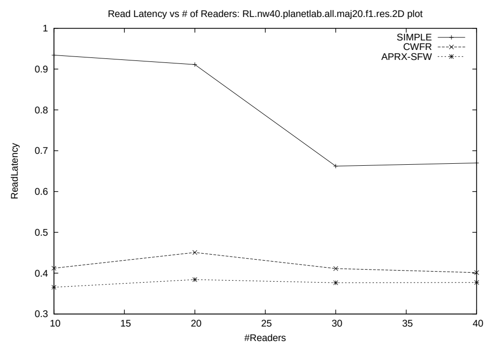
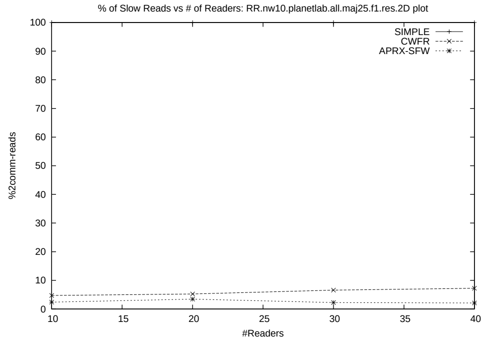
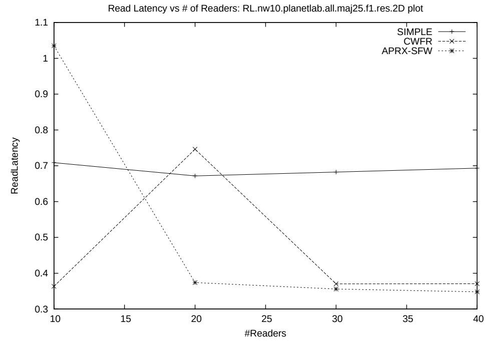


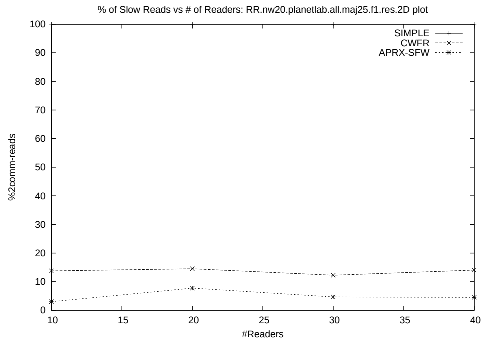
Figure 8: 19-wise quorum system ($|\mathcal{S}| = 20,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

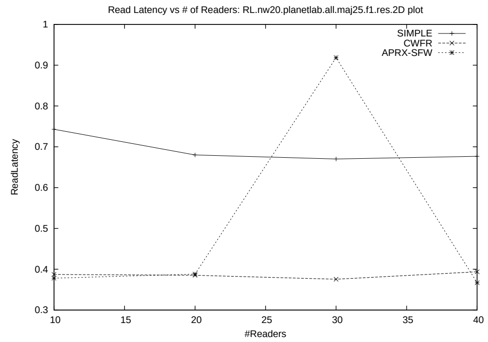
10 Writers:



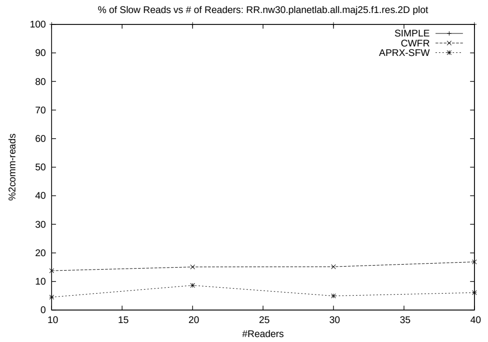


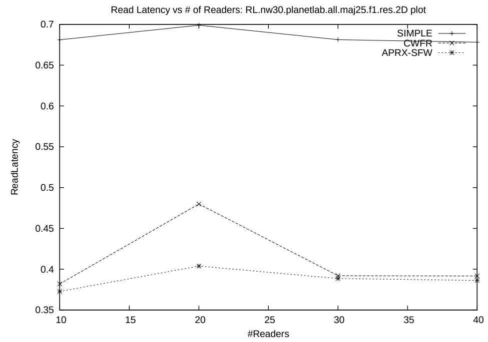
20 Writers:



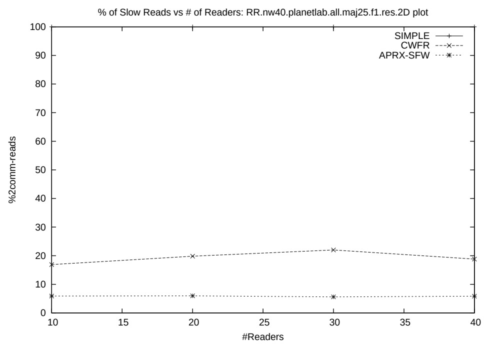


30 Writers:





40 Writers:



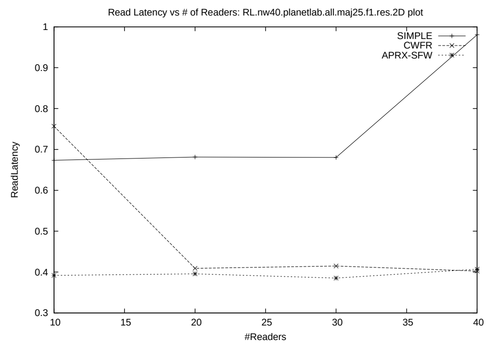
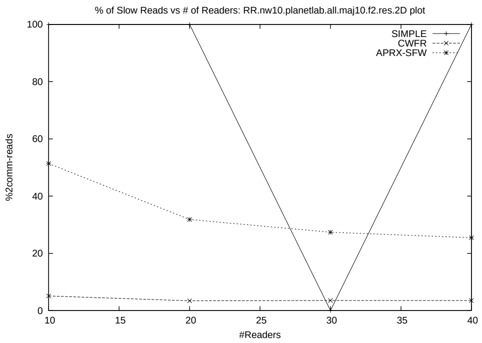
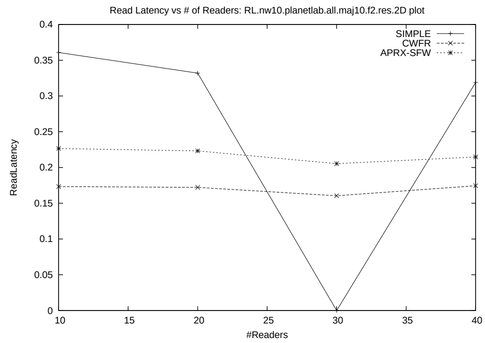


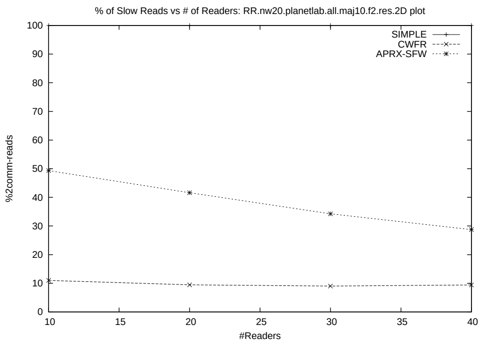
Figure 9: 24-wise quorum system ($|\mathcal{S}| = 25,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

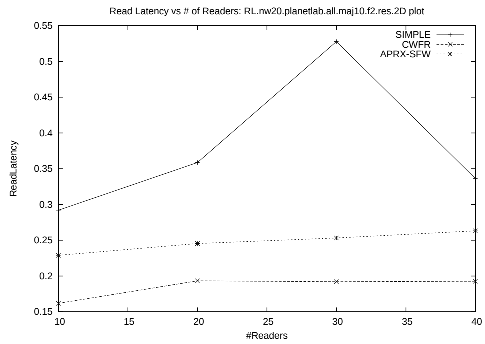
10 Writers:



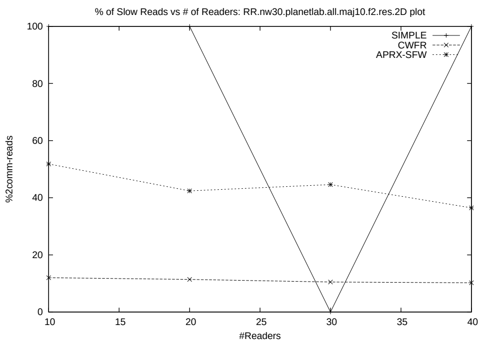


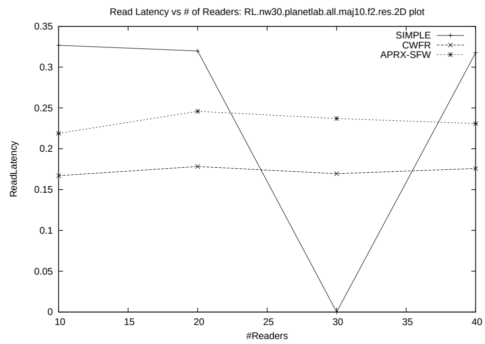
20 Writers:



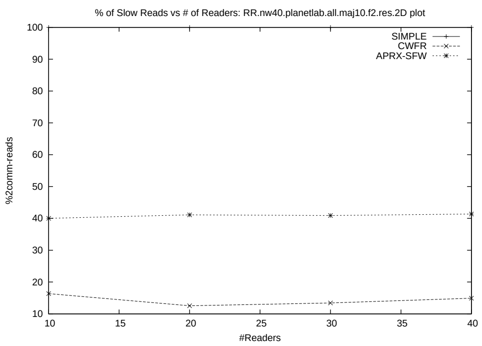


30 Writers:





40 Writers:



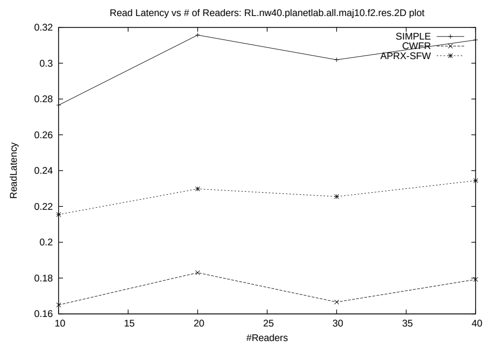
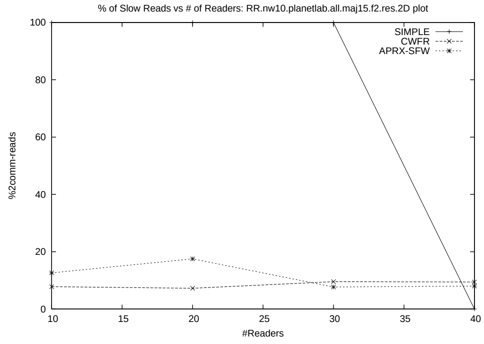
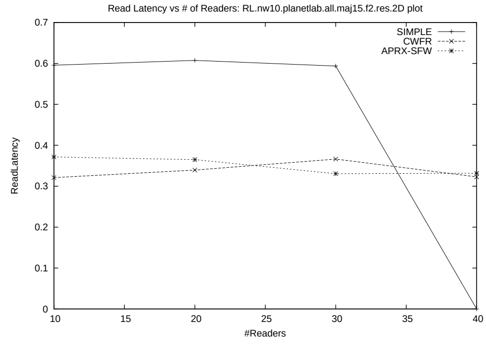


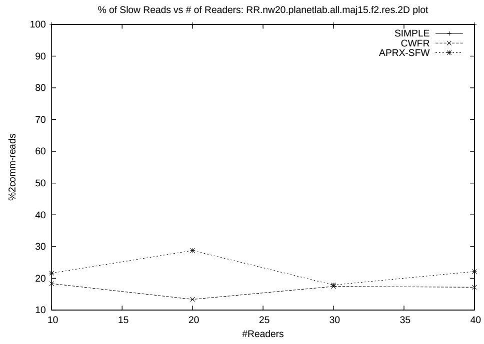
Figure 10: 4-wise quorum system ($|\mathcal{S}| = 10$, $f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

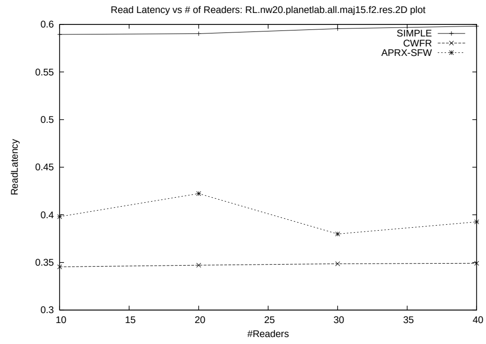
10 Writers:



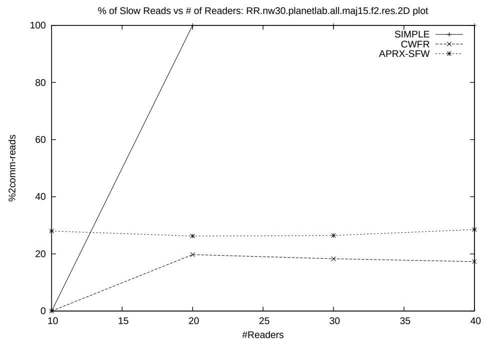


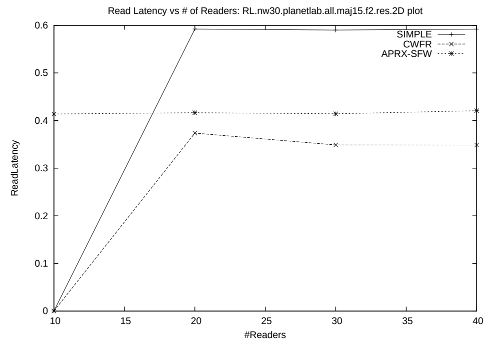
20 Writers:



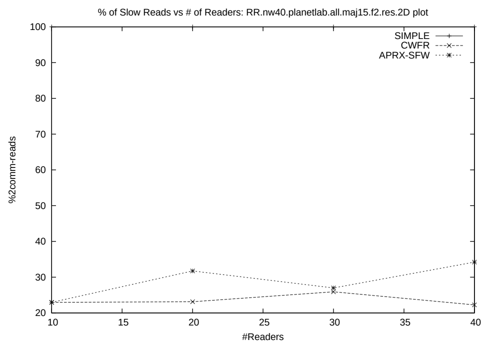


30 Writers:





40 Writers:



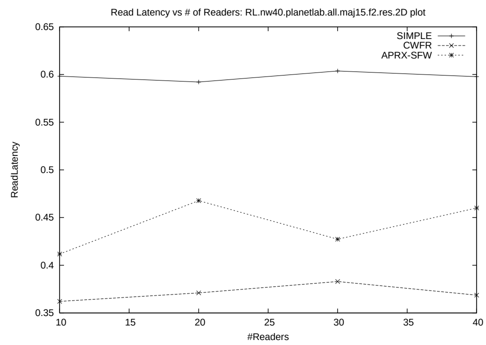
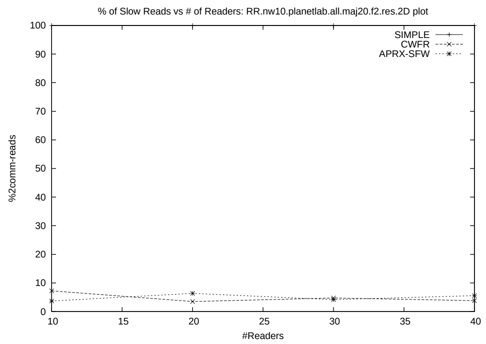
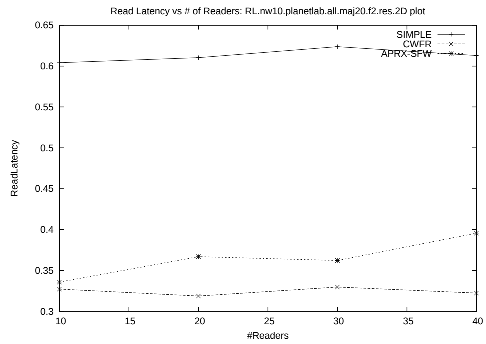


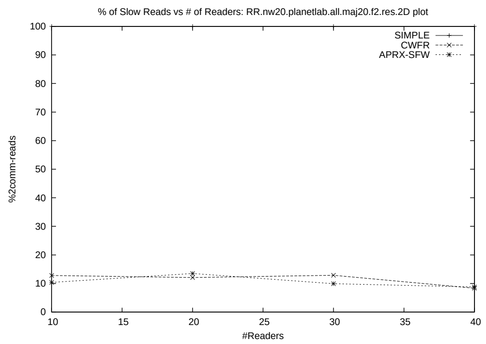
Figure 11: 6-wise quorum system ($|\mathcal{S}| = 15,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

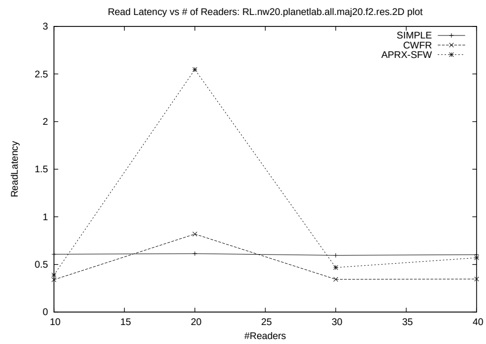
10 Writers:



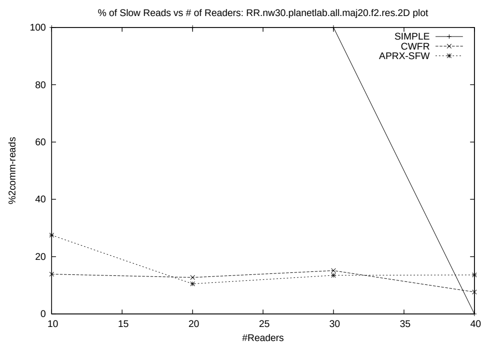


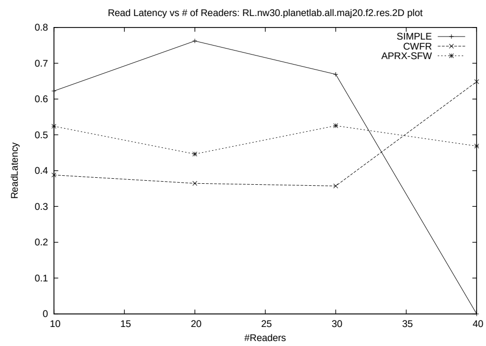
20 Writers:



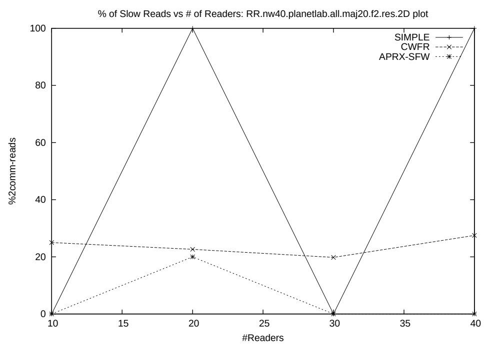


30 Writers:





40 Writers:



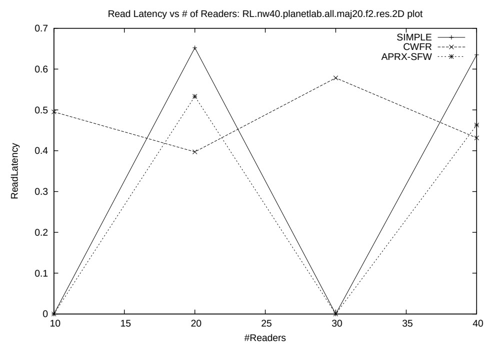
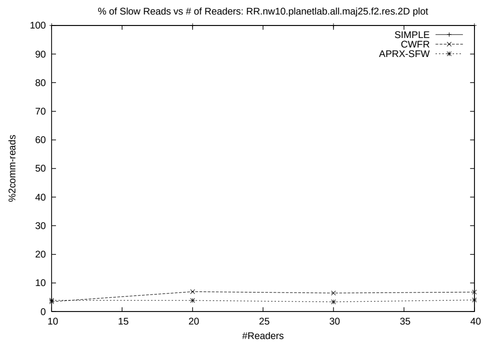
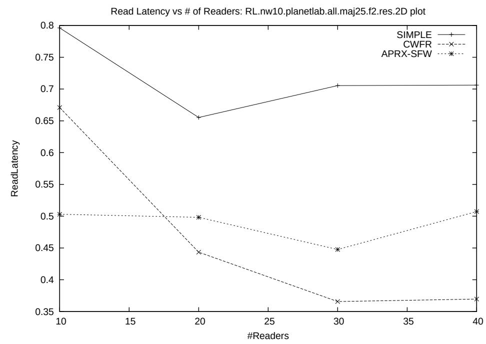


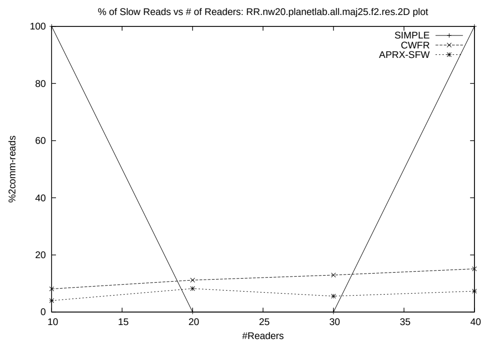
Figure 12: 9-wise quorum system ($|\mathcal{S}| = 20$, $f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

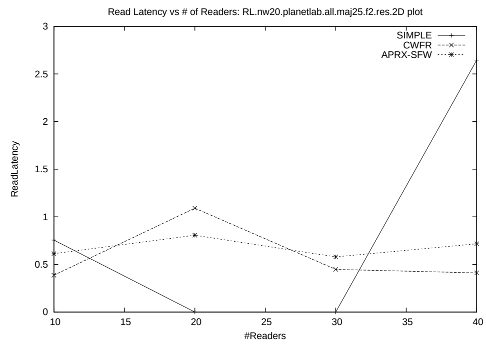
10 Writers:



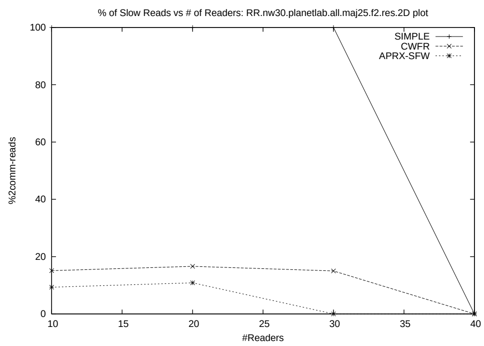


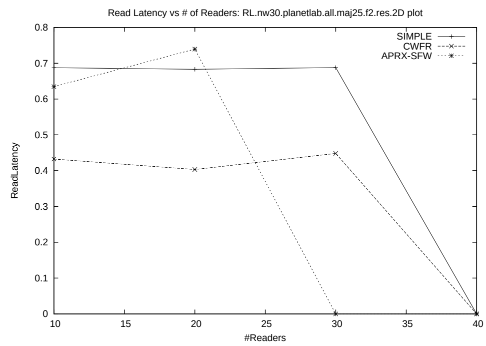
20 Writers:



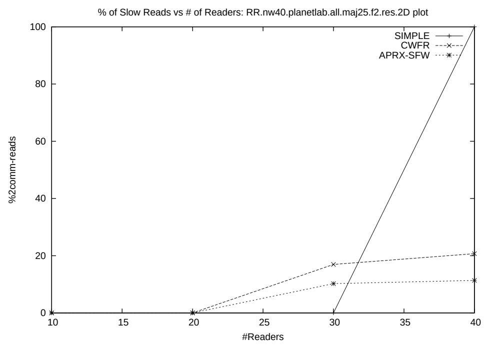


30 Writers:





40 Writers:



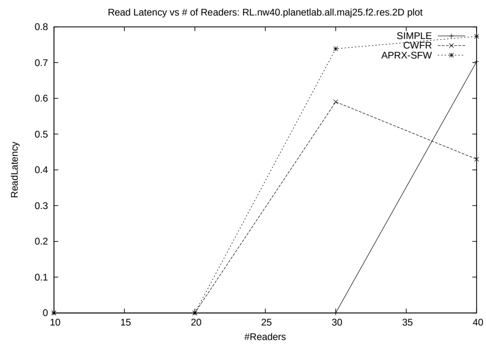
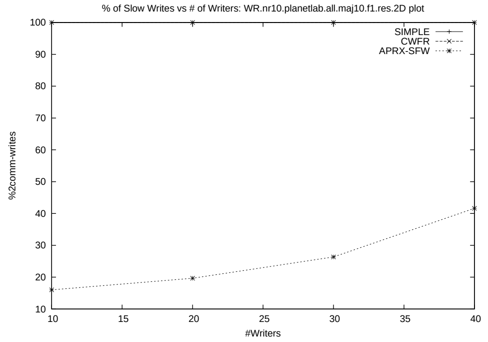


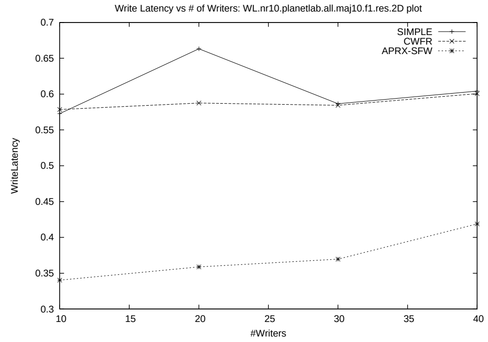
Figure 13: 11-wise quorum system ($|\mathcal{S}| = 25,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

B Write Performance

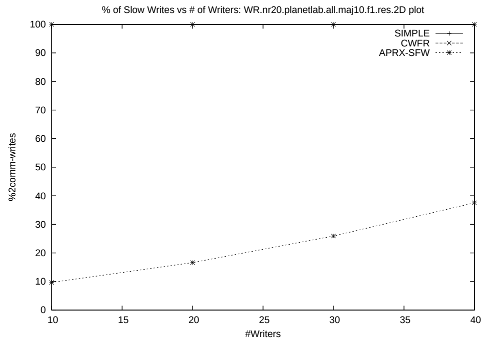
Below we present the plots regarding the write performance under variable number of readers and quorum constructions. In particular, we run the Simple, CwFr, and Aprx-Sfw algorithms using quorum constructions with eight different intersection degrees by setting the number of servers to 10, 15, 20, and 25, and by tolerating 1 and 2 server failures. We assume majority quorums, where each quorum Q_i has size $|Q_i| = |\mathcal{S}| - f$, where f the maximum number of server failures. We test each quorum system, using 10,20, 30, and 40 readers and Readers. By fixing the intersection degree and the number of readers a plot depicts the performance of write operations as we increase the number of Readers in the system. Such plots help us determine the scalability of the algorithms in terms of writer participation.

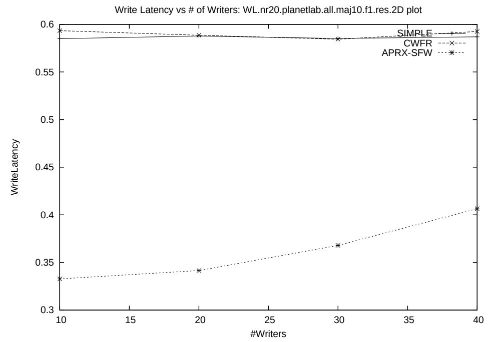
10 Readers:



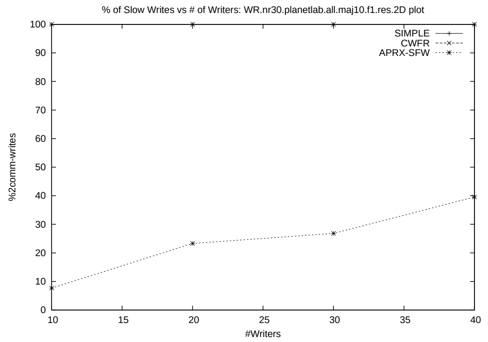


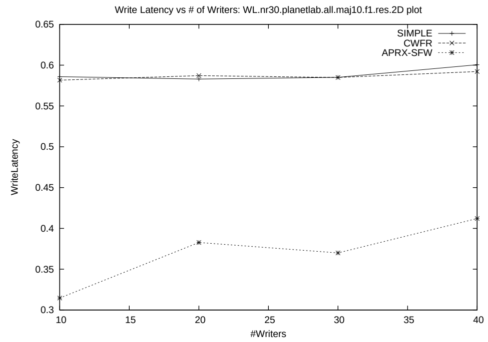
20 Readers:



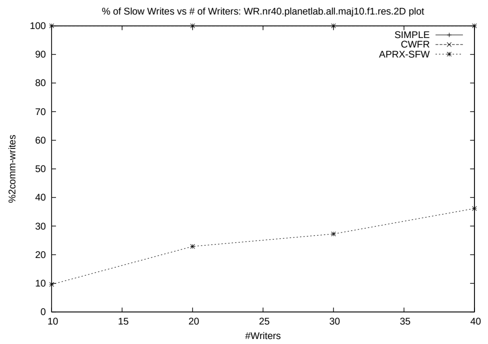


30 Readers:





40 Readers:



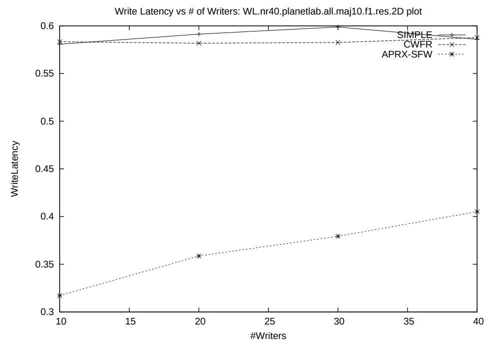
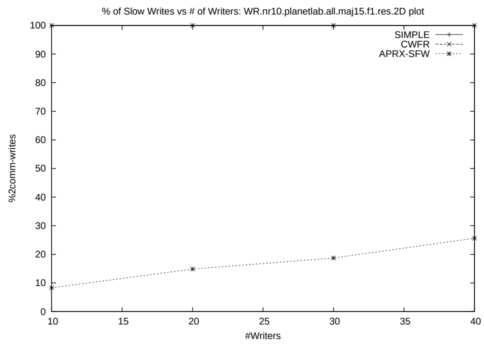
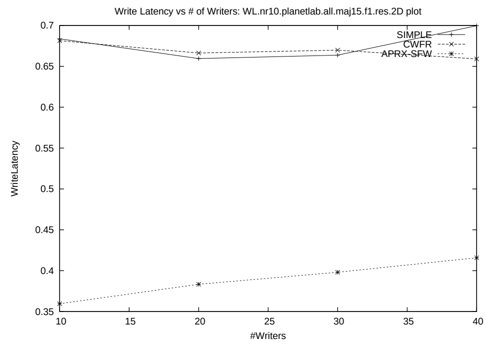


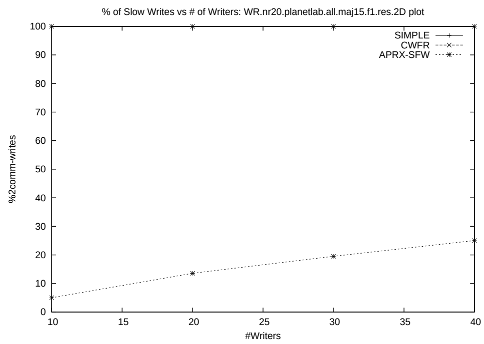
Figure 14: 9-wise quorum system ($|\mathcal{S}| = 10,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

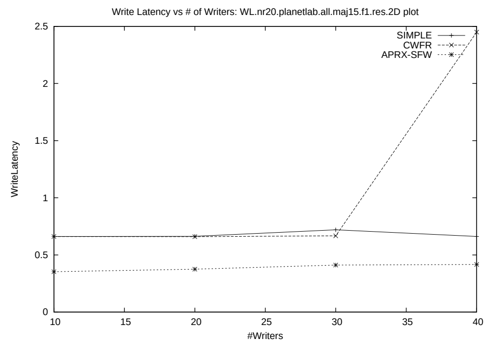
10 Readers:



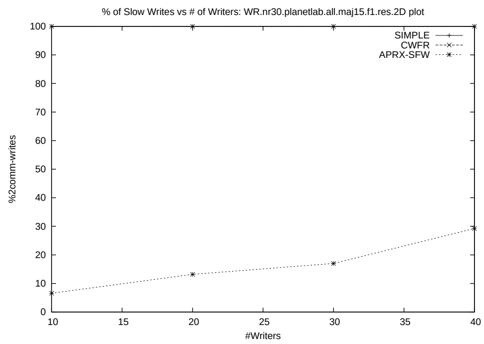


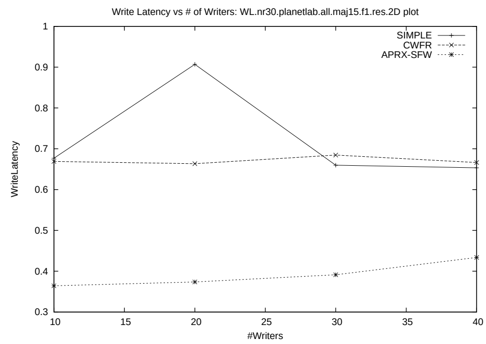
20 Readers:



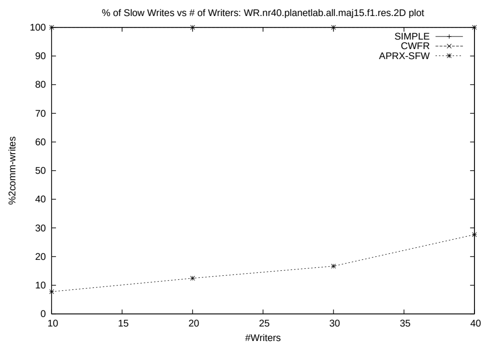


30 Readers:





40 Readers:



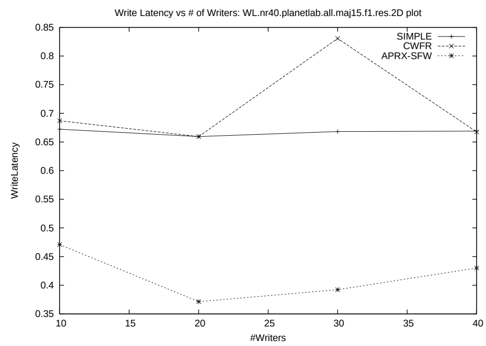
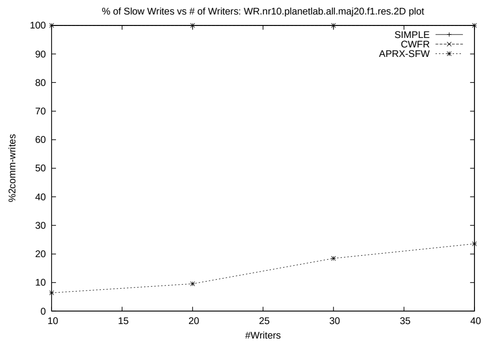
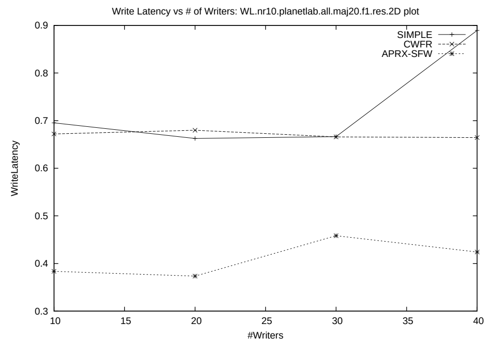


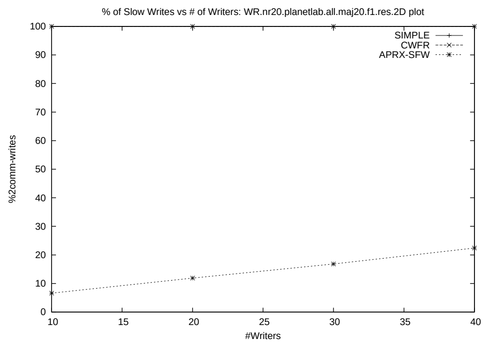
Figure 15: 14-wise quorum system ($|\mathcal{S}| = 15,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

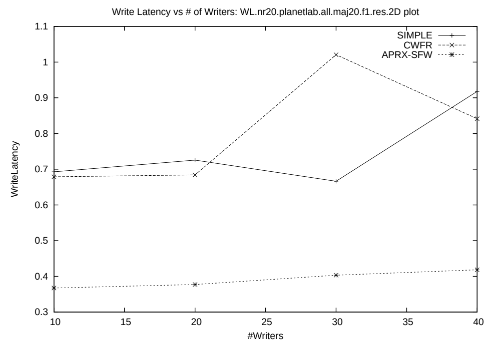
10 Readers:



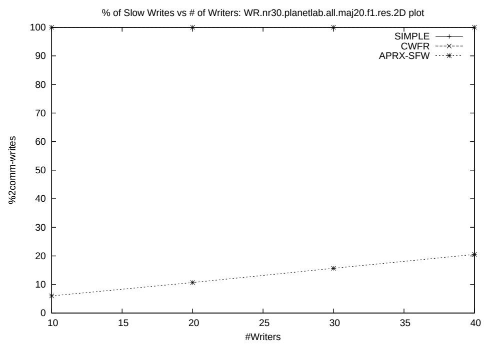


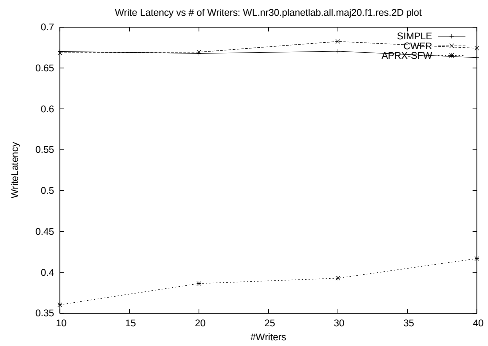
20 Readers:



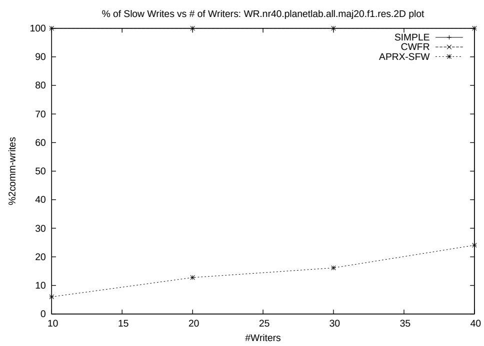


30 Readers:





40 Readers:



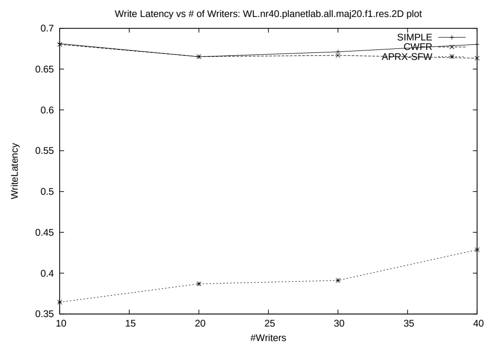
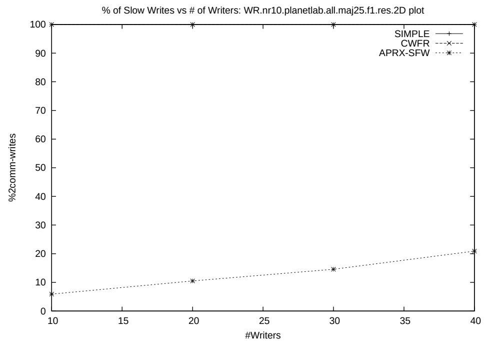
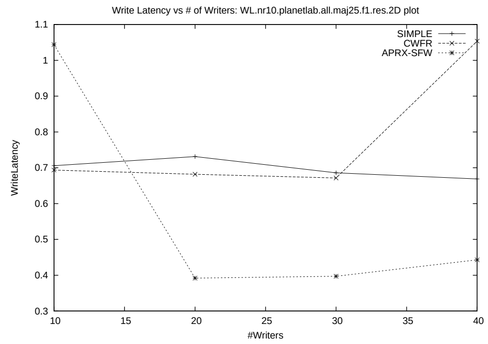


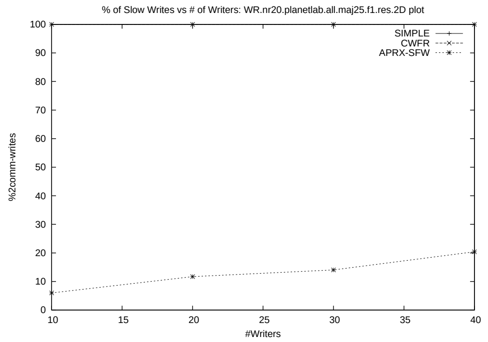
Figure 16: 19-wise quorum system ($|\mathcal{S}| = 20$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

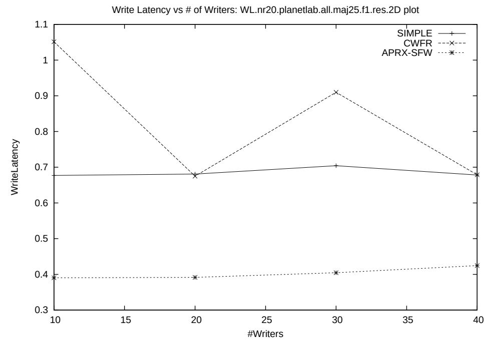
10 Readers:



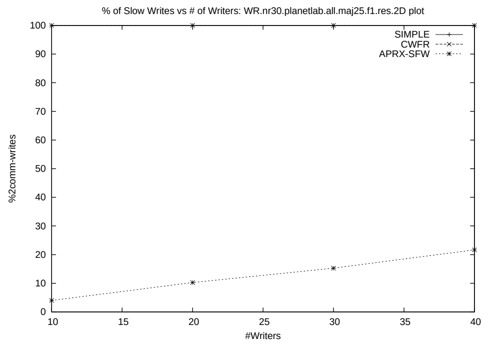


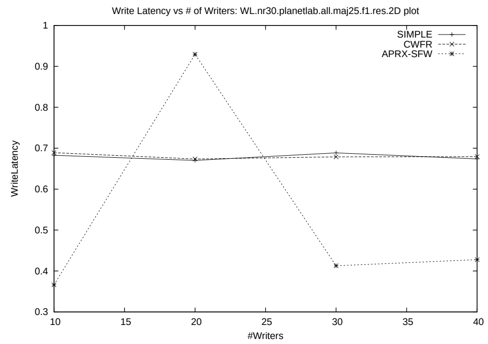
20 Readers:



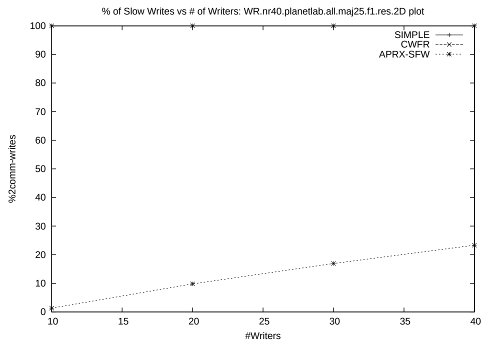


30 Readers:





40 Readers:



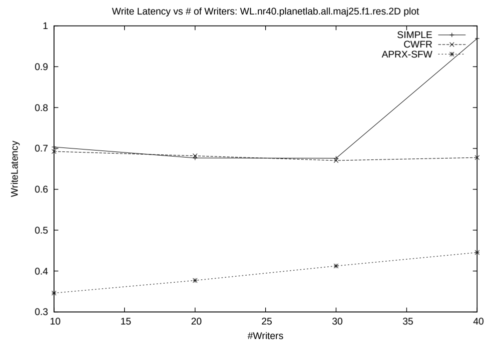
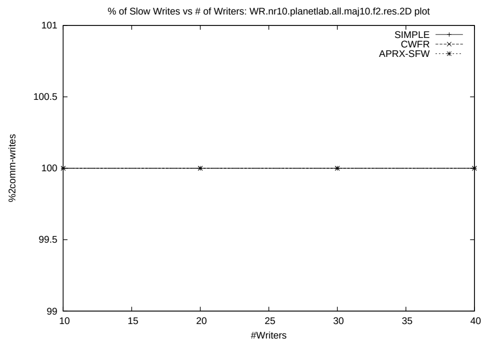
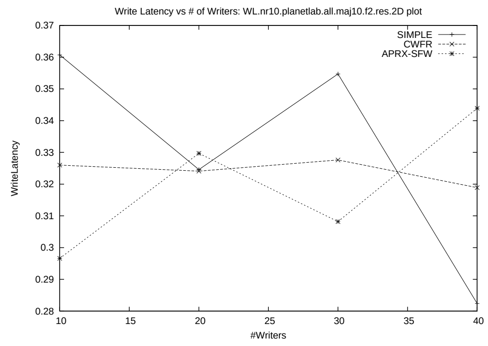


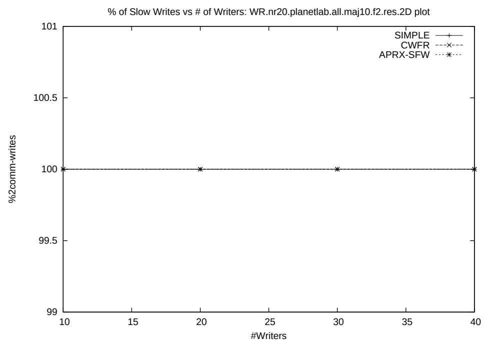
Figure 17: 24-wise quorum system ($|\mathcal{S}| = 25$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

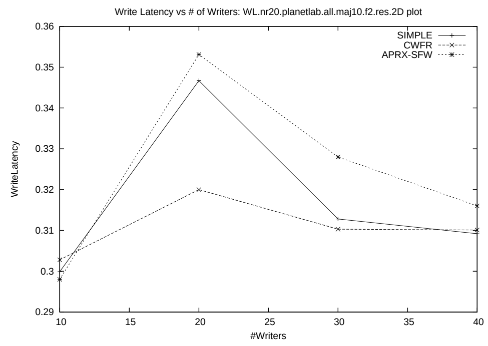
10 Readers:



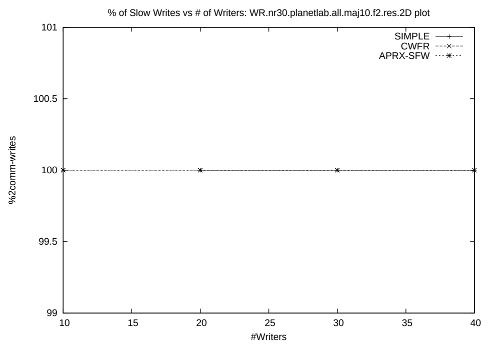


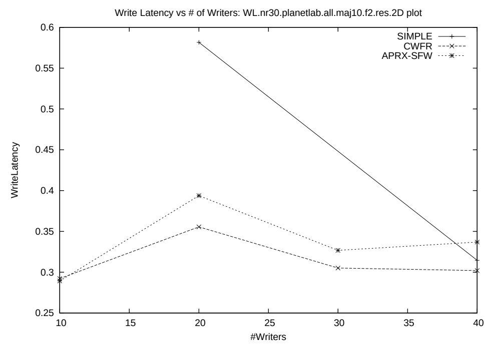
20 Readers:



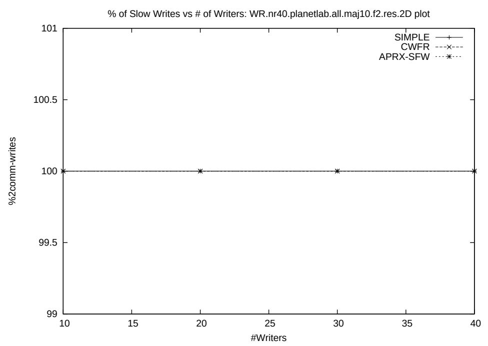


30 Readers:





40 Readers:



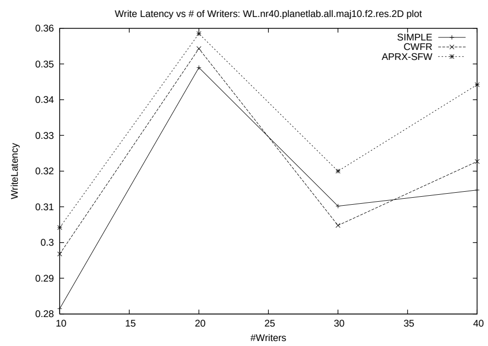
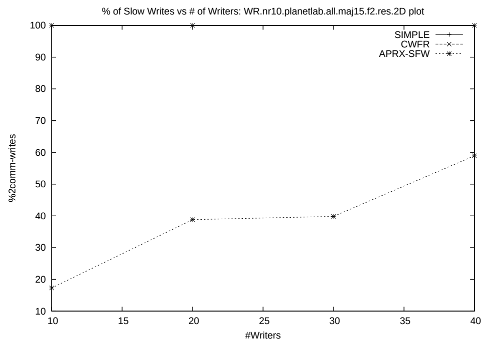
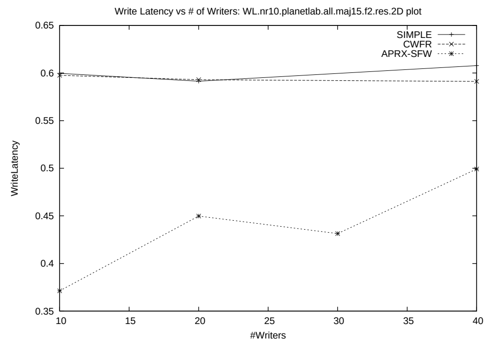


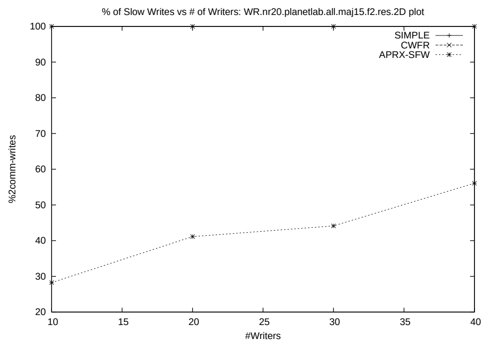
Figure 18: 4-wise quorum system ($|\mathcal{S}| = 10,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

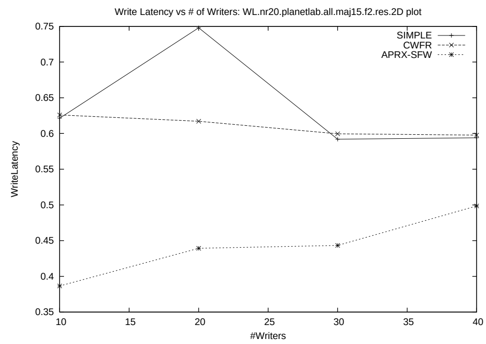
10 Readers:



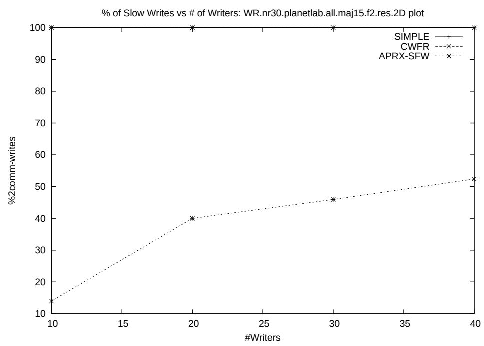


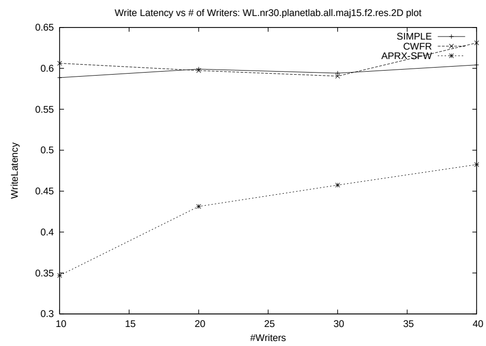
20 Readers:



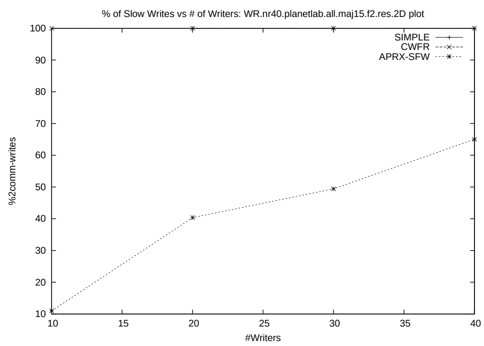


30 Readers:





40 Readers:



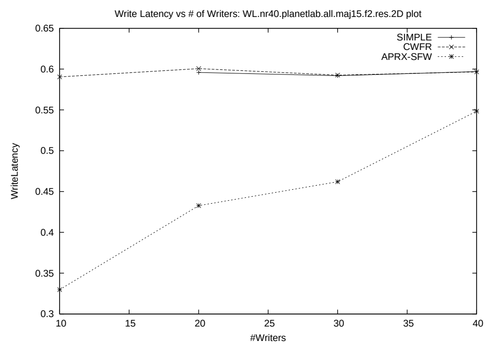
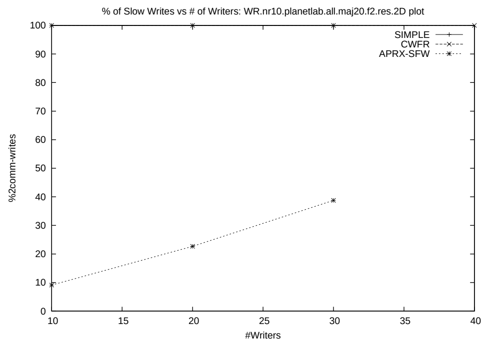
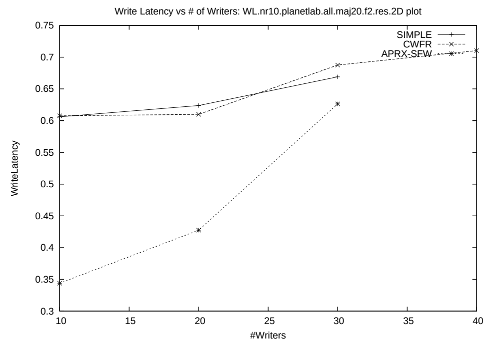


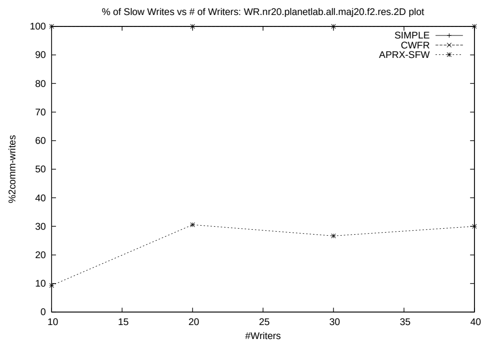
Figure 19: 6-wise quorum system ($|\mathcal{S}| = 15,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

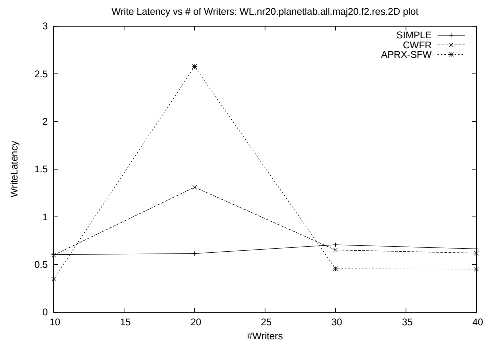
10 Readers:



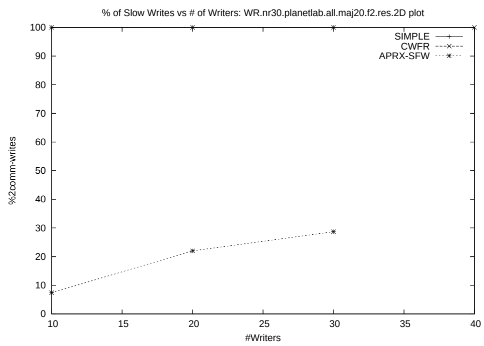


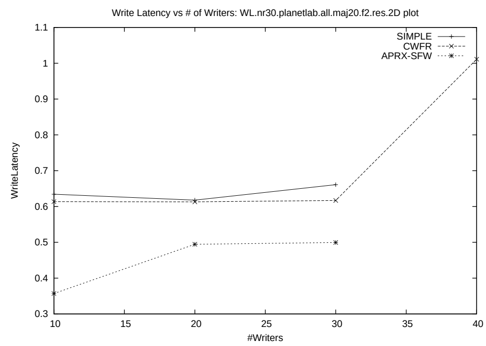
20 Readers:



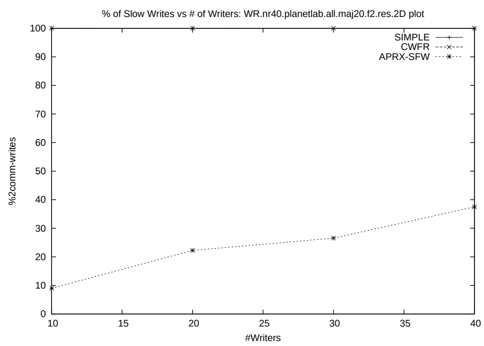


30 Readers:





40 Readers:



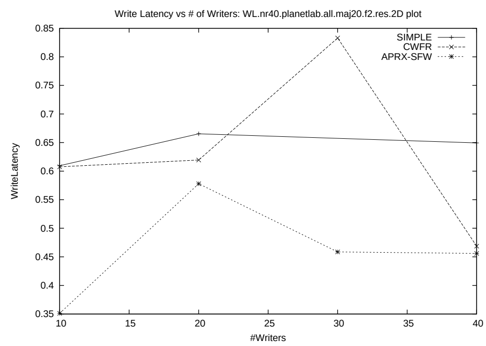
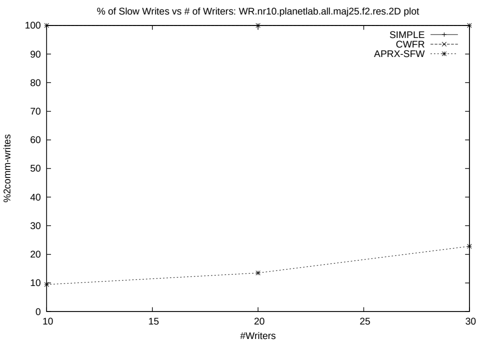
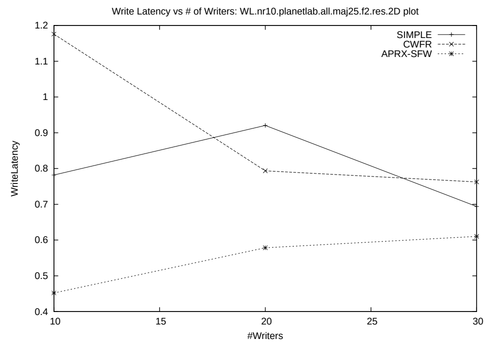


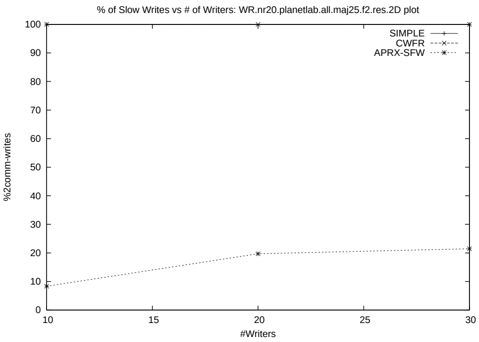
Figure 20: 9-wise quorum system ($|\mathcal{S}| = 20,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

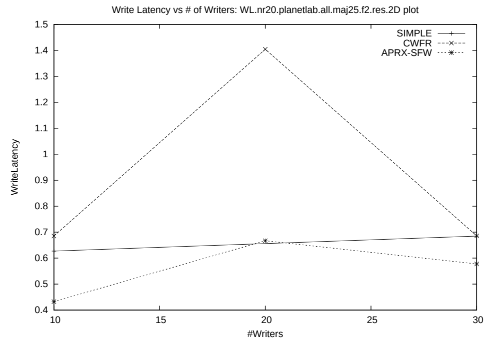
10 Readers:



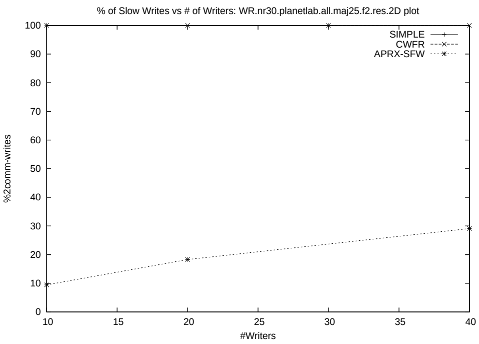


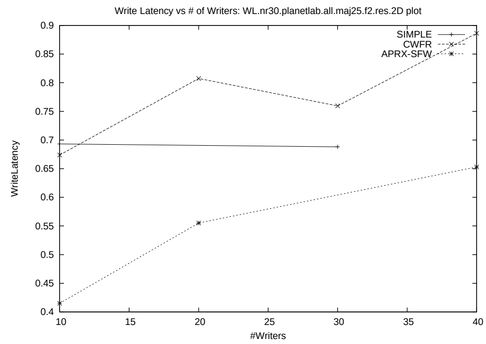
20 Readers:



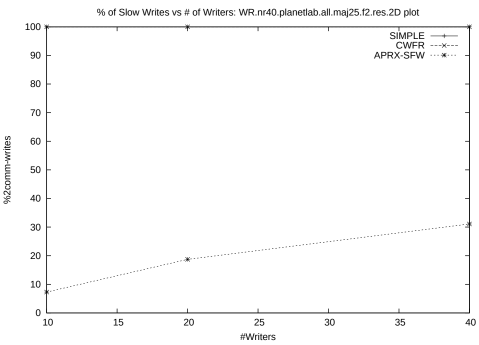


30 Readers:





40 Readers:



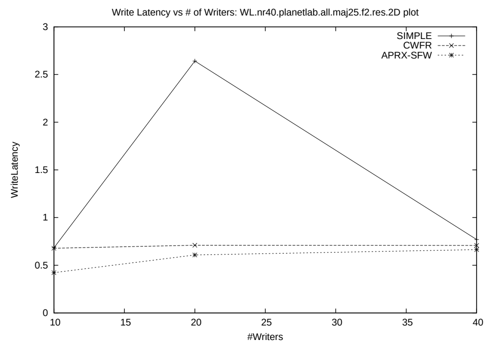
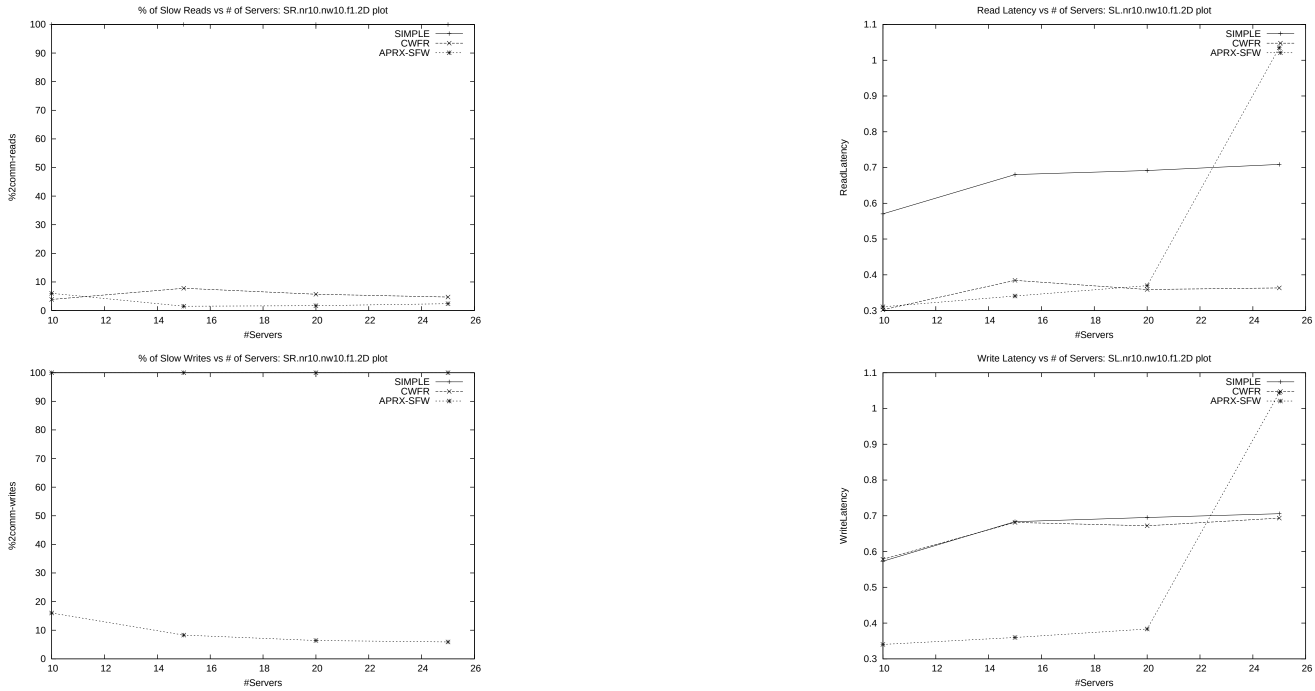


Figure 21: 11-wise quorum system ($|\mathcal{S}| = 25,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

C Operation Performance Relative to Quorum Construction

The plots below illustrate the performance of read operations over the number of quorums in the system. For the following graphs we fix the number of reader and writer participants and we change the number of servers in the system. The quorum membership is also changed when we modify the maximum number of server failures in the service. The left column of each figure presents how the percentage of slow reads is affected whereas the right column illustrates the impact of the quorum membership on read latency.

10 Readers, 10 Writers, f=1:



10 Readers, 10 Writers, f=2:

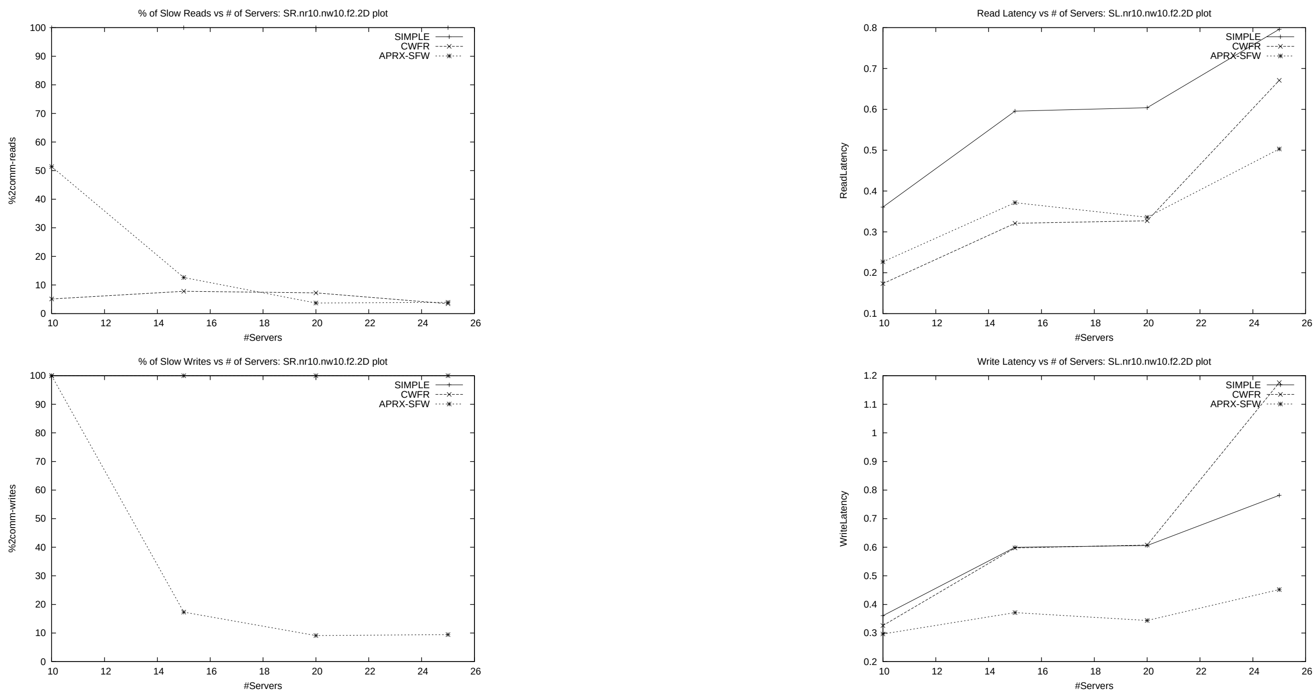
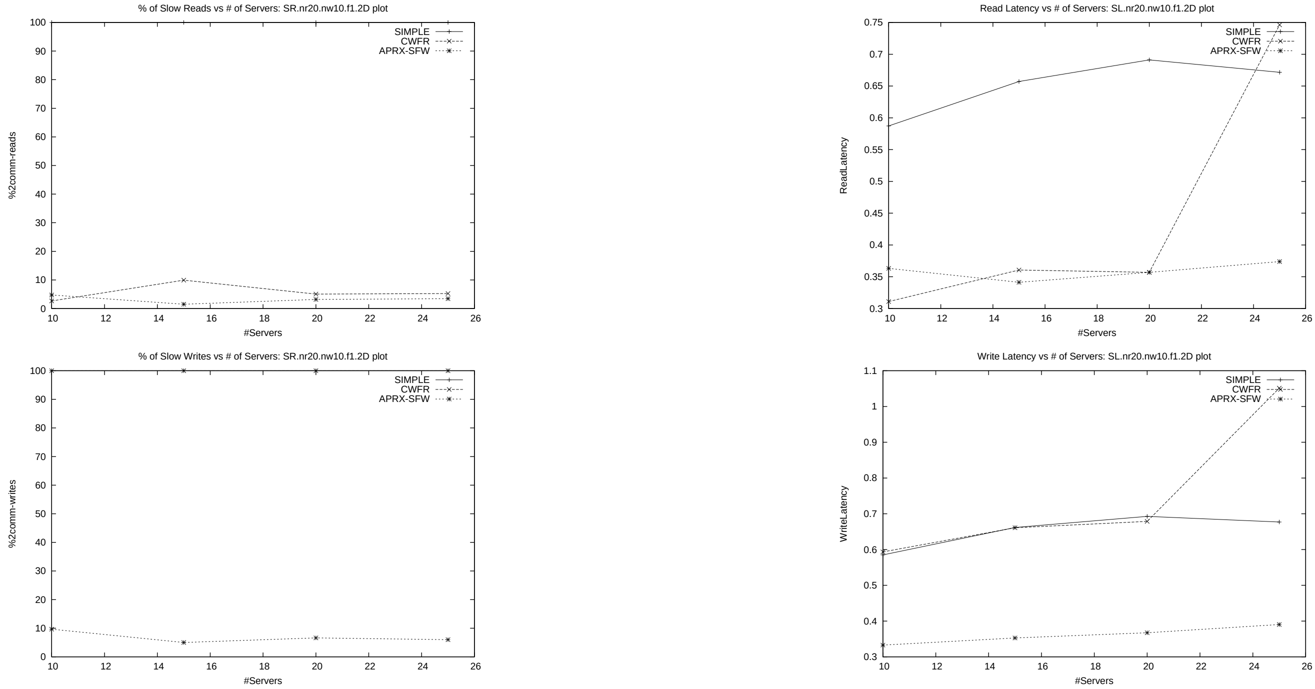


Figure 22: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 10 Writers, f=1:



20 Readers, 10 Writers, f=2:

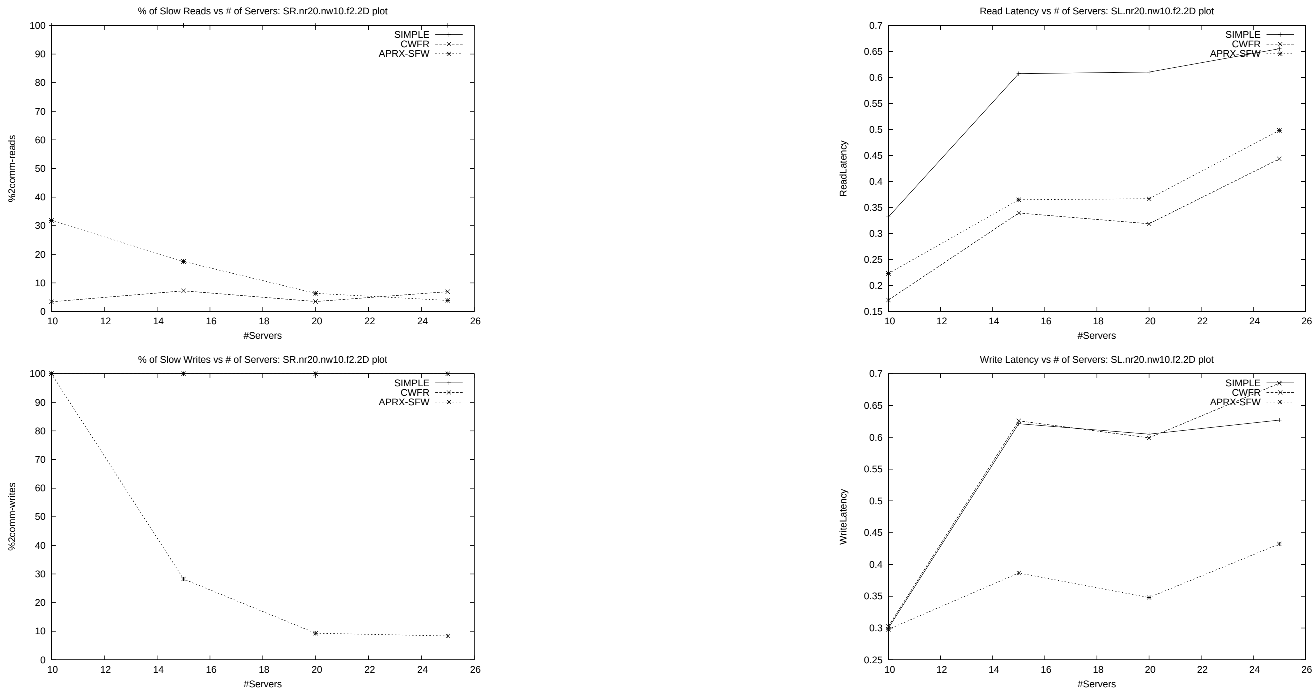
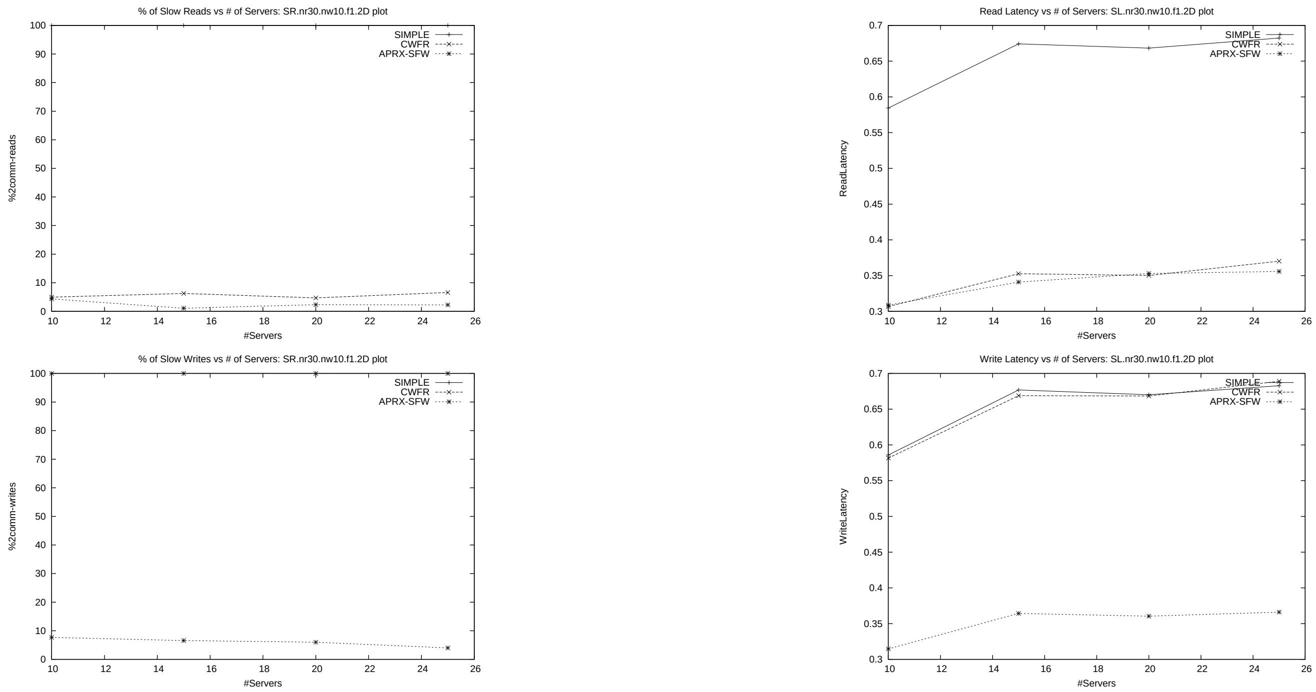


Figure 23: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

30 Readers, 10 Writers, f=1:



30 Readers, 10 Writers, f=2:

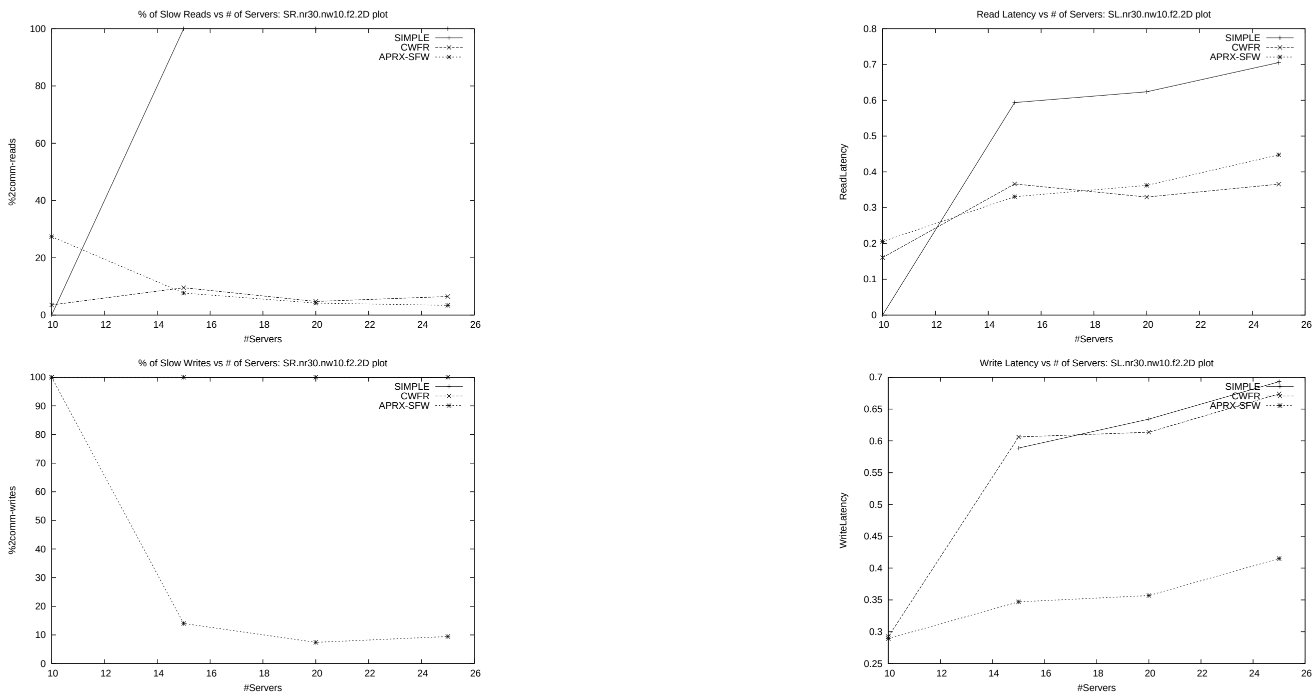
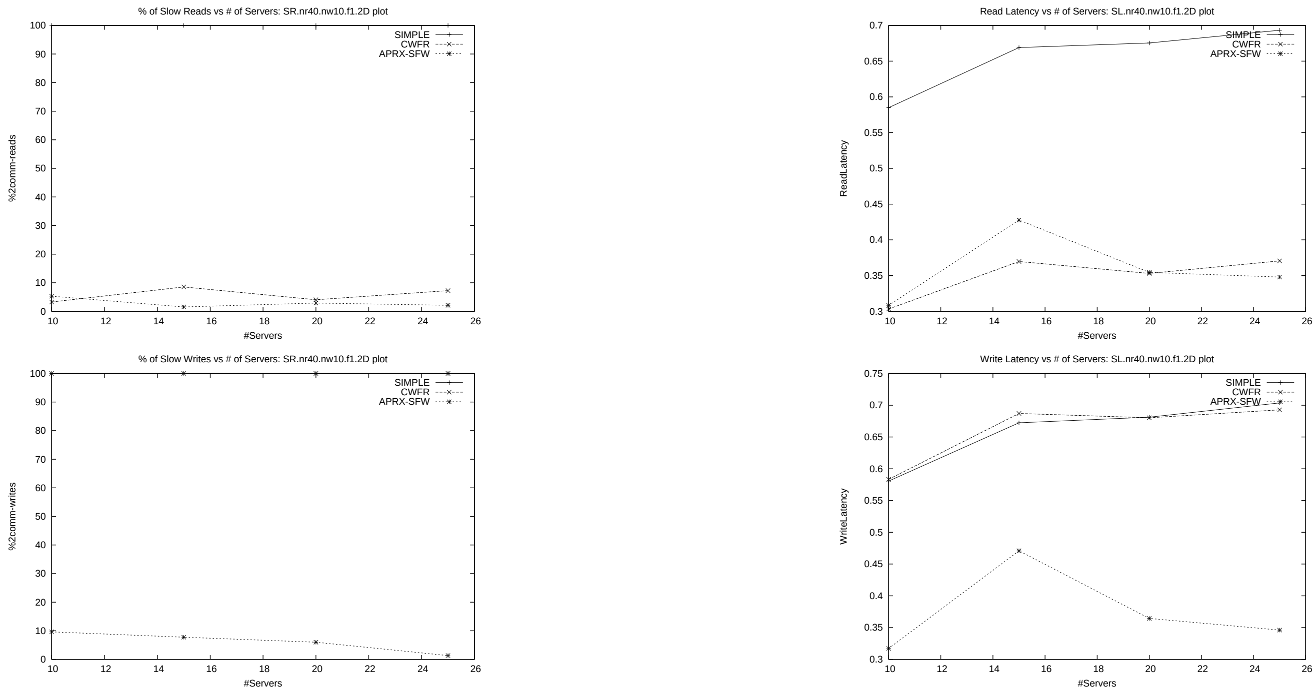


Figure 24: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 10 Writers, f=1:



40 Readers, 10 Writers, f=2:

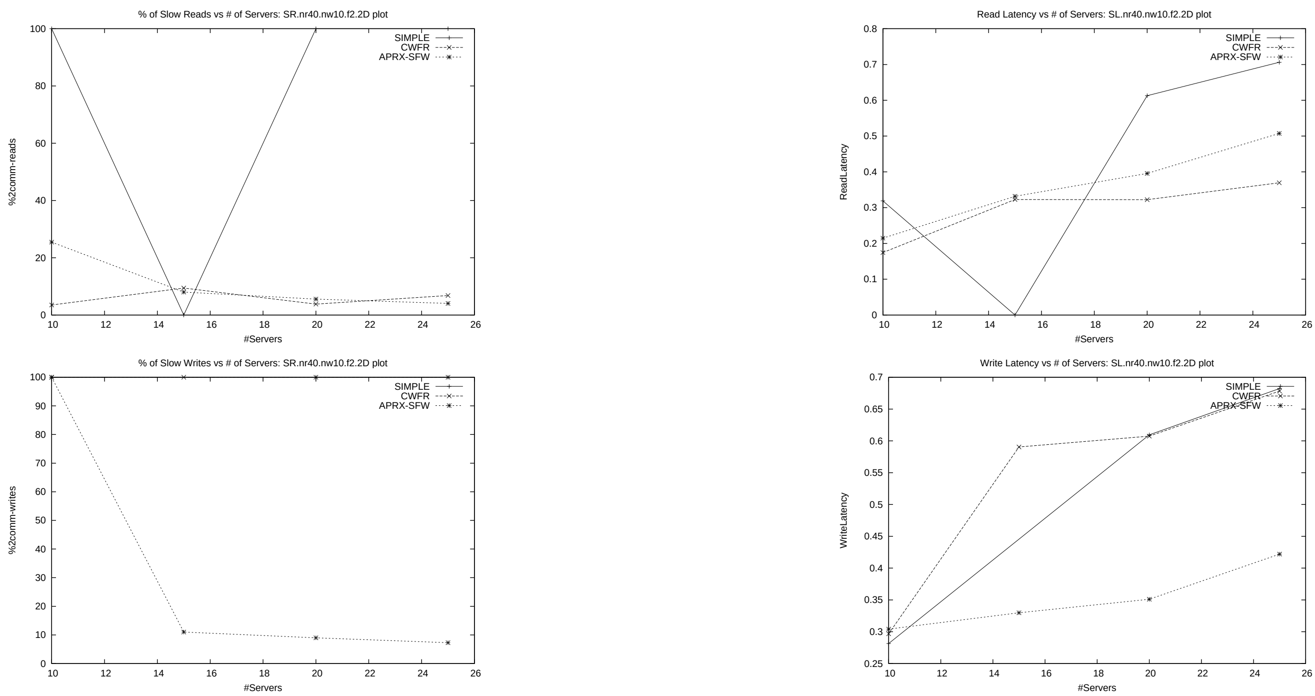
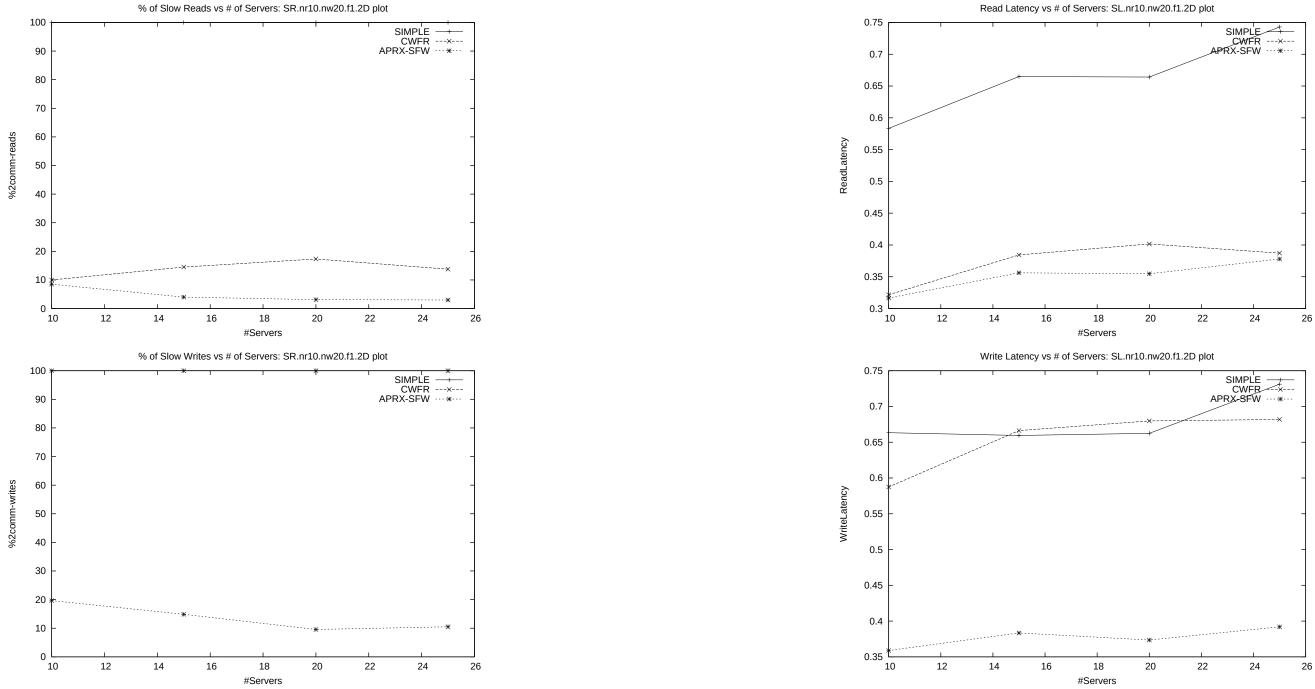


Figure 25: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

10 Readers, 20 Writers, f=1:



10 Readers, 20 Writers, f=2:

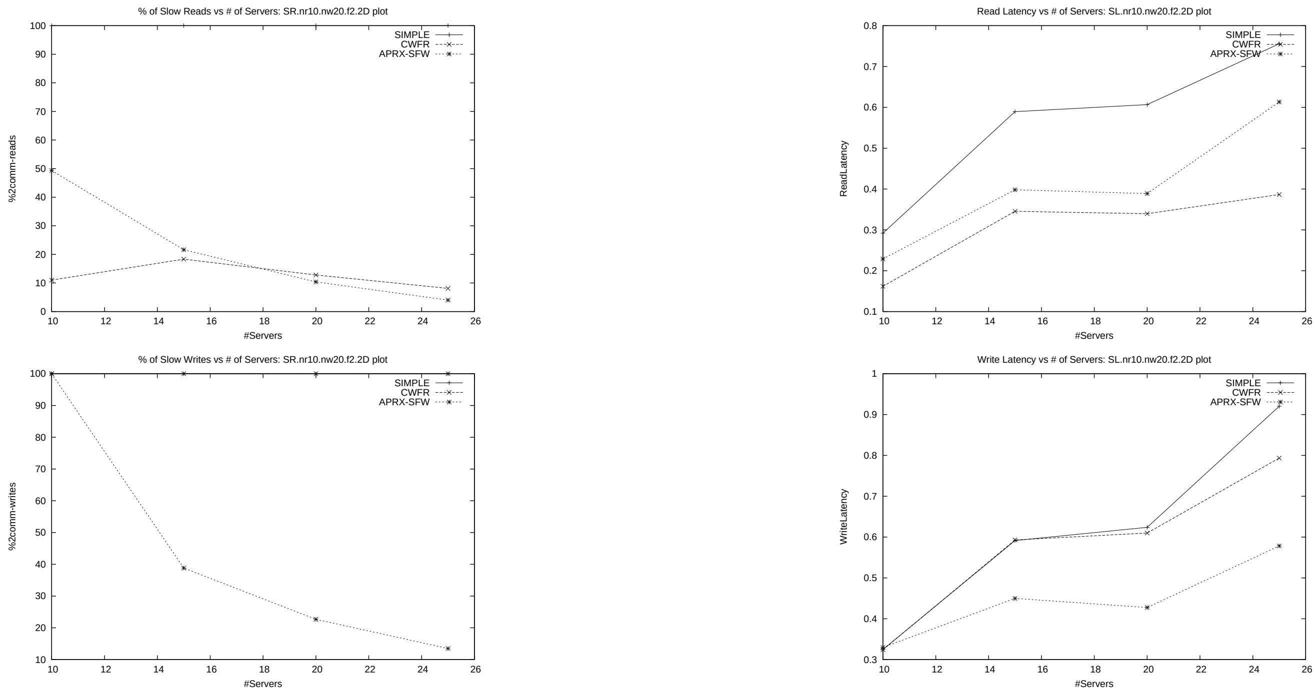
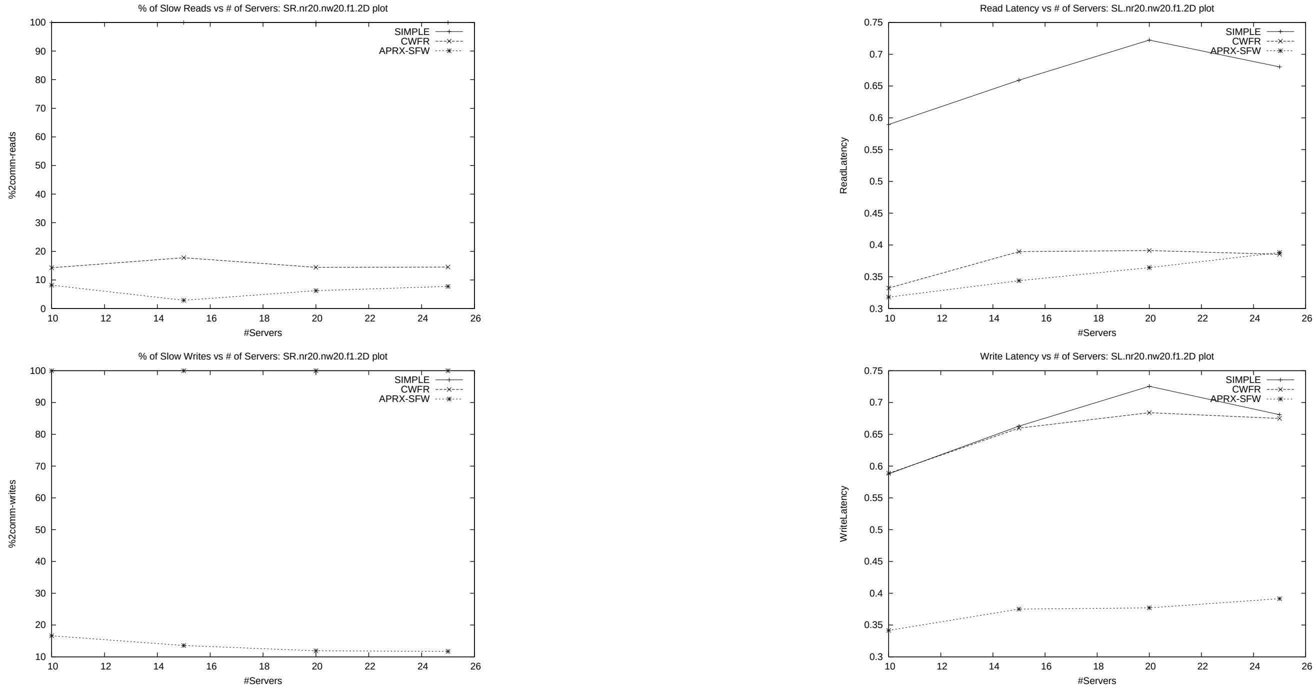


Figure 26: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 20 Writers, f=1:



20 Readers, 20 Writers, f=2:

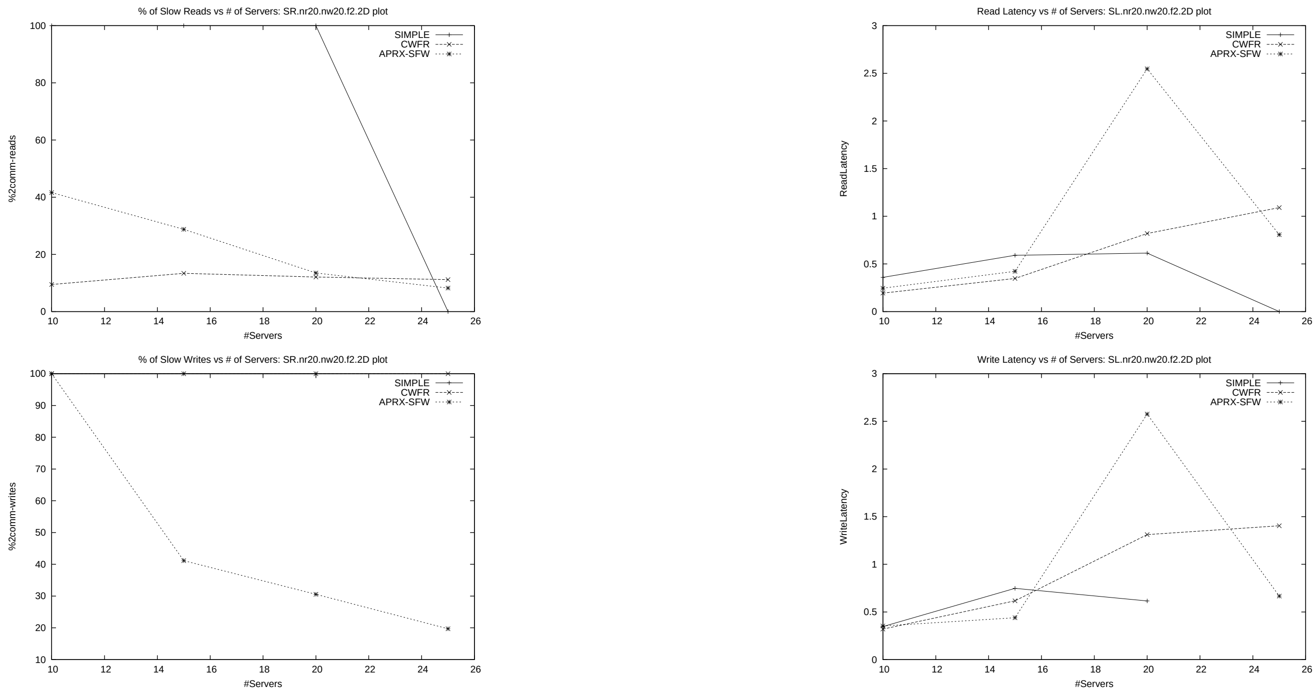
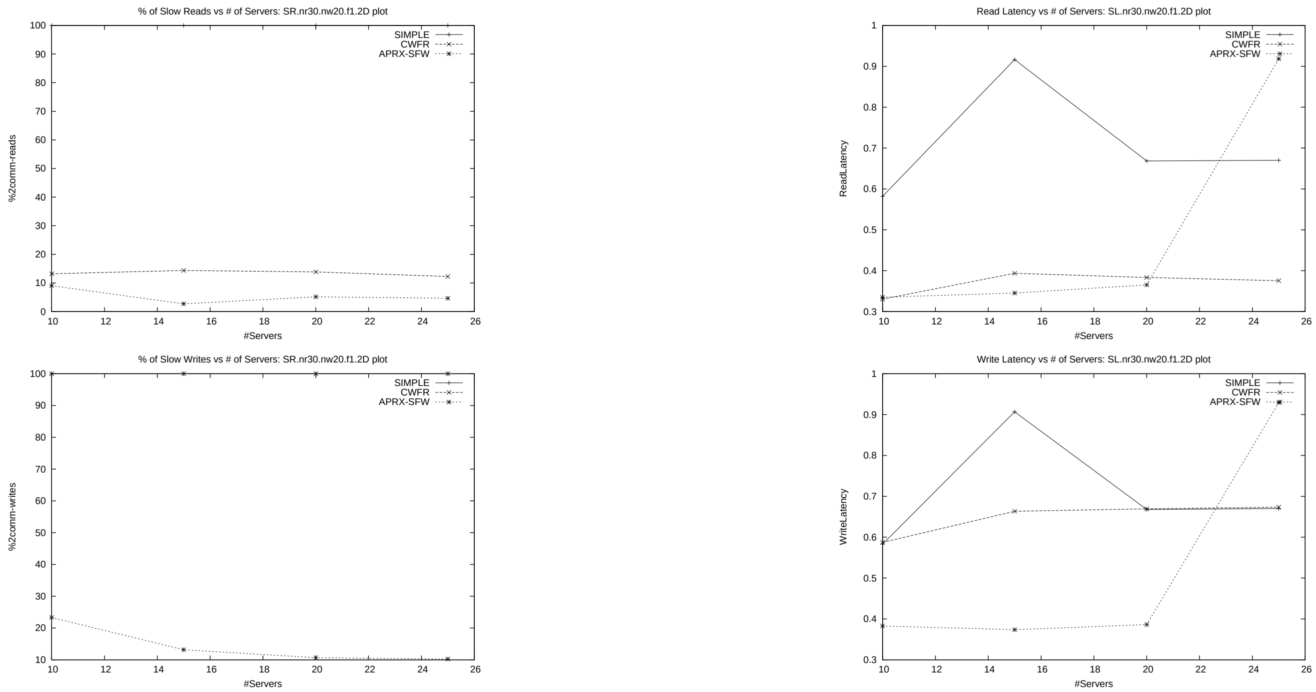


Figure 27: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

30 Readers, 20 Writers, f=1:



30 Readers, 20 Writers, f=2:

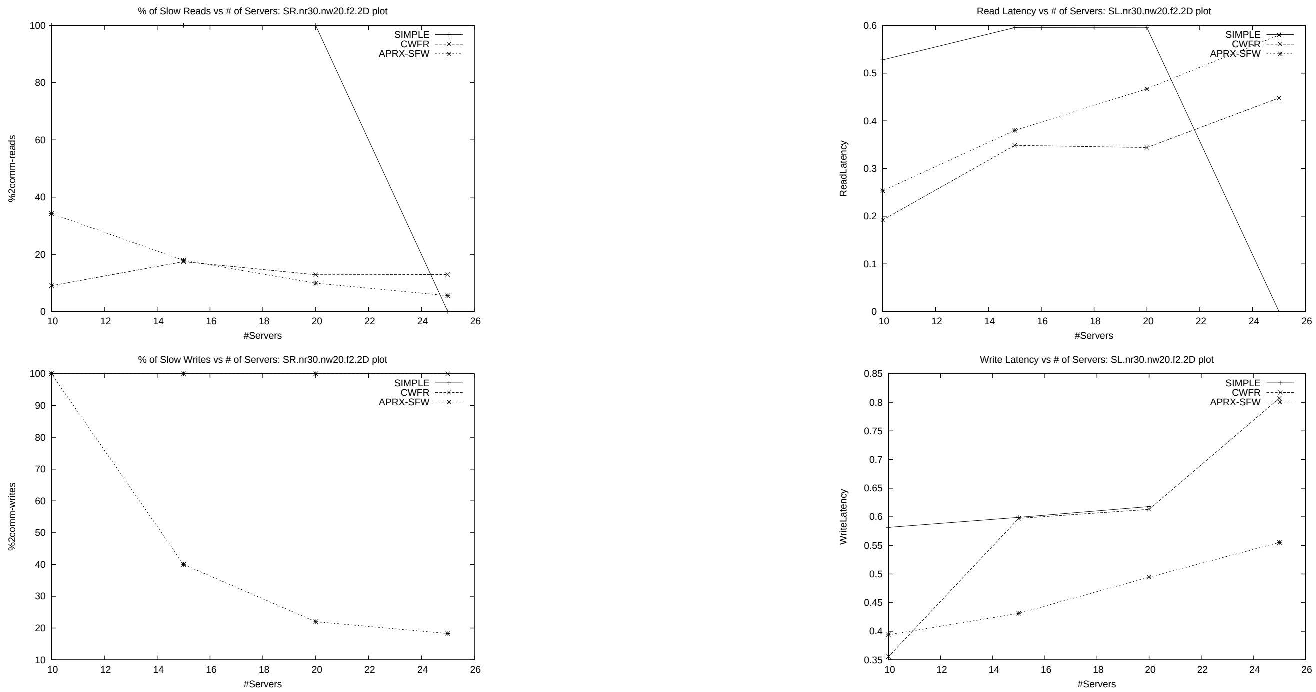
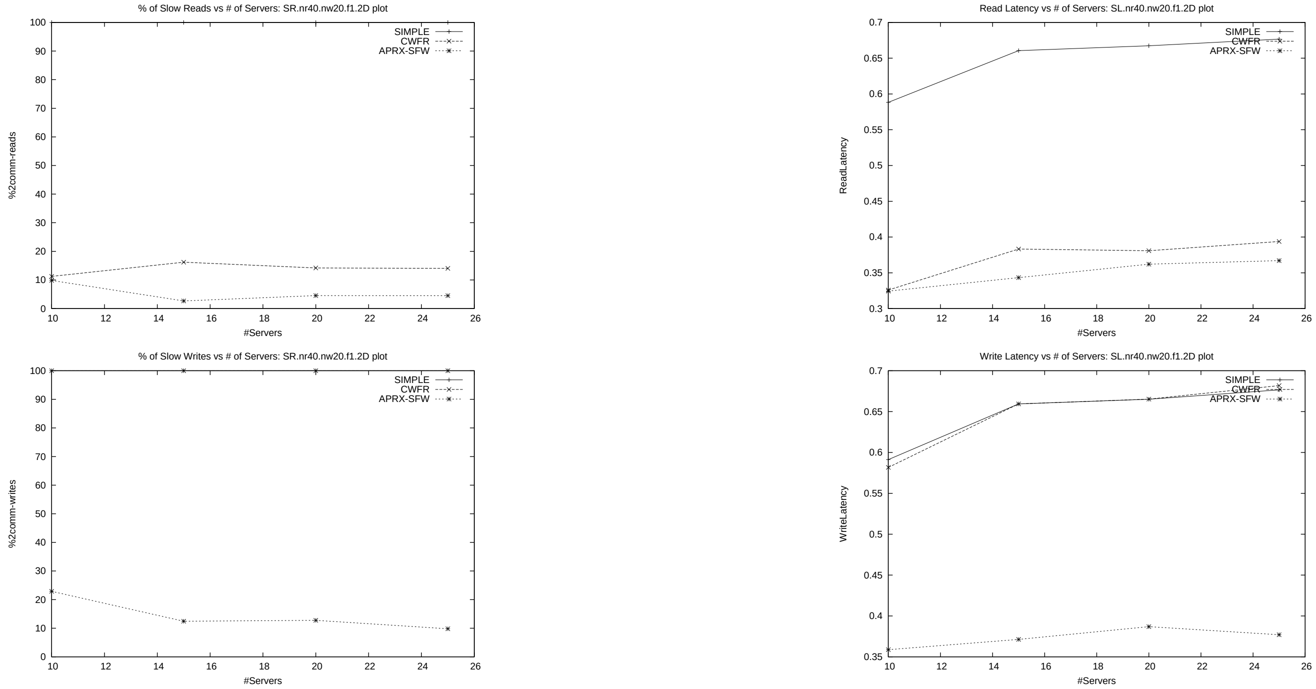


Figure 28: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 20 Writers, f=1:



40 Readers, 20 Writers, f=2:

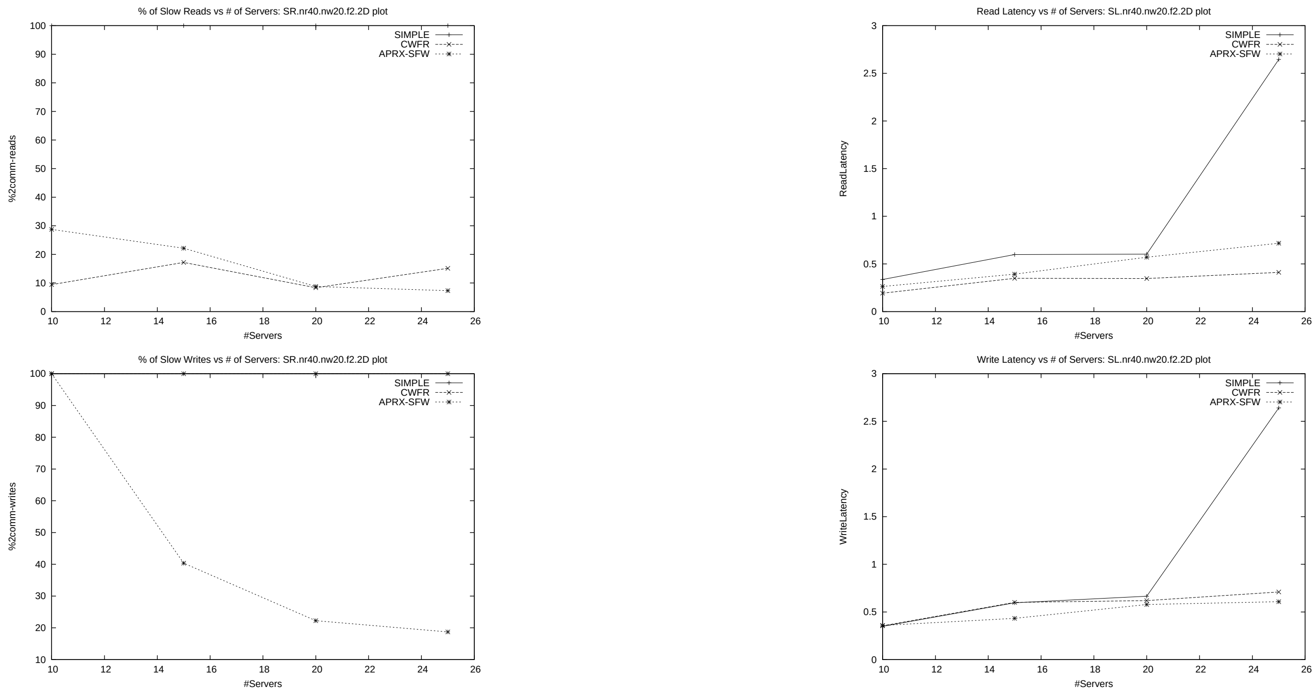
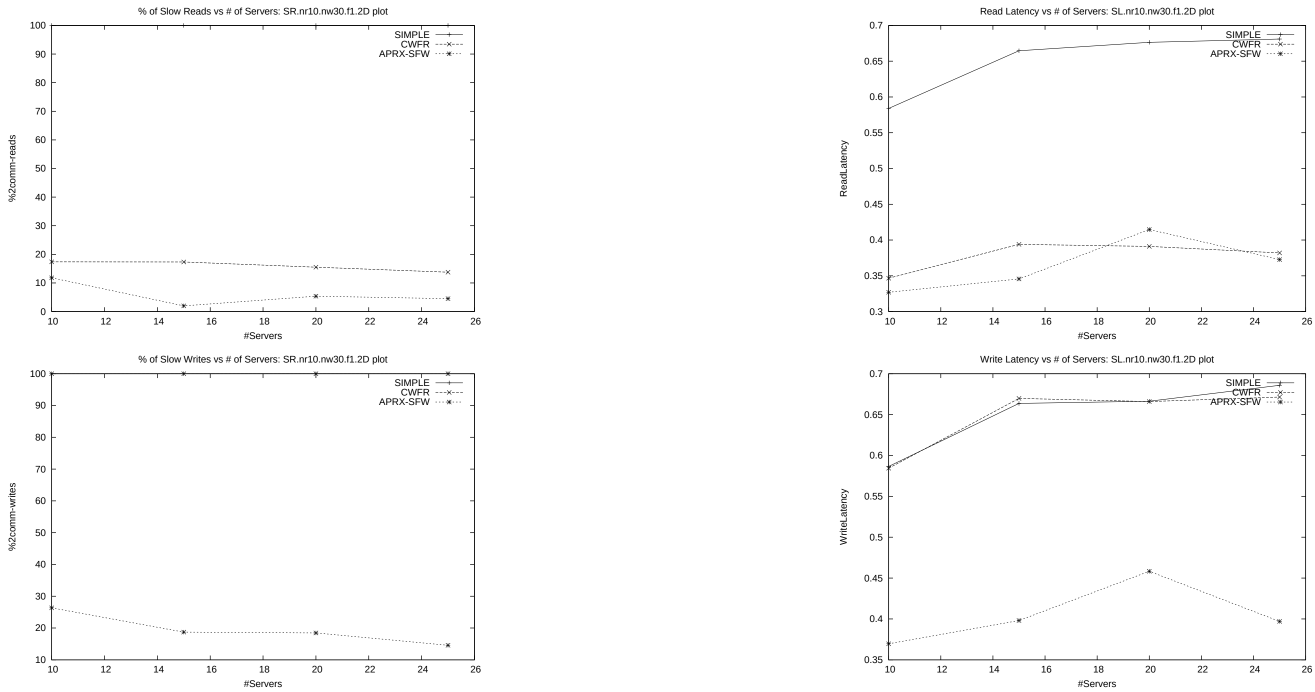


Figure 29: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

10 Readers, 30 Writers, f=1:



10 Readers, 30 Writers, f=2:

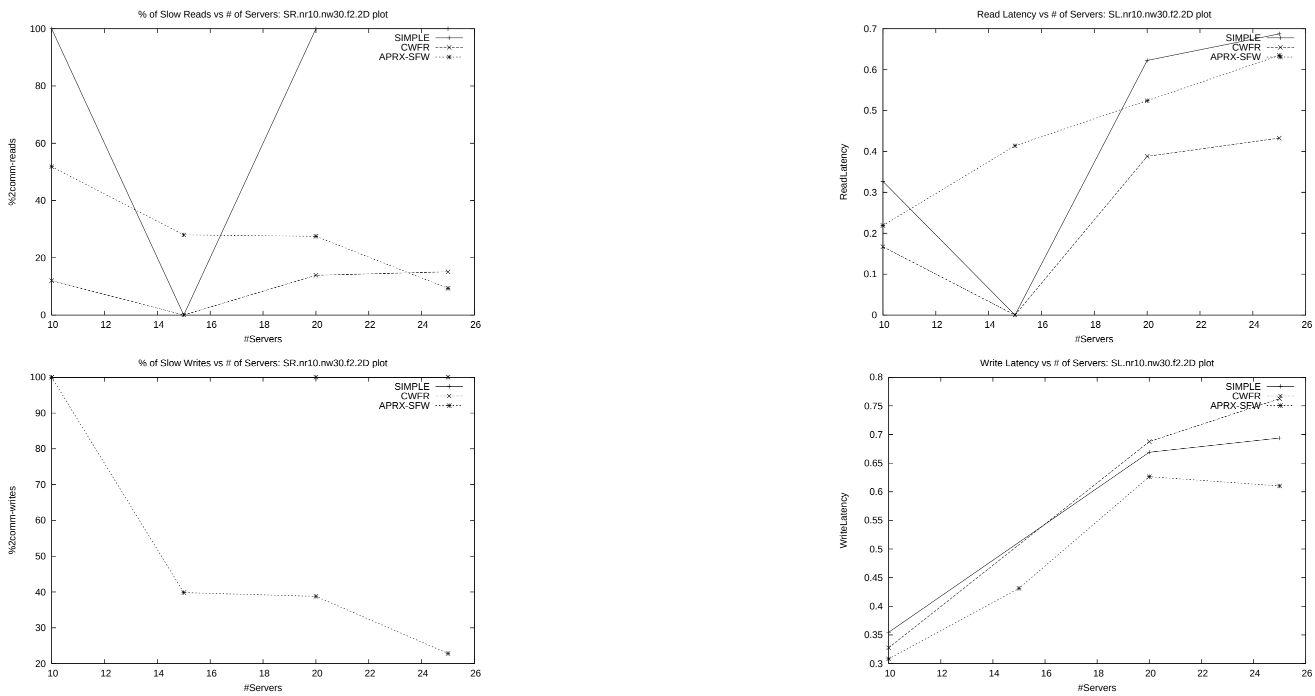
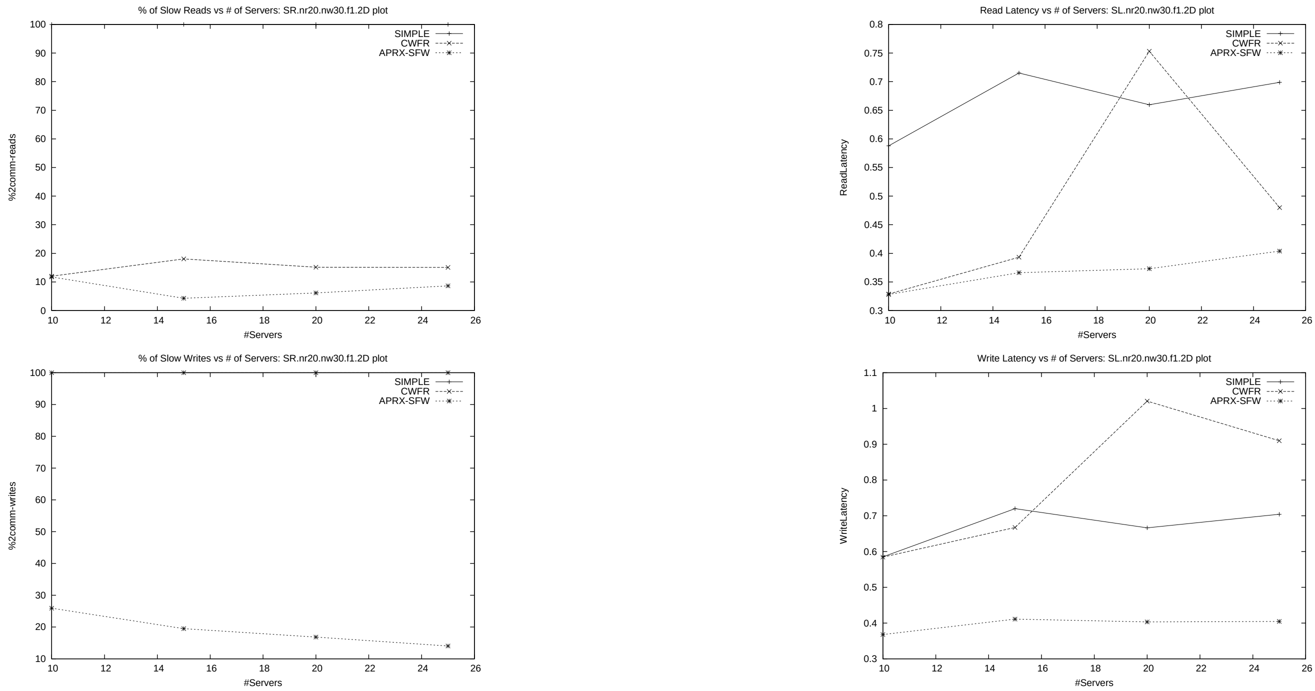


Figure 30: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 30 Writers, f=1:



20 Readers, 30 Writers, f=2:

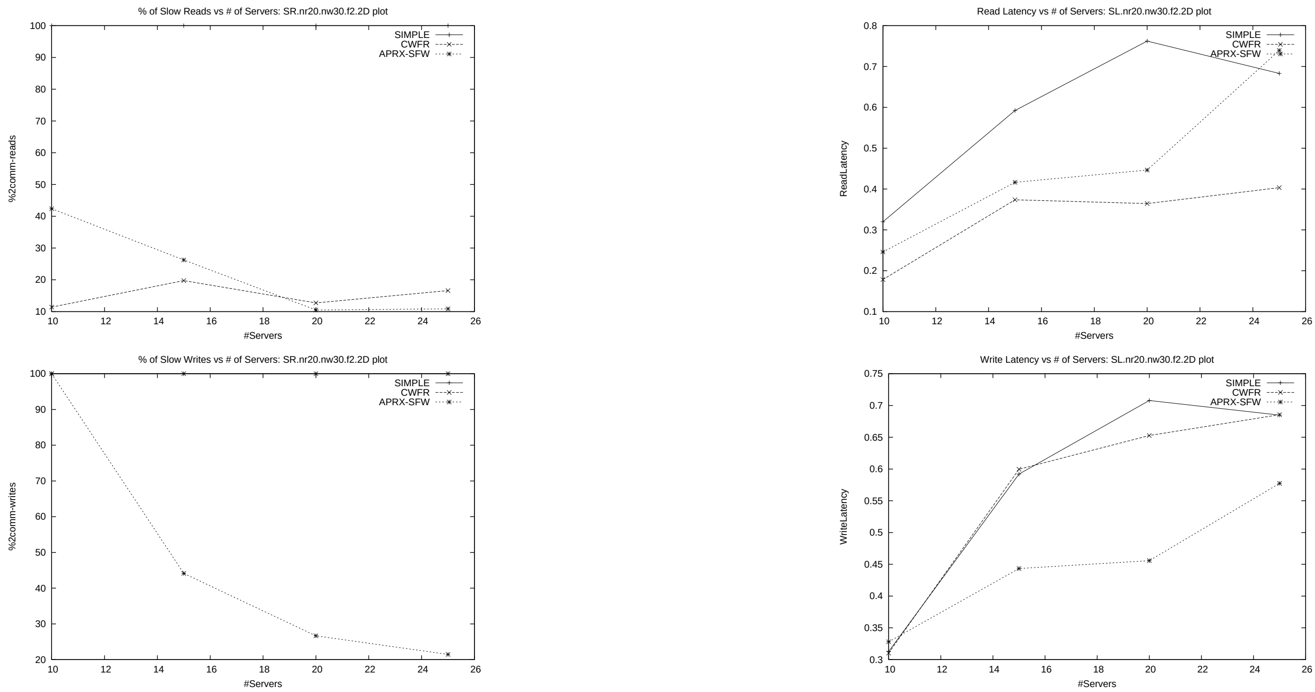
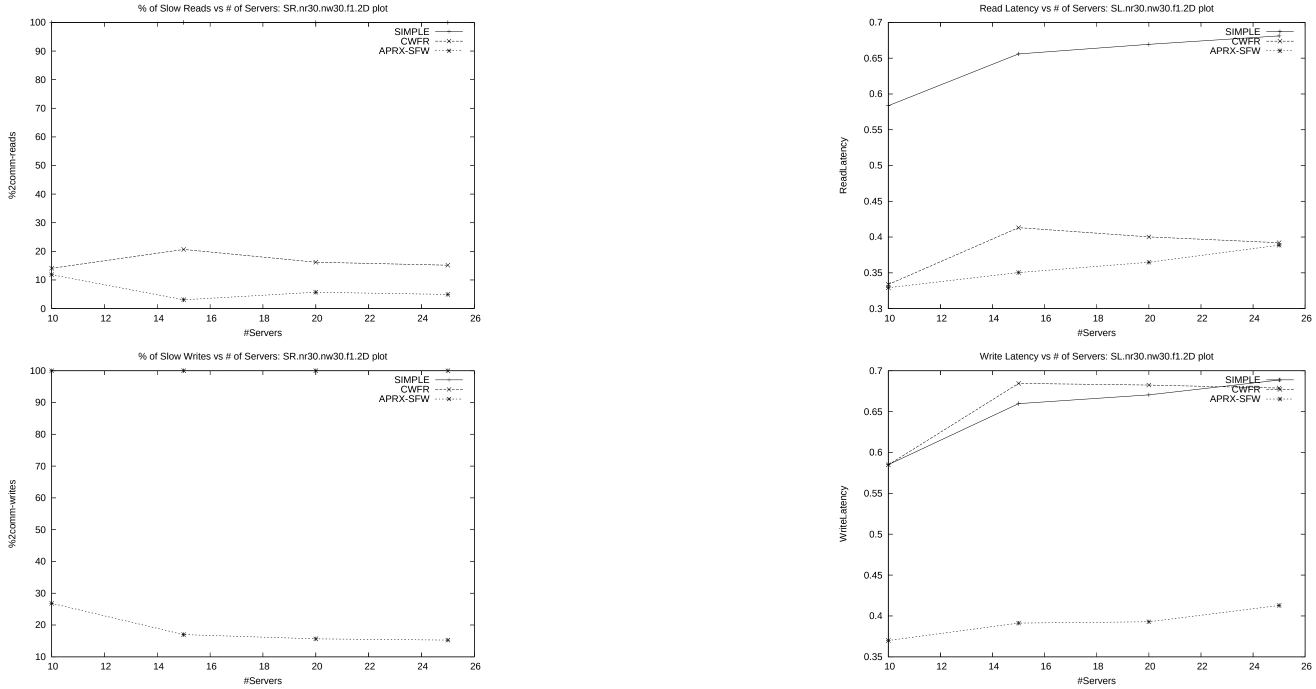


Figure 31: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

30 Readers, 30 Writers, f=1:



30 Readers, 30 Writers, f=2:

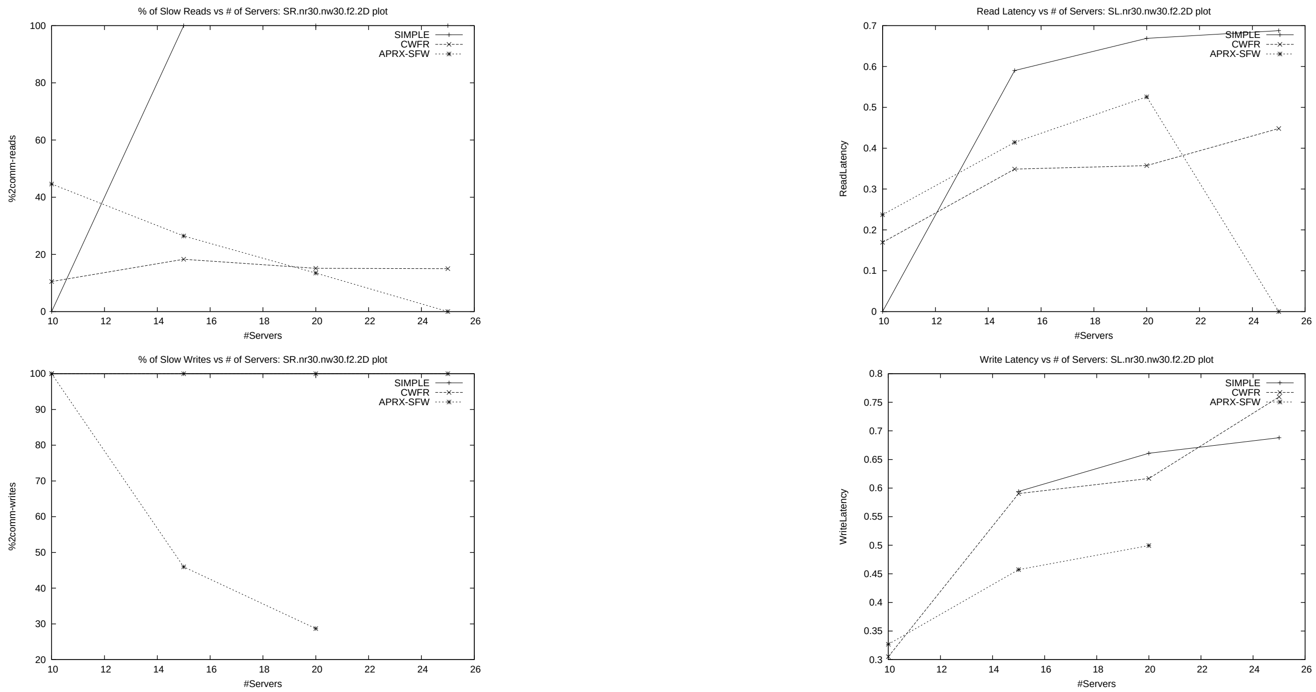
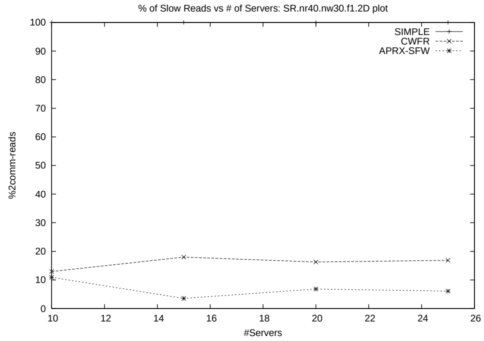
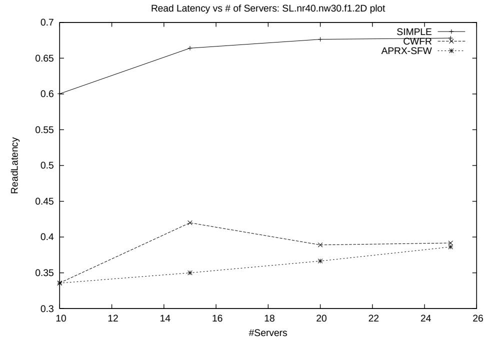
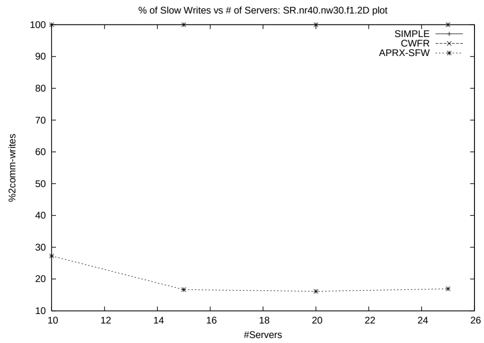


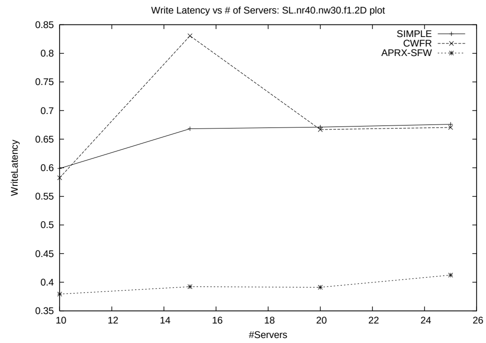
Figure 32: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 30 Writers, f=1:

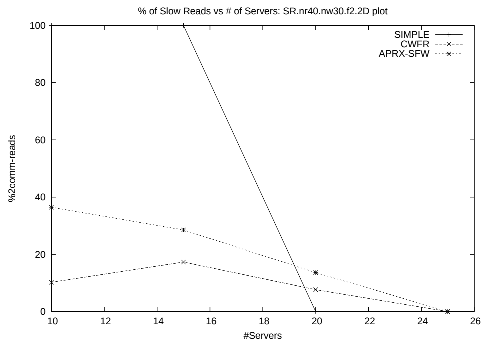


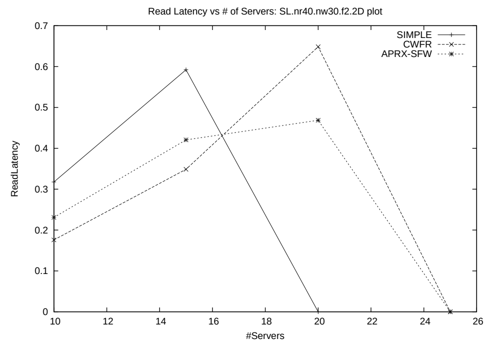


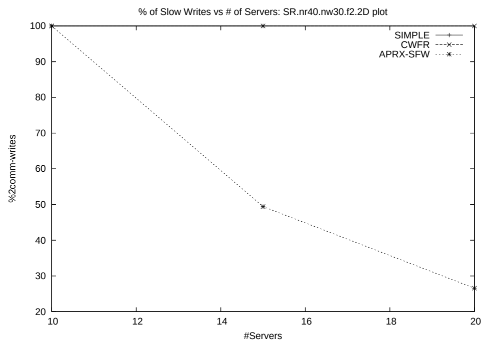




40 Readers, 30 Writers, f=2:







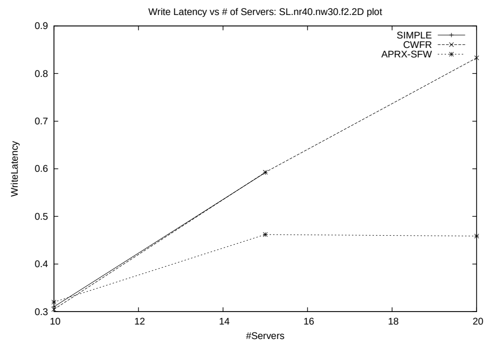
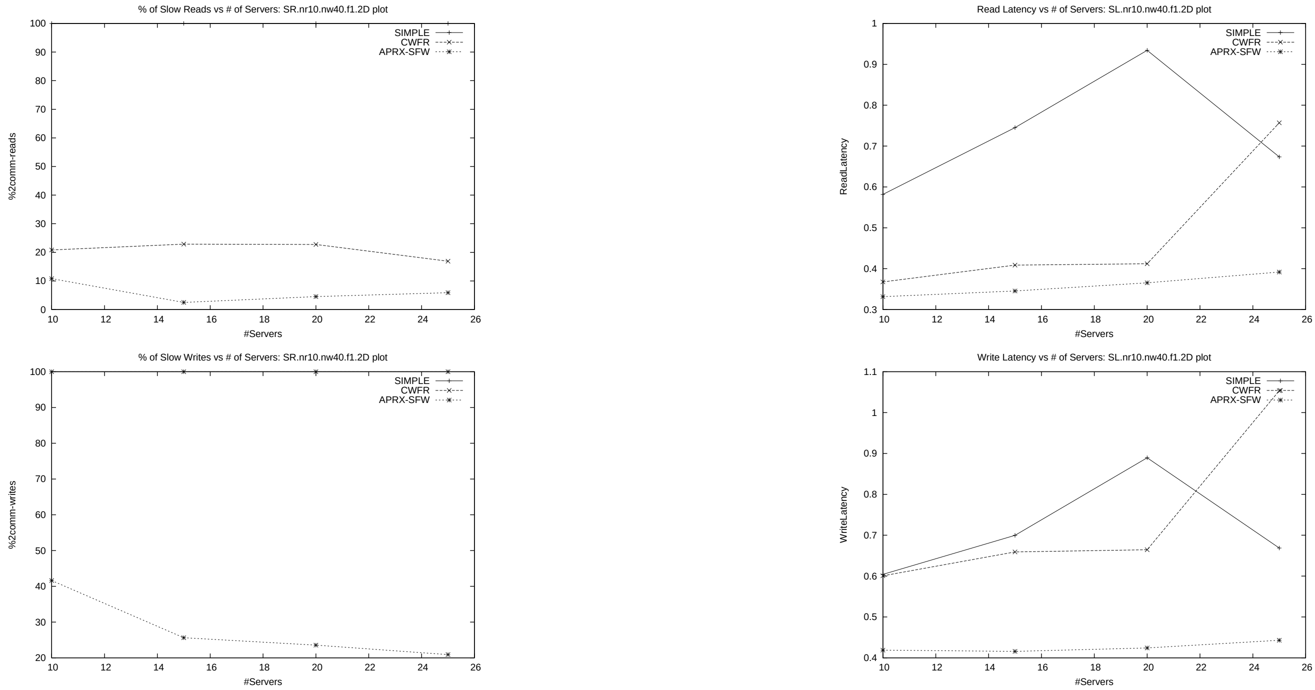


Figure 33: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

10 Readers, 40 Writers, f=1:



10 Readers, 20 Writers, f=2:

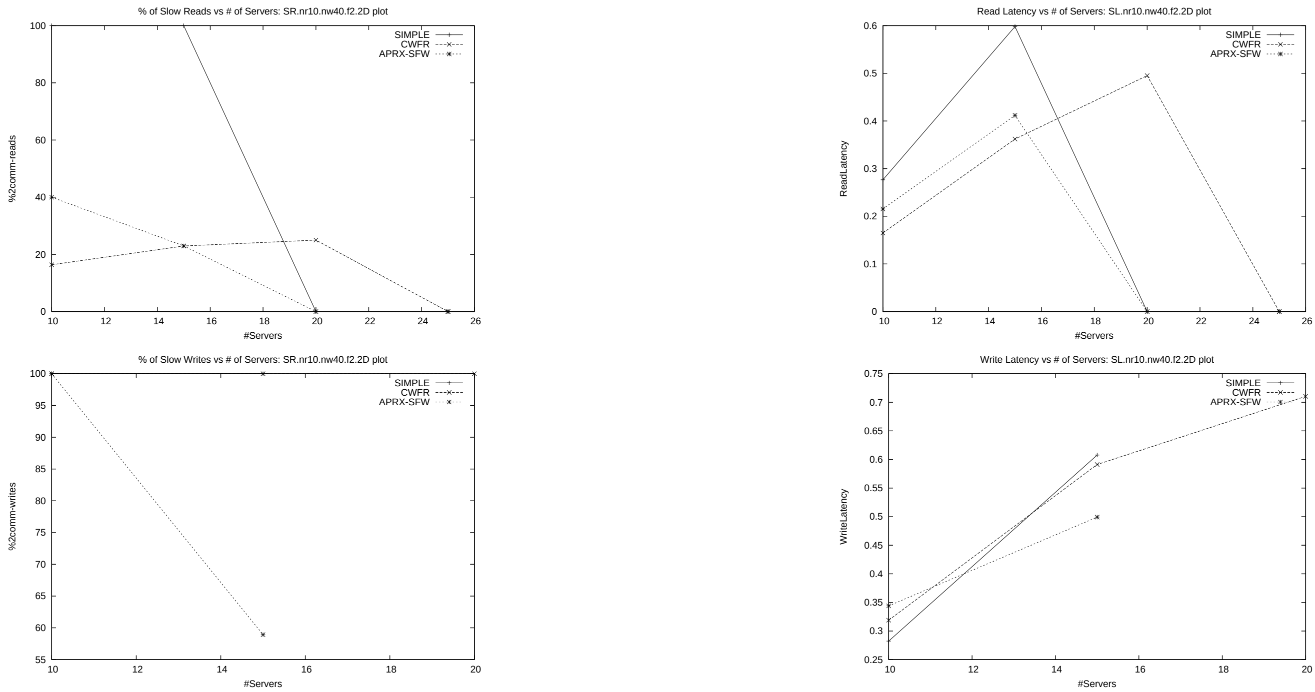
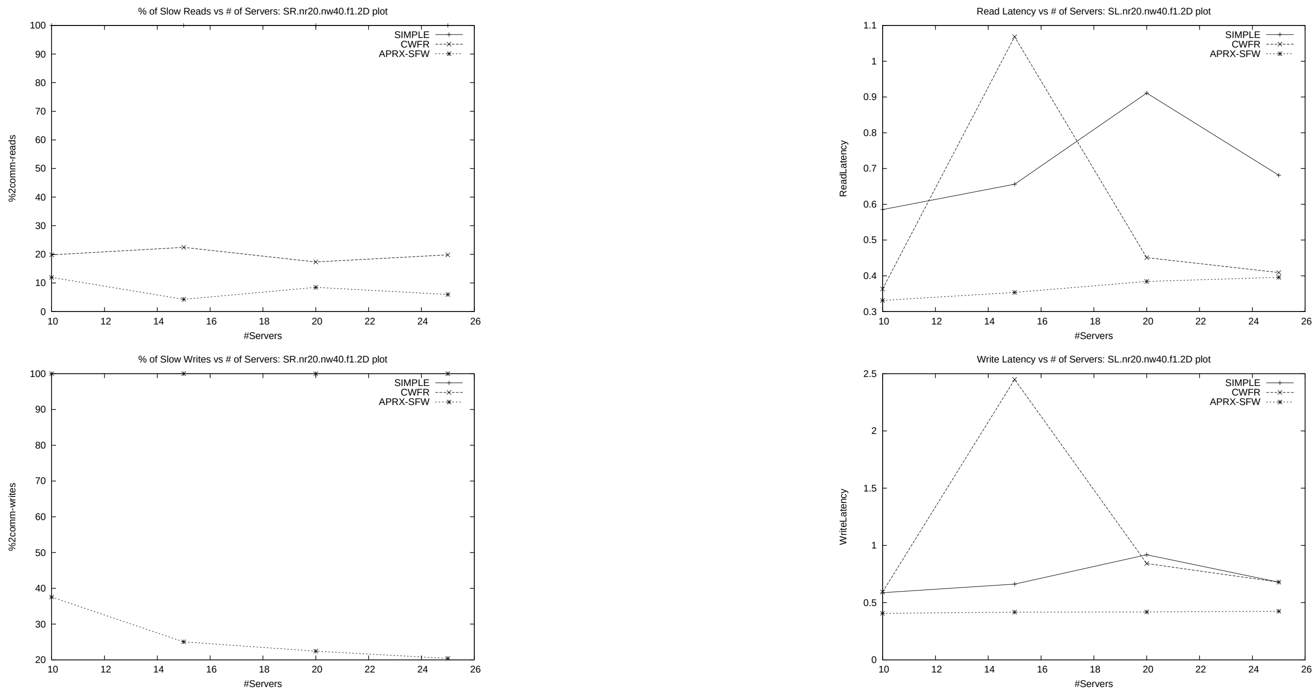


Figure 34: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 40 Writers, f=1:



20 Readers, 40 Writers, f=2:

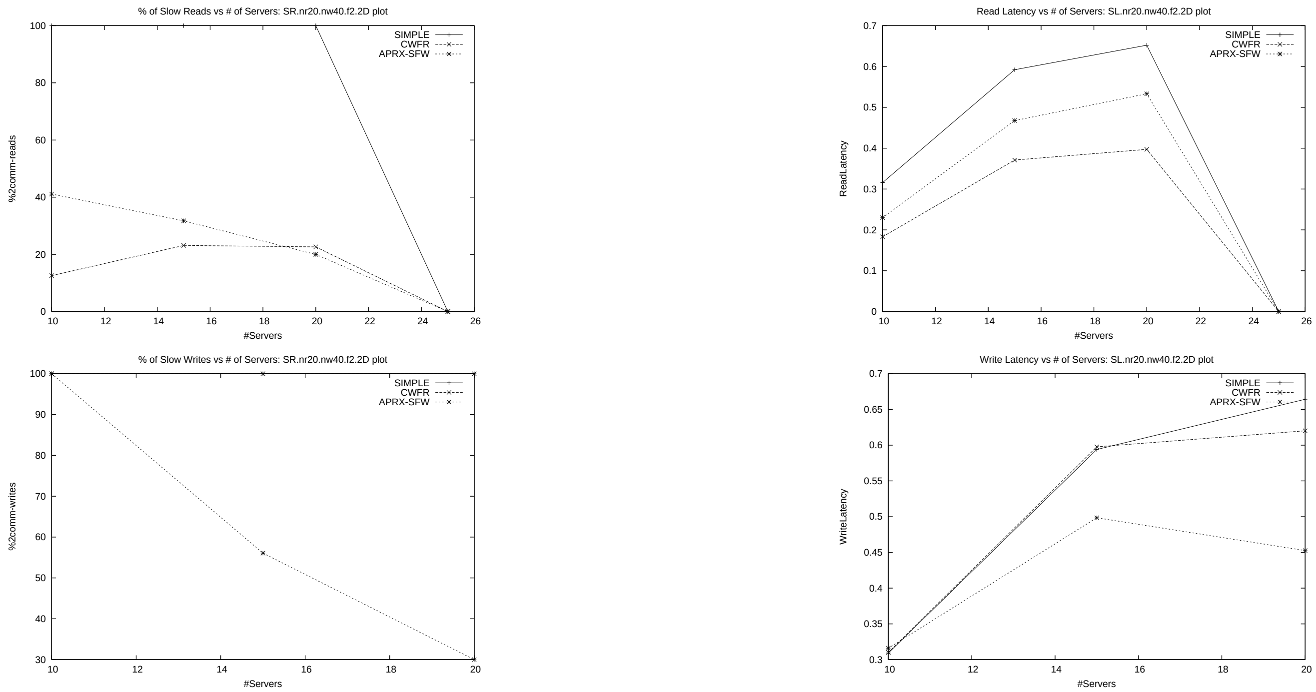
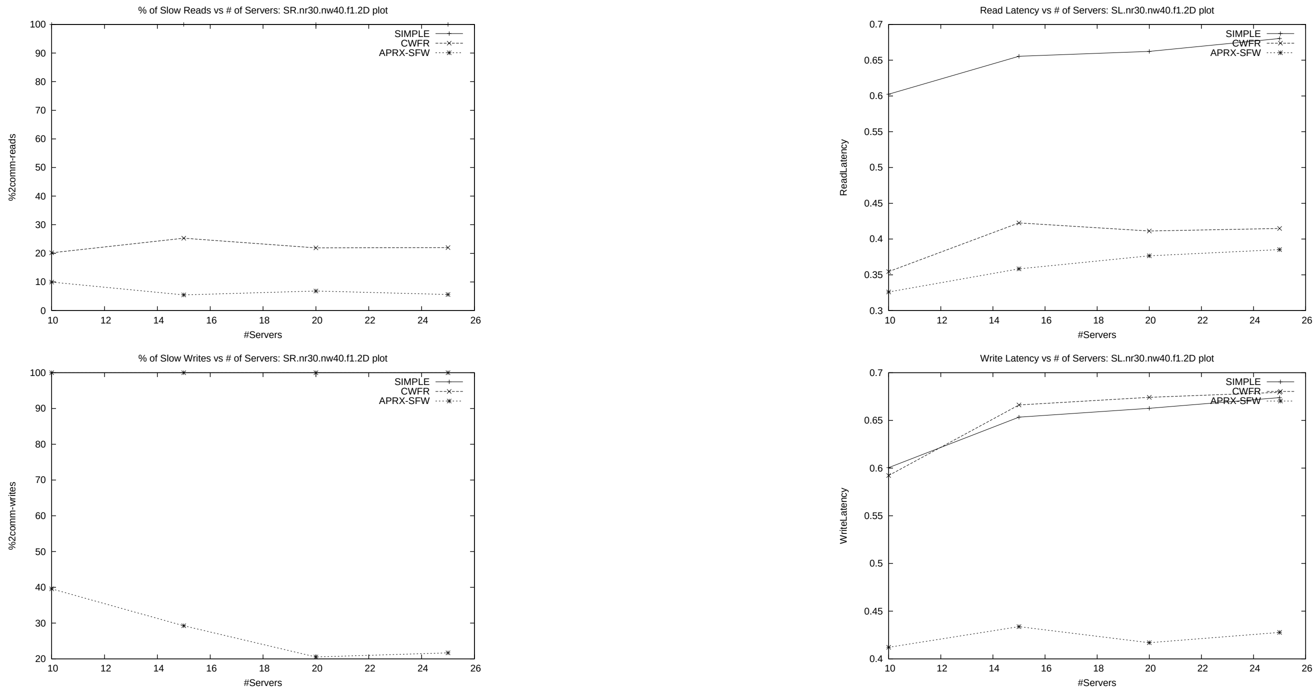


Figure 35: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

30 Readers, 40 Writers, f=1:



30 Readers, 40 Writers, f=2:

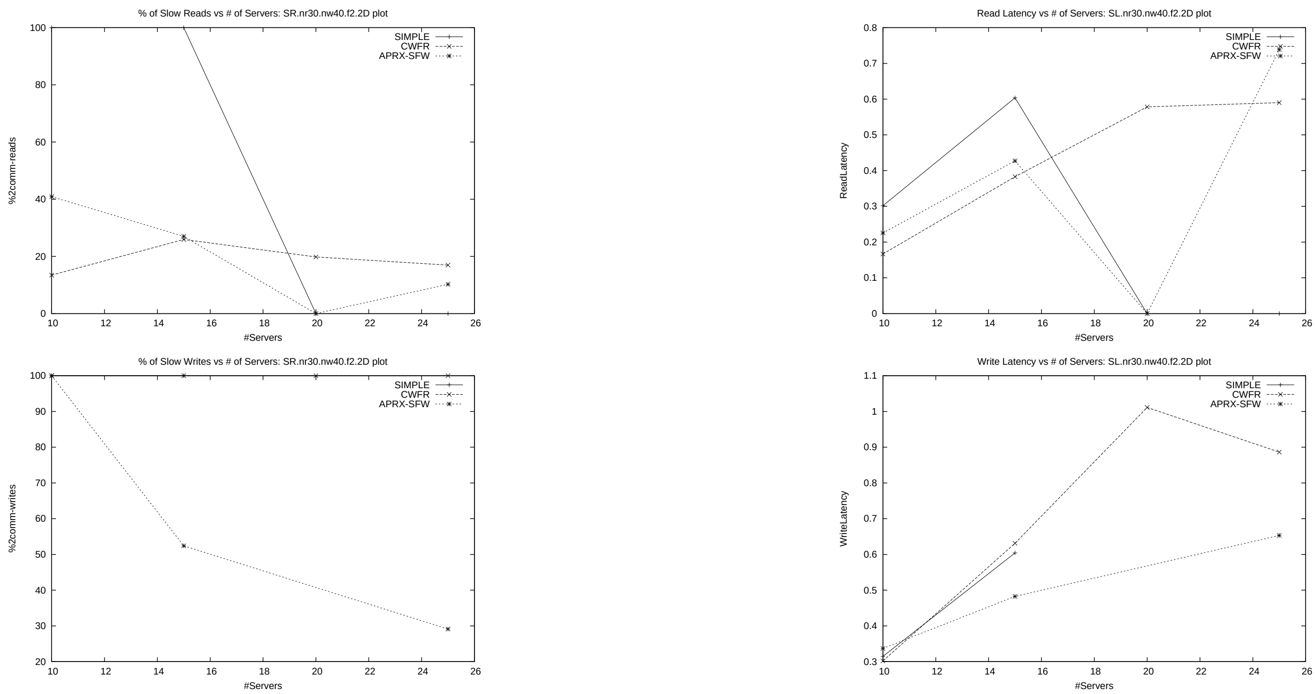
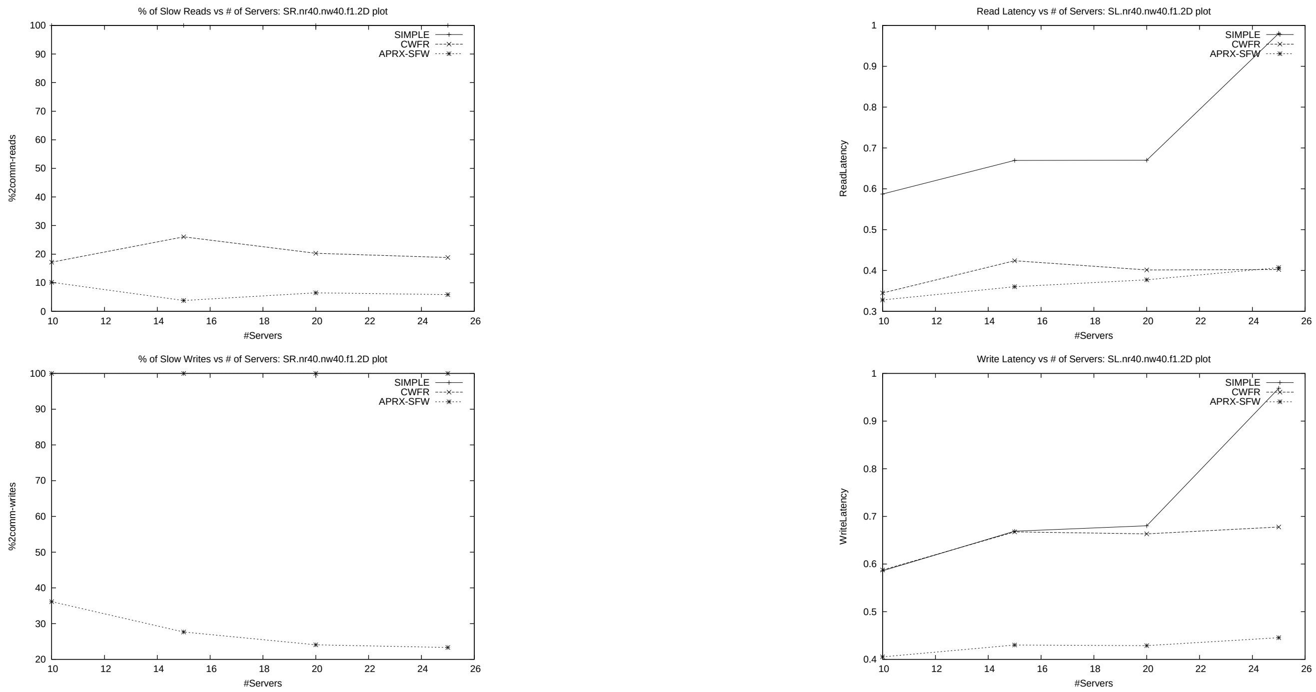


Figure 36: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 40 Writers, f=1:



40 Readers, 40 Writers, f=2:

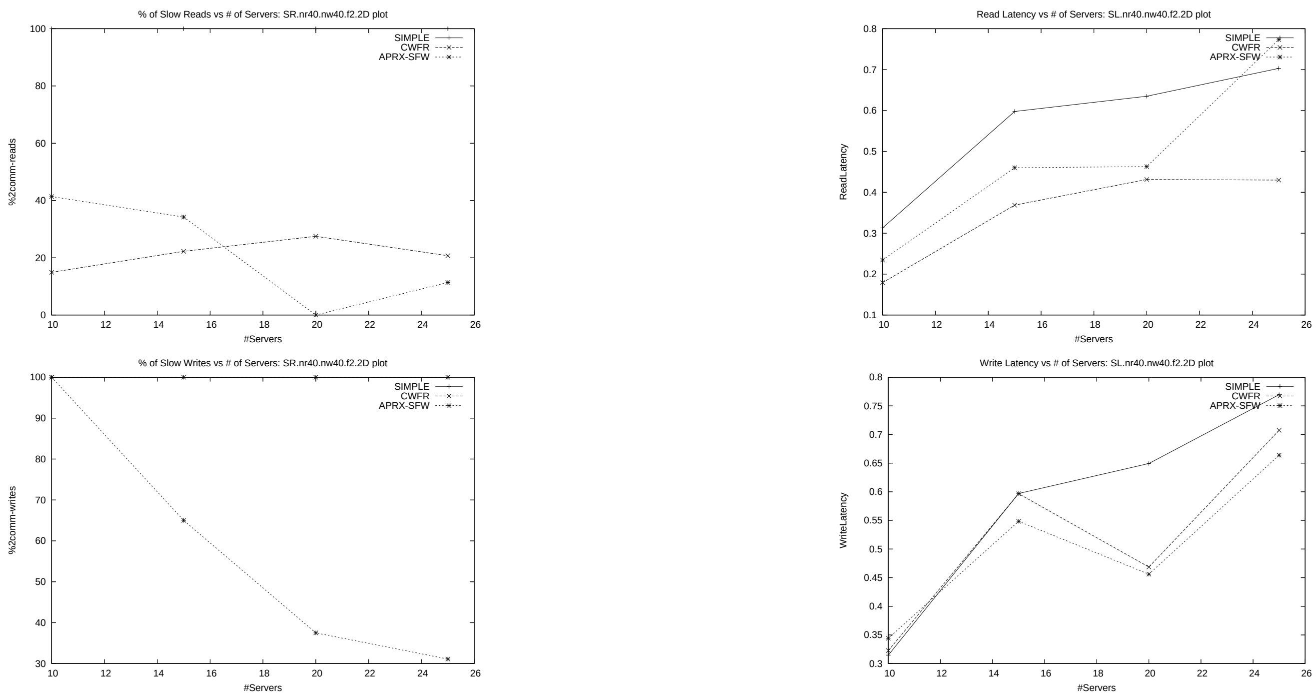


Figure 37: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations