Simulating Efficient MWMR Atomic Register Implementations on the NS2 Network Simulator *

Chryssis Georgiou †   Nicolas C. Nicolaou† ‡

Abstract

This work evaluates experimentally the performance of four multiple-writer/multiple-reader (MWMR) read/write atomic register implementations, designed for the asynchronous, crash-prone, message passing environment. 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. For our experiments we implemented the four algorithms in the form of protocols that are embedded and used through the NS2 network simulator. For our experiments we consider the following algorithms: Sfw from [7], Aprx-Sfw and CwFr from [10], and the generalization of the traditional algorithm of [3] in the MWMR environment, which we call Simple. Aprx-Sfw is a polynomial approximation algorithm for Sfw. So we first perform a proof-of-concept comparison of the two algorithms. Then, we compare algorithms Aprx-Sfw, CwFr, and Simple. Due to its simplistic nature, Simple requires two communication round-trips per read or write operation, but almost no local computation. On the other hand, Aprx-Sfw and CwFr allow operations to complete in a single round-trip, but they rely on computationally demanding predicates. Through our comparison we attempt to identify the trade-offs between communication and computation burdens for these three algorithms. We run our simulations under various scenarios that help us obtain a comprehensive comparison for the tested algorithms.

Technical Report TR-11-06
Department of Computer Science
University of Cyprus
May 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.

‡Department of Computer Science and Engineering, University of Connecticut, CT, USA.

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. [14] 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 problem of computing the predicates used by the algorithm is 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* [13], 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.

In this work we evaluate the latency of the afforementioned algorithms, by simulating them in the NS2 network simulator.

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., [18, 8, 17, 9, 15]). 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. [14] 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 imposibility of semifast MWMR implementations.

Georgiou et al. [13] 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 [13] 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 efficiency of MWMR atomic register implementations. We focus on a controlled simulation environment, provided by the NS2 [2] simulator, and we implement and compare the following MWMR atomic register algorithms: Sfw, Simple, Aprx-Sfw, and CwFr. The operation latency of the algorithms is tested under different simulation scenarios: (1) Variable number of readers/writers/servers, (2) Deployment of different quorum constructions, and (3) Variable network delay.

A general conclusion from our findings is that algorithms CwFr and Aprx-Sfw over-perform algorithm Simple in most of the scenarios. Also, increasing the number of participants has a negative impact on the latency of the operations for all the algorithms but especially for Aprx-Sfw. Finally, we observe that a long network delay promotes algorithms with high computational demands and fewer communication rounds (such as algorithm Aprx-Sfw).

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 the NS2 simulator, we present our testbed and provide the scenarios we consider. Simulation results and comparisons of the algorithms are given in Section 5. Finally, in Section 6 we discuss our findings and our future research plans.

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 $\pi_1 \to \pi_2$. 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 [16]. 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, 14, 13]:

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 four algorithms we evaluate. 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 htag, valuei pairs where a tag contains a timestamp and the writers 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 $inprogress_s(\omega)$ 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: $\tau \in \{\langle ., \omega\rangle : \langle ., \omega\rangle \in inprogress_s(\omega) \wedge s \in Q\}$, $A \subseteq \mathbb{Q}, 0 \leq |A| \leq \frac{n}{2}-1$, and $MS = \{s : s \in Q \wedge \tau \in inprogress_s(\omega)\}$, 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:

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.

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 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 [13]. The algorithm employs two techniques:(i) the typical query and propagate approach (two rounds) for write operations, and (ii) analysis of Quorum Views [13] 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 communication and computation burdens of the four algorithms we consider. 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	$O(\|\mathcal{S}\|)$	$O(\|\mathcal{S}\|)$	Highest Tag
Sfw	1 or 2	1 or 2	$O(2^{\|\mathbb{Q}\|-1})$	$O(2^{\|\mathbb{Q}\|-1})$	Predicates
Aprx-Sfw	1 or 2	1 or 2	$O(\|\mathcal{W}\|\|\mathcal{S}\|^2\|\mathbb{Q}\|)$	$O(\|\mathcal{S}\|^2\|\mathbb{Q}\|)$	Predicate Approximation
CwFr	2	1 or 2	$O(\|\mathcal{S}\|\|\mathbb{Q}\|)$	$O(\|\mathcal{S}\|)$	Quorum Views / Highest Tag

Table 1: Comparison of the four algorithms.

4 NS2-Simulation

In this section we describe in detail our NS2 simulations. We provide some basic details about the NS2 simulator and then we present our testbed. Following our testbed, we present the parameters we considered and the scenarios we run for our simulations.

4.1 The NS2 Network Simulator

NS2 is a discrete event network simulator [2]. It is an open-source project built in C++ that allows users to extend its core code by adding new protocols. Because of its extensibility and plentiful online documentation, NS2 is very popular in academic research. Customization of the NS2 simulator allows the researcher to obtain full control over the event scheduler and the deployment environment. Complete control over the simulation environment and its components will help us investigate the exact parameters that are affected by the implementation of the developed algorithms. Performance of the algorithms is measured in terms of the ratio of the number of fast over slow R/W operations (communication burden), and the total time it takes for an operation to complete (communication + computation = operation latency). Measurements of the performance involves multiple execution scenarios; each scenario is dedicated in investigating the behavior of the system affected by a particular system characteristic. The following system components will be used to generate a variety of simulation executions and enable a more comprehensive evaluation of the developed algorithms. Notice that each component affects a different aspect of the modeled environment. Thus, studying executions affected by the variation of a single or multiple components are both of great importance.

4.2 Experimentation Platform

Our test environment consists of a set of writers, readers, and servers. Communication between the nodes is established via bidirectional links, with:

- 1Mb bandwidth,
- latency of $10ms$, and

- DropTail queue.

To model local asynchrony, the processes send messages after a random delay between 0 and 0.3 *sec*. We ran NS2 in Ubuntu, on a Centrino 1.8GHz processor. The average of 5 samples per scenario provided the stated operation latencies.

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 participant node via an external service (out of the scope of this work).

We model server failures by selecting some quorum of servers (unknown to the participants) to be correct and allowing any other server to crash. The positive time parameter $cInt$ is used to model the failure frequency or reliability of every server s. For our simulations we initialize $cInt$ to be equal to the one third of the total simulation time. Each time a server checks for failure, it cuts $cInt$ in half until it becomes less than one. A failure is generated as following. First, the server determines whether it belongs in the correct quorum. If not the server sets its crash timeout to $cInt$. Once $cInt$ time is passed, the server picks a random number between 0 and 100. If the number is higher than 95 then the server stops. In other words a servers has 5% chance to crash every time the timer expires.

We use the positive time parameters $rInt = 4sec$ and $wInt = 4sec$ 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.

Finally we specify the number of operations each participant should invoke. For our experiments we allow participants to perform up to 25 operations (this totals to 500-4000 operations in the system).

4.3 Scenarios

The scenarios were designed to test (i) scalability of the algorithms as the number of readers, writers and servers increases, (ii) the relation between quorum system deployment and operation latency, and (iii) whether network delays may favor the algorithms that minimize the communication rounds. In particular we consider the following parameters:

1. **Number of Participants:** We run every test with 10, 20, 40, and 80 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.

3. **Network Latency:** Operation latency is affected by local computation and communication delays. As the speed of the nodes is the same, it is interesting to examine what is the impact of the network latency on the overall performance of the algorithms. In this scenario we examine whether higher network latencies may favor algorithms (like CwFr and Aprx-Sfw) that, although they have high computation demands, they allow single round operations. For this scenario we change the latency of the network from 10ms to 500ms and we deploy a 6-wise quorum construction.

Another parameter we experimented with 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 Simulation Results

In this section we discuss our findings. First we compare the operation latency in algorithms Sfw with Aprx-Sfw to examine our theoretical claims about the computational hardness of the two algorithms. Then 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 simulations appear in the Appendix. In the following sections we make clear references to those plots.

5.1 Algorithm Sfw vs. Aprx-Sfw

In [10] the authors showed that the predicates used by Sfw were NP-complete. That motivated the introduction of the Aprx-Sfw approximation algorithm, that could provide a polynomial, log approximation solution to the computation of the read and write predicates used in Sfw. To provide an experimental proof of the results presented in [10] we implemented both algorithms in NS2. We run the two algorithms using different environmental parameters and we observed the latency of read and write operations in any of those cases. In particular we run scenarios with $|\mathcal{R}| \in [10, 20, 40, 80]$ readers, with $|\mathcal{W}| = [10, 20, 40]$ writers, and with $|\mathcal{S}| = [10, 15, 20]$ servers where $f = 1$ may crash. The exceedingly large delay of Sfw in scenarios with many servers and writers prevented us from obtaining results for bigger $\mathcal{S}$ and $\mathcal{W}$. The results we managed to obtain however were enough to reveal the difference between the two approaches. The plots of our results appear in Appendix A.

Figure 2 presents a specific scenario where $|\mathcal{R}| = |\mathcal{W}| = 20$. Examining the latency of the two algorithms, including both communication and computation costs, provides evidence of the heavy computational burden of algorithm Sfw. It appears that the average latency of the read operations (Figure 2 right column) in algorithm Sfw grows exponentially with respect to the number of servers (and thus quorum members) in the deployed quorum system. As it appears in the figure, the average latency for every read in Sfw was little lower than 200sec when using 15 quorums, and then it exploded close to 1200sec when the number of quorums is 20. On the other hand the average latency of read operations in Aprx-Sfw grows very slowly. The average number of slow reads as it appears on the left plot of Figure 2 shrinks as the number of quorums grows, for both algorithms. Notice that as the number of servers grows the intersection degree which is equal to $n = (\frac{|\mathcal{S}|}{f} - 1)$ grows as well since f is fixed to 1. From the figure we also observe that although the approximation algorithm may invalidate the predicate when there is actually a solution, its average of slow reads does not diverge from the average of slow reads in Sfw. This can be explain from the fact that read operations may be fast even when the predicate does not hold, but there is a confirmed tag propagated in sufficient servers.

20 Readers, 20 Writers, f=1:

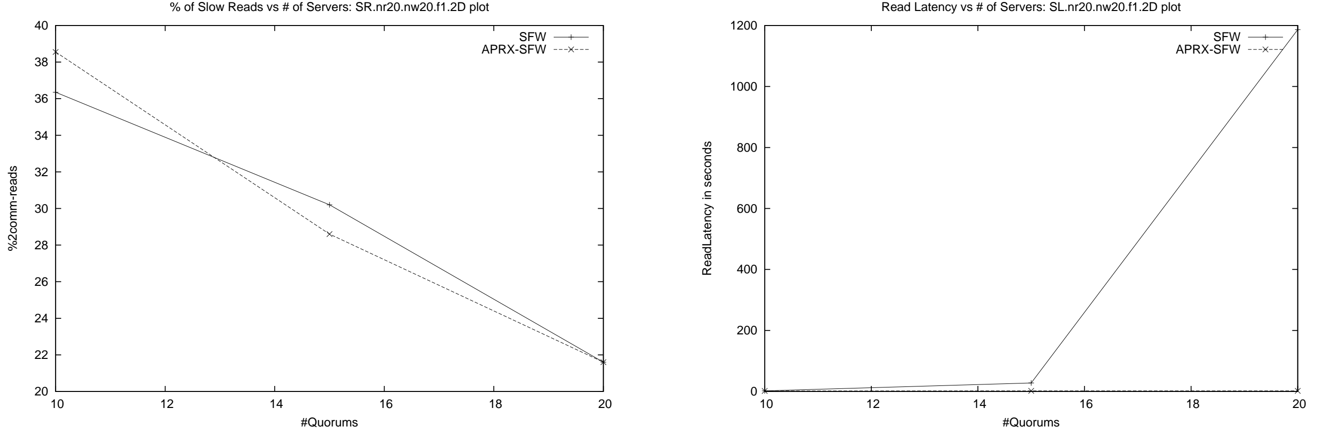


Figure 2: **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

10 Readers:

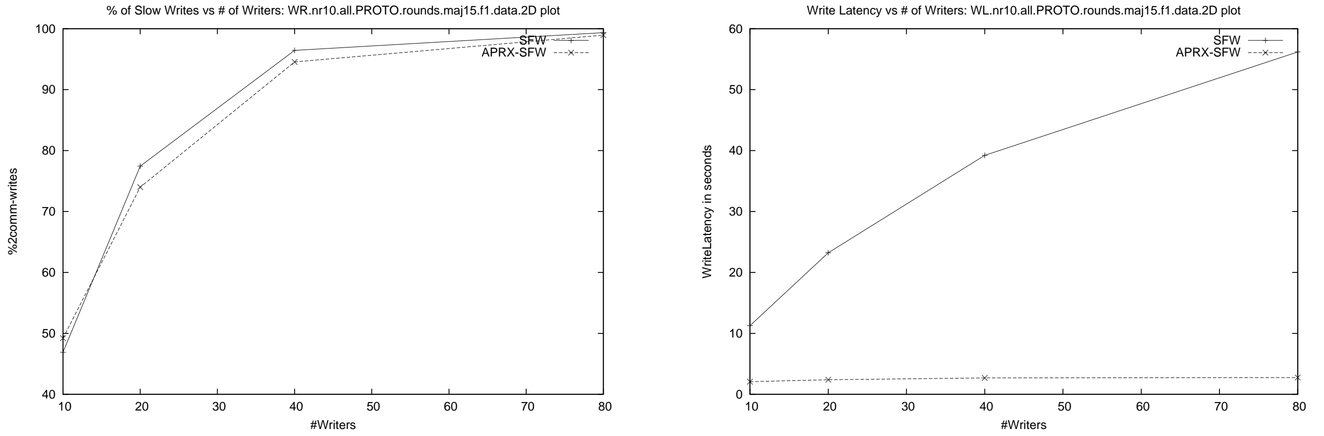


Figure 3: 14-wise quorum system ($|\mathcal{S} = 15,\ f = 1$): **Left Column:** Percentage of slow writes, **Right Column:** Latency of write operations

A writer performs two rounds only when the predicate does not hold. Thus, counting the number of two-round writes reveals how many times the predicate does not hold for an algorithm. According to our theoretical findings, algorithm Aprx-Sfw should allow no more than $\log|\mathcal{S}| \cdot RR$ two-round reads or $\log|\mathcal{S}| \cdot WR$ two-round writes in each scenario, where RR and WR are the number of two-round reads and writes allowed by the algorithm, respectively. Our experimental results are within the theoretical upper bound, illustrating the fact that algorithm Aprx-Sfw implements a $\log|\mathcal{S}|$-approximation relative to algorithm Sfw. Figure 3 presents the average amount of slow writes and the average write latency when we fix $|\mathcal{R}| = 10$, $|\mathcal{S}| = 15$ and $f = 1$. The number of writers vary from $|\mathcal{W}| = [10, 20, 40, 80]$. This is one of the few scenarios we could run Sfw with 80 writers. As we can see the two algorithm experience a huge gap on the completion time of each write operation. Surprisingly, Aprx-Sfw appears to win on the number of slow writes as well. Even though we would expect that Aprx-Sfw would contain more slow writes than Sfw this is not an accurate measure of the predicate validation. Notice that read operations may be invoked concurrently with write operations, and each read may also propagate a value in the system. This may favor or delay write operations. As the conditions on which the write operations try to evaluate their predicates are difficult to find, it suffices to observe that there is a small gap on

the number of rounds for each write for the two algorithms. Thus we claim the clear benefit of using algorithm Aprx-Sfw over algorithm Sfw.

5.2 Algorithm Simple vs. CwFr vs. Aprx-Sfw

In this section we compare algorithm Aprx-Sfw with algorithm CwFr. To examine the impact of computation on the operation latency, we compare both algorithms to 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 4.3. A general conclusion that can be extracted from the simulations is that in most of the tests 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.

Variable Participation: For this scenario we tested the scalability of the algorithms when the number of readers, writers, and servers changes. The plots that appear in the Appendices B and C present the results of this scenario for the read and write performance respectively.

The plots in Appendix B present how the performance of read operation is affected as we change the number of readers. Each figure contains the plots we obtain when we fix the number of servers and server failures, and we vary the number of readers and writers. So, for instance, Figure 4 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 $|\mathcal{W}| \in [10, 20, 40, 80]$. Each row in a figure contains a pair of plots that presents the percentage of slow reads (left plot) and the latency of each read (right plot) as we increase the number of readers.

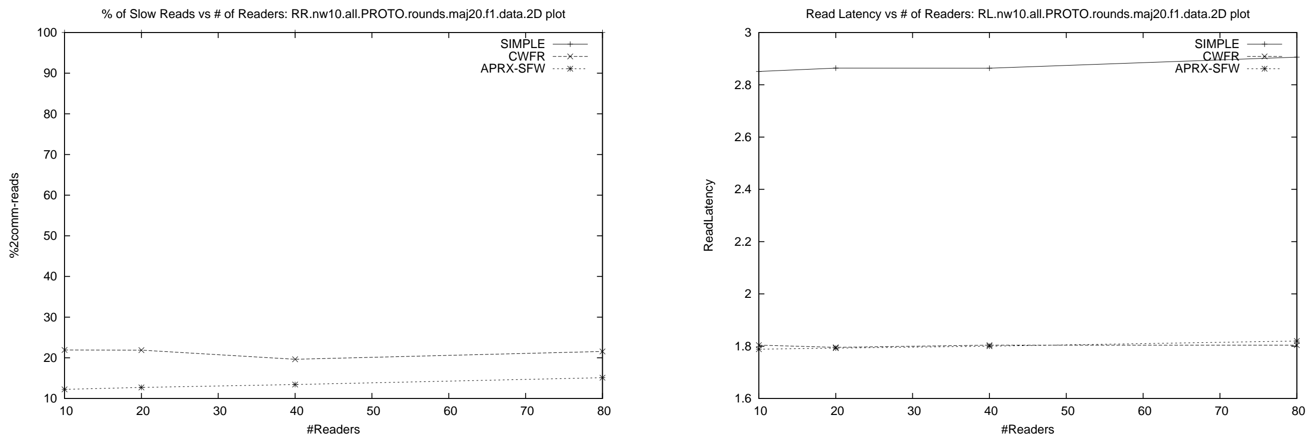
From the plots we observe that the read performance of neither algorithm is affected a lot by the number of readers in the system. On the other hand we observe that the number of writers seem to have an impact on the performance of the Aprx-Sfw algorithm. As the number of writers grow we observe that both the number of slow read operations and inevitably the latency of read operations for Aprx-Sfw increases. Both CwFr and Aprx-Sfw require fewer than 20% of reads to be slow when no more than 20 writers exist in the service. That is true for most of the server participation scenarios and leaves the 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. Once the number of writers grows larger than 40 the read performance of Aprx-Sfw degrades both in terms the number of slow reads and the average latency of read operations.

Unlike Aprx-Sfw, algorithm CwFr is not affected by the participation of the service. The number of slow reads seem to by sustained below 30% in all scenarios and the average latency of each read remains under the 2 second marking. Aprx-Sfw over-performs CwFr only when the intersection degree is large and the number of writers is small.

Our outcomes can be seen in Figure 4. The 20-wise intersection allows Aprx-Sfw to enable more single round read operations. Due to computation burden however the average latency of each read in Aprx-Sfw is almost identical to the average read latency in CwFr. It is also evident that the reads in Aprx-Sfw are greatly affected by the number of writers in the system. That was expected as by Aprx-Sfw, each read operation may examine a single tag per writer. From the plots on the second row of Figure 4 we observe that although close to half reads are fast the average read latency of Aprx-Sfw exceeds the average latency of Simple, or otherwise the average latency of 2 communication rounds. This is evidence that the computation time of Aprx-Sfw exceeded by a great margin the time required for a communication round.

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.

10 Writers:



80 Writers:

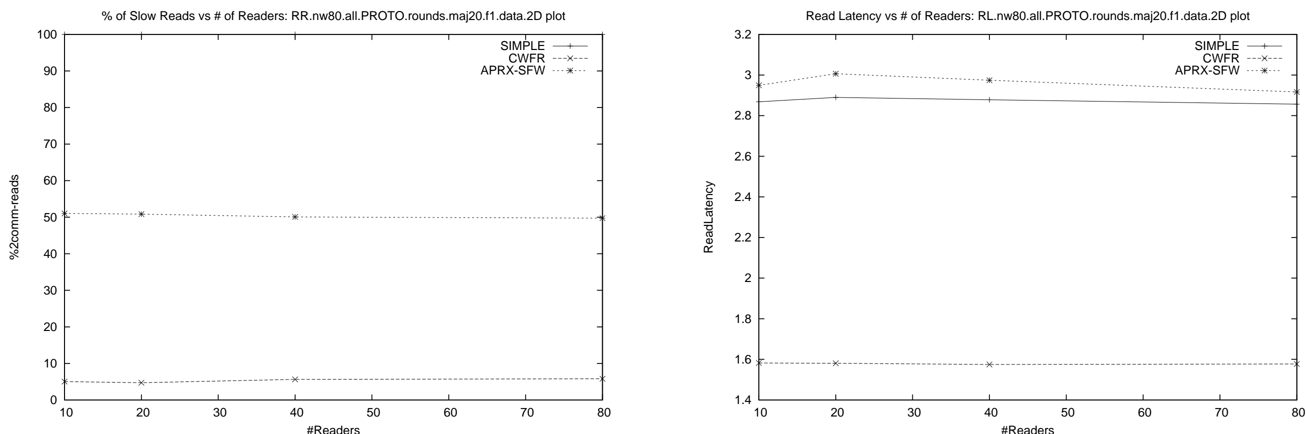


Figure 4: Plots from Figure 15 - 19-wise quorum system ($|\mathcal{S} = 20$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

The only algorithm that allows single round write operations is Aprx-Sfw. The number of writers and servers however, introduce high computation demands for the write operations and as a result, despite the fast writes, the average write latency of Aprx-Sfw can be higher than the average write latency of the other two algorithms. Note here that unlike the read operations, writes can be fast in Aprx-Sfw only when the write predicate holds. This characteristic can be also depicted from the plot presented in Figure 5 (part of Figure 25) where the 4-wise intersection does not allow for the write predicate to hold. Thus, every write operation in Aprx-Sfw performs two communication round in this case. 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 caused by the simulation randomness.

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 6 by modifying the number of servers and the maximum number of server failures.

In Appendix D 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

20 Readers:

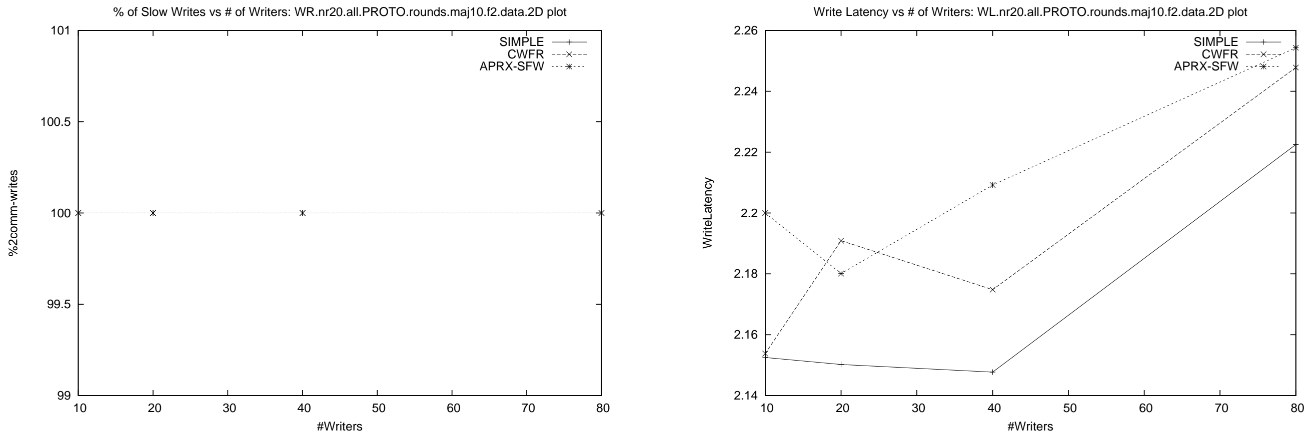


Figure 5: 4-wise quorum system ($|\mathcal{S} = 10,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

Servers $\lvert\mathcal{S}\rvert$	**Server Failures** f	**Int. Degree** n	**Quorums** $\lvert\mathbb{Q}\rvert$
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 6: Quorum system parameters.

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.

We observe from the plots that the incrementing number of servers, and thus cardinality of the quorum system, reduces the percentage of slow reads for both Aprx-Sfw and CwFr. 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. 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 1.5 sec in every scenario. With this read latency CwFr over-performs algorithm Simple in every scenario as the latter maintains a read latency between 2.5 and 3 sec. Unlike CwFr, algorithm Aprx-Sfw experiences a more aggressive change on the latency of read operations. The read latency in Aprx-Sfw is affected by both

40 Readers, 80 Writers, f=2:

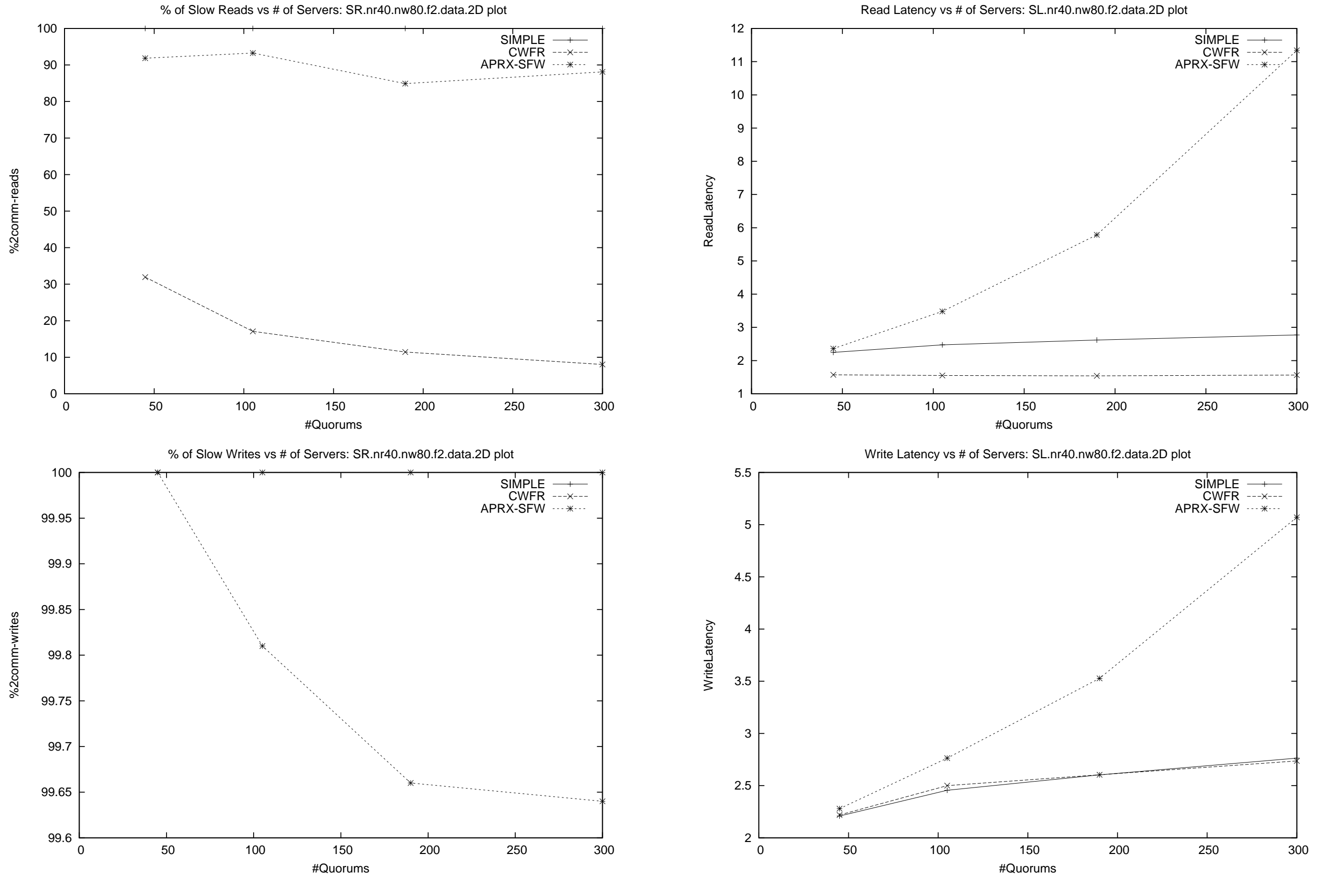


Figure 7: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

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.

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. The reduction on the amount of the slow writes however, is not proportional to the latency of each write. Thus, the average write latency of Aprx-Sfw increases as the number of quorums increases in the system. Interestingly however, unlike the latency of read operations, the latency of writes do not exceed the write latency of Simple by more than 3 times. 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 higher than the latency of write operations. An example of this behavior can be seen in Figure 7 (part of Figure 43). In this example

20 Writers:

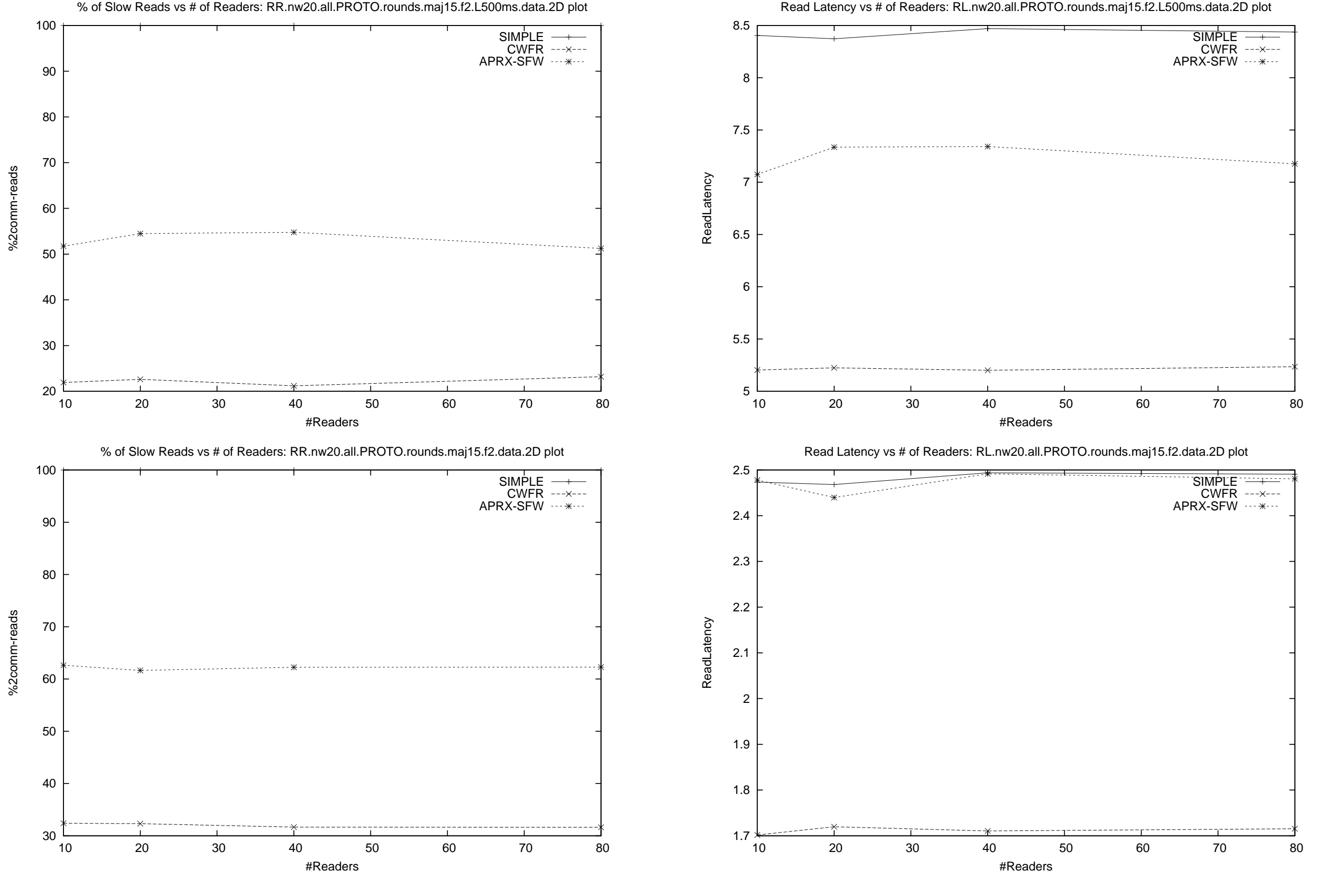


Figure 8: 6-wise quorum system ($|\mathcal{S}| = 15$, $f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

less than 90% of reads need to be slow and the average read latency climbs to almost 12 seconds. On the other hand almost every write operation is slow and the average write latency climbs just above 5 seconds. The simple explanation for this behavior lie 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.

Network Latency: During our last scenario we considered increasing the latency of the network infrastructure from 10ms to 500ms. With this scenario we want to examine whether in slow networks is more preferable to minimize the amount of rounds, even if that means higher computation demands. The results of this scenario appear in Appendix E. We considered just a single setting for this scenario where the number of servers is 15, the maximum number of failures is 2 and we pick the number of readers and writers to be one of [10, 20, 40, 80].

In order to establish meaningful conclusions we need to compare the outcomes of the operation performance of this scenario with the respective scenario where the latency is 10ms. We notice that the largest network delay reduces the amount of slow reads for both CwFr and Aprx-Sfw. In addition the delay indeed helps Aprx-Sfw to perform better than Simple in scenarios where Aprx-Sfw was performing identical or worse than Simple when the delay was 10ms. This can be seen in Figure 8 (part of Figure 45). As we can see in the figure the latency of the read operations of Aprx-Sfw was aligning with the read latency of Simple when the delay was 10ms. When we increased the network

delay to 500ms the average read latency of Aprx-Sfw was remarkably smaller than the average latency of Simple under the same participation and failure conditions. Similar observations can be made for the write operations as can be seen in Figures 47 and 48.

So we can safely conclude that the network delay can be one of the factors that may affect the decision on which algorithm is suitable for a particular application.

6 Conclusions

This work experimentally compares the operation latency of four 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 NS2 network simulator. At first we provided a comparison between algorithm Sfw and Aprx-Sfw. Then we compared the performance algorithms CwFr,Aprx-Sfw and Simple. Under the controlled environment offered by NS2 we were able to manipulate and test the algorithms under various environmental conditions, and extract valuable data regarding the characteristics of the three approaches. In particular, we tested the scalability of the algorithms by varying the number of readers, writers and servers in the system. Apart from scalability we tested how the performance of the algorithms is affected when deploying different quorum systems and when we increase the delay of the underlying network.

The comparison between algorithm Sfw over the approximation algorithm Aprx-Sfw demonstrated the computation gap between the two algorithms. The results for CwFr,Aprx-Sfw and Simple, suggested that algorithms CwFr and Aprx-Sfw over-perform algorithm Simple in most scenarios we tested. From our experiments we 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 especially on Aprx-Sfw. The algorithms CwFr and Simple are also affected, but insignificantly. This behavior agrees with our theoretical bounds presented in Table 1. According to the table the computation of every read operation in Aprx-Sfw is affected by the number of writers, servers, and quorums in the system. This explains the difference from CwFr and Simple whose read operation is not affected by the number of writers in the system. Similar for the write operations where Aprx-Sfw is affected by both the number of servers and the quorums in the system. A factor that favors Aprx-Sfw is the intersection degree of the underlying quorum system. We observed that large intersection degrees allowed for more fast read operations and as a result the latency of Aprx-Sfw was sometimes over-performing the latency of CwFr. Finally, the network delay has a negative impact on the operation latency of every algorithm we tested. We observed however, that network delays promote in some cases the use of algorithms with high computation demands that minimize the communication rounds, like algorithm Aprx-Sfw.

The next step is to better evaluate the practicality of the examined algorithms by deploying them on a real-time planetary scale environment, such as Planetlab [1]. In Planetlab the algorithms will adhere to the constraints imposed by the networked environment. Thus, we expect to obtain a better picture on the *realistic* performance of the algorithms as they will be tested under realistically adverse conditions.

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., 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.

[14] 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).

[15] 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.

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

[17] 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.

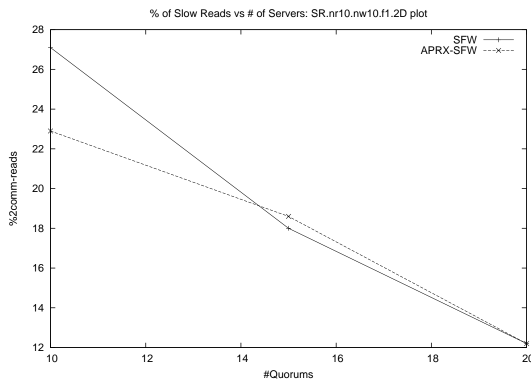
[18] Lynch, N. A., and Shvartsman, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.

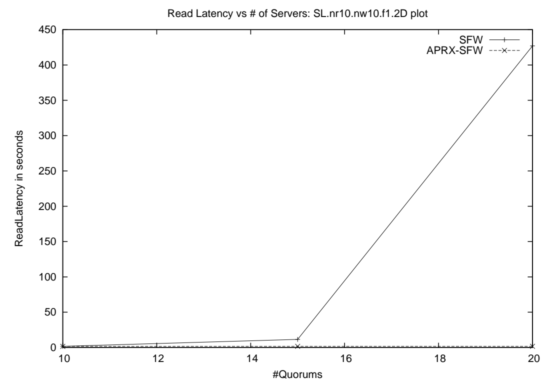
Appendix: Figures

A Sfw vs Aprx-Sfw

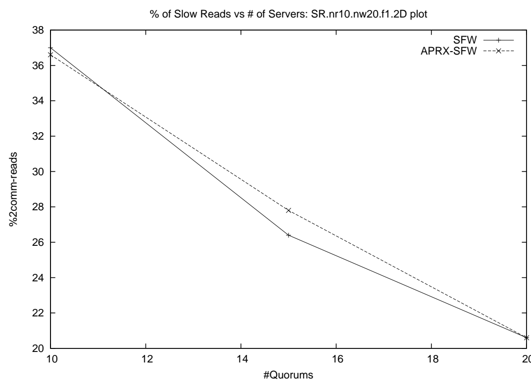
The plots below illustrate the latency of read operations with respect to the size of the quorum system under algorithms Sfw and Aprx-Sfw. The left column of each figure presents the percentage of slow read operations required by each algorithm.

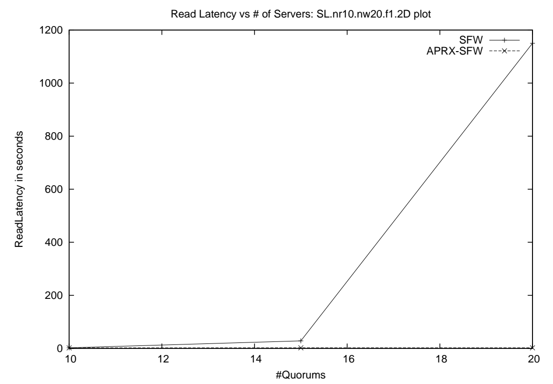
10 Readers, 10 Writers, f=1:



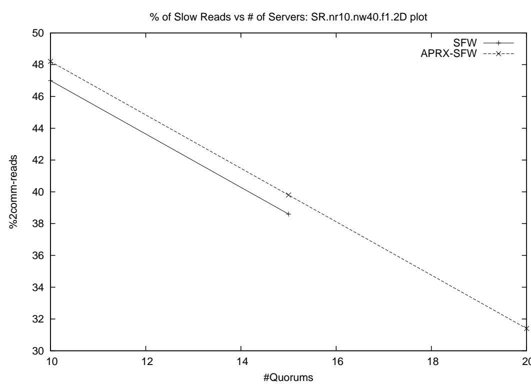


10 Readers, 20 Writers, f=1:





10 Readers, 40 Writers, f=1:



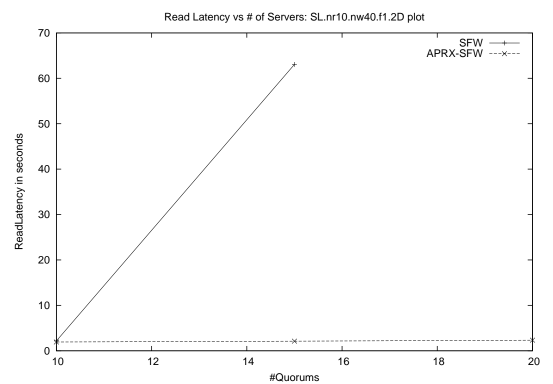
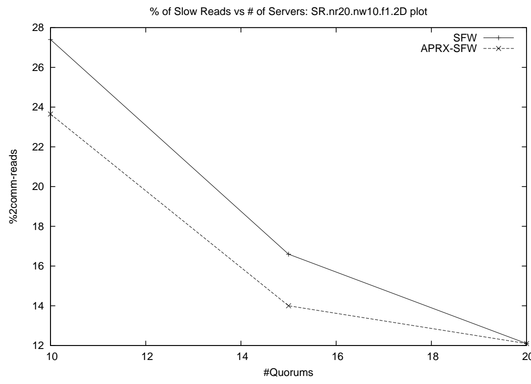
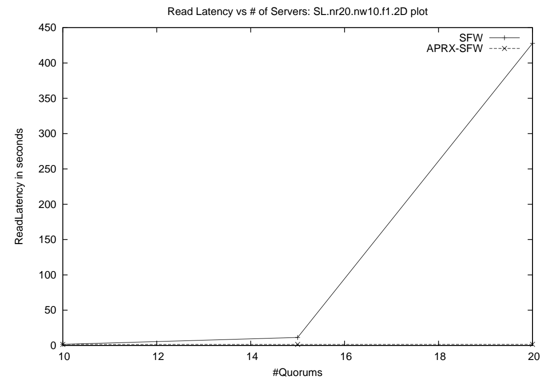


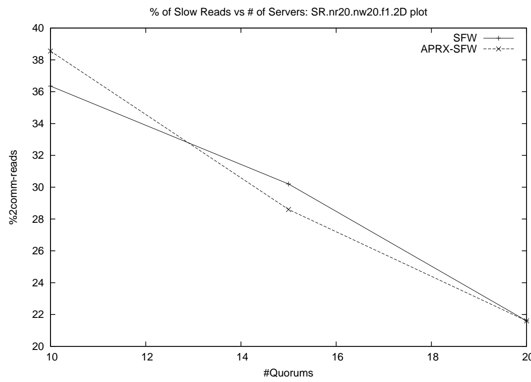
Figure 9: **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

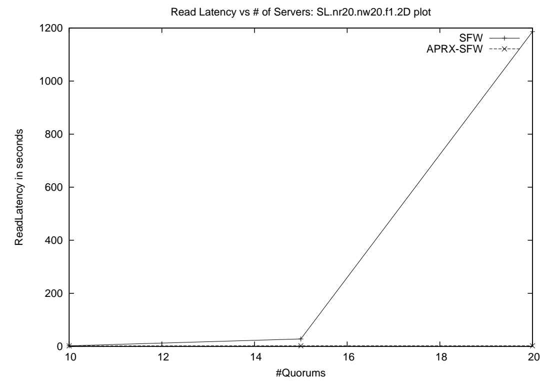
20 Readers, 10 Writers, f=1:



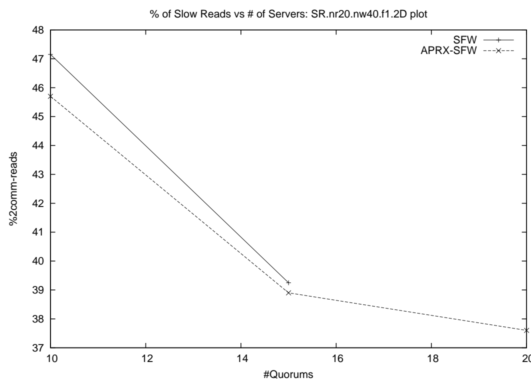


20 Readers, 20 Writers, f=1:





20 Readers, 40 Writers, f=1:



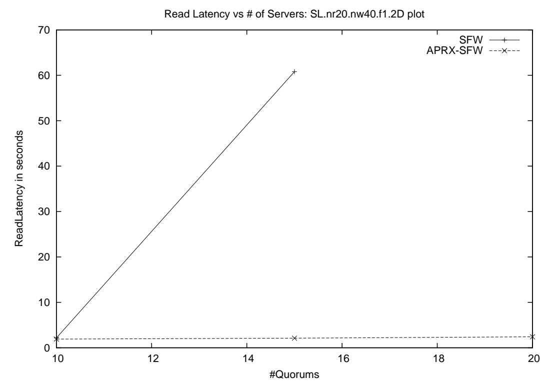
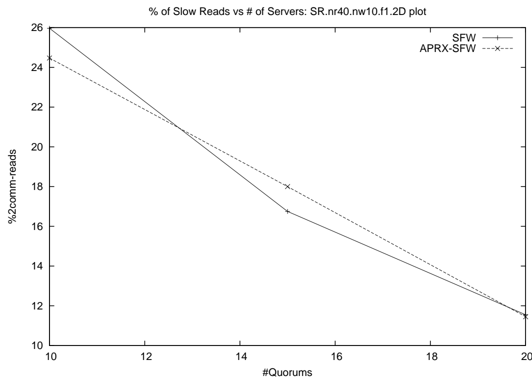
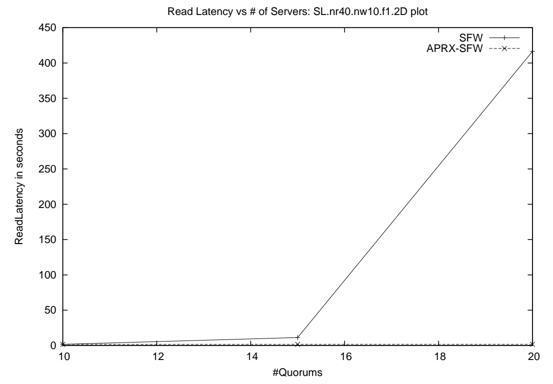


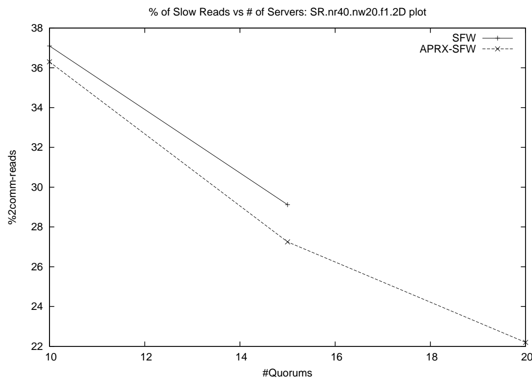
Figure 10: **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

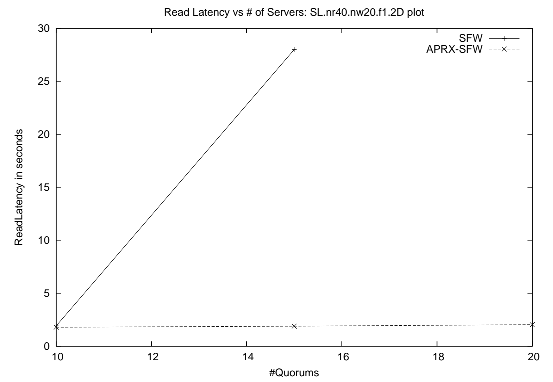
40 Readers, 10 Writers, f=1:



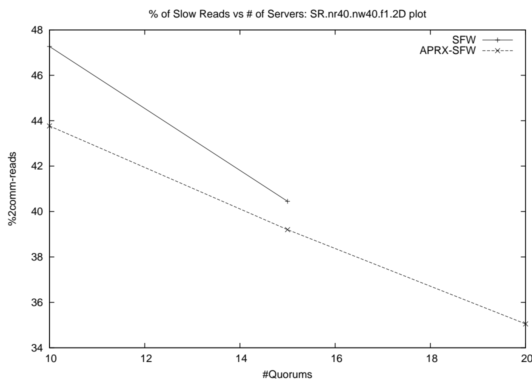


40 Readers, 20 Writers, f=1:





40 Readers, 40 Writers, f=1:



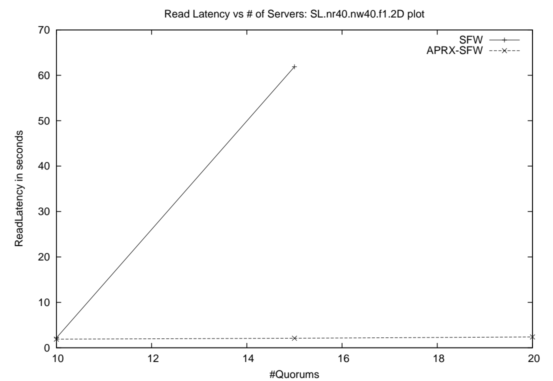
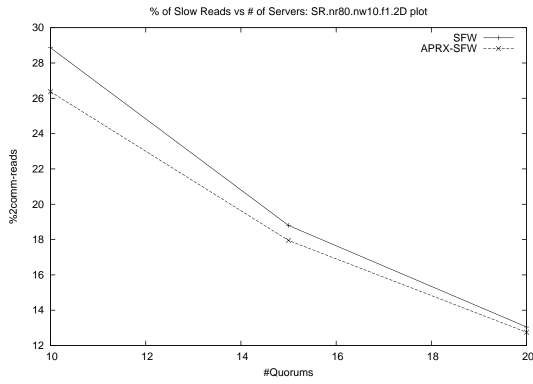
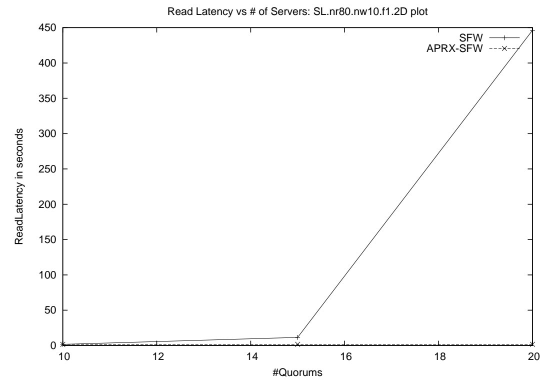


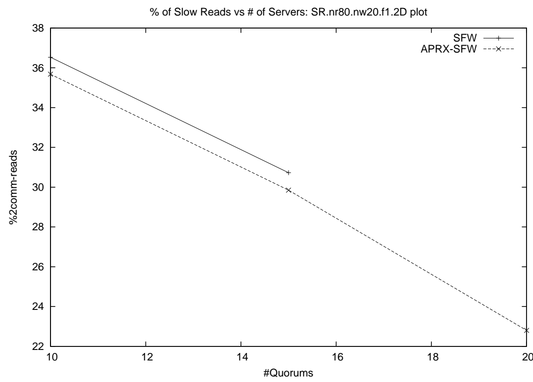
Figure 11: **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

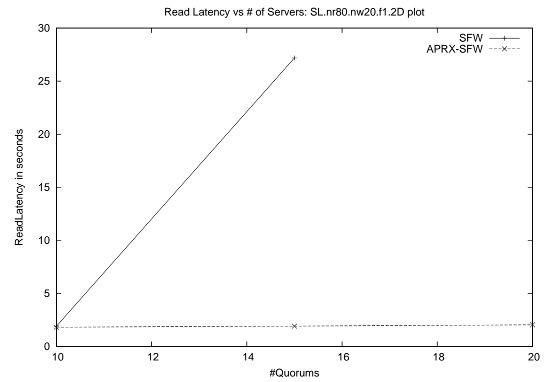
80 Readers, 10 Writers, f=1:



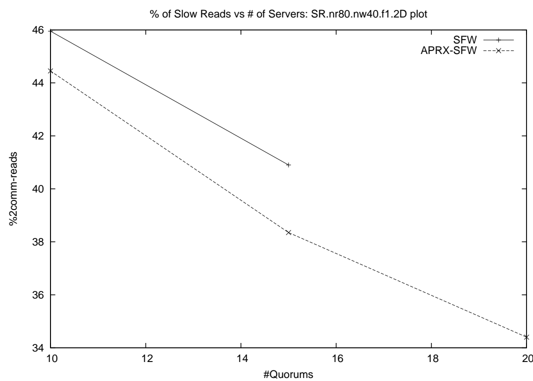


80 Readers, 20 Writers, f=1:





80 Readers, 40 Writers, f=1:



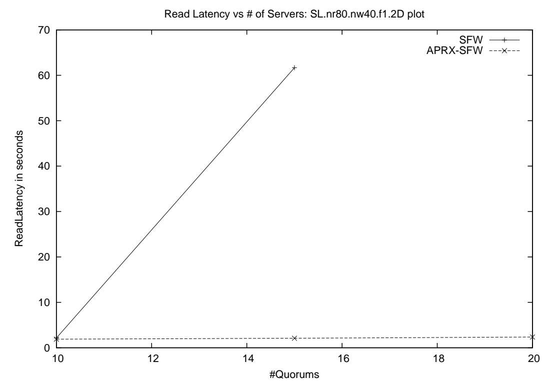
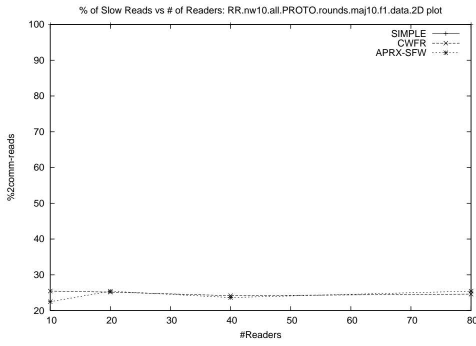


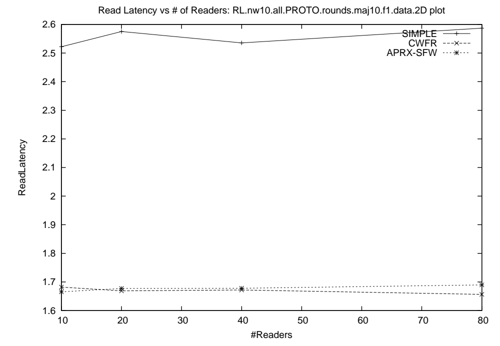
Figure 12: **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

B Read Performance

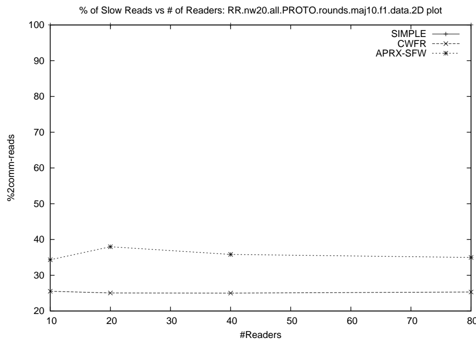
Below we present the plots regarding the read performance under variable number of writers and quorum constructions. In particular, we run the Simple, CwFr, and Aprx-Sfw algorithms using quorum constructions with eight different intersection degrees by setting the number of servers to 10, 15, 20, and 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.

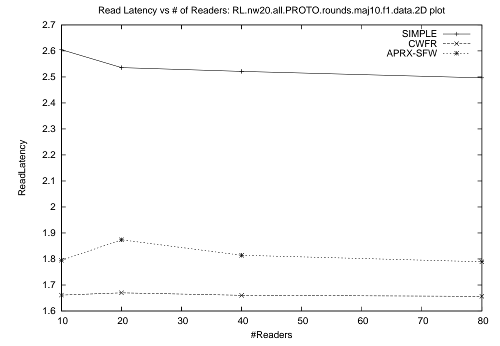
10 Writers:



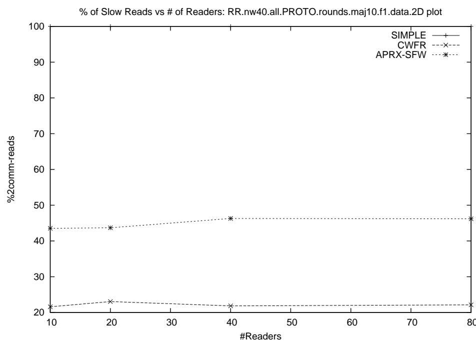


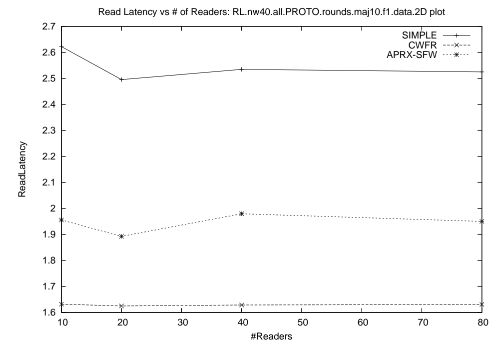
20 Writers:



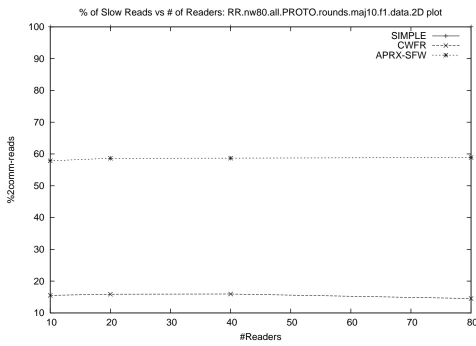


40 Writers:





80 Writers:



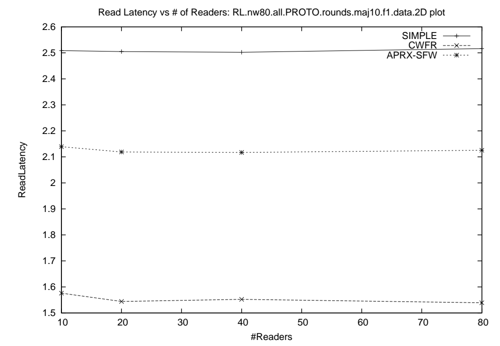
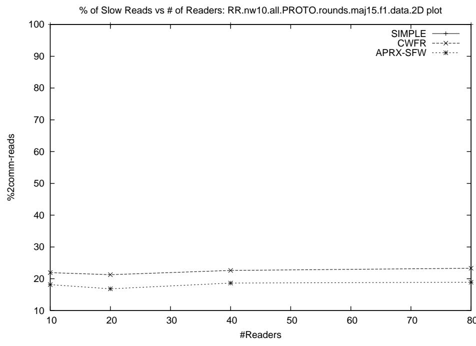
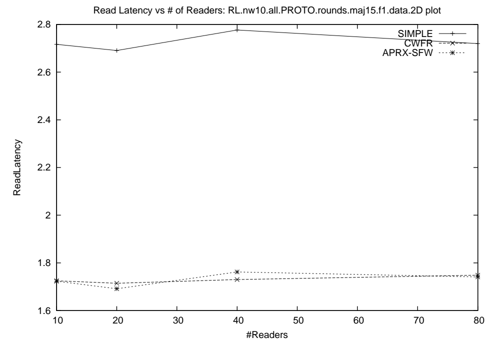


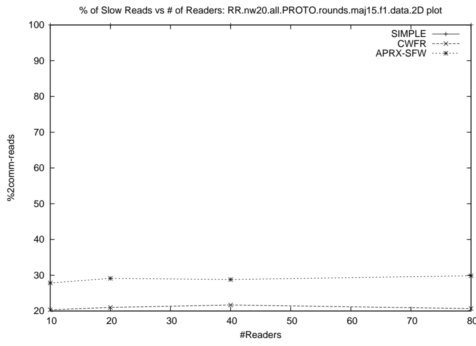
Figure 13: 9-wise quorum system ($|\mathcal{S}| = 10,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

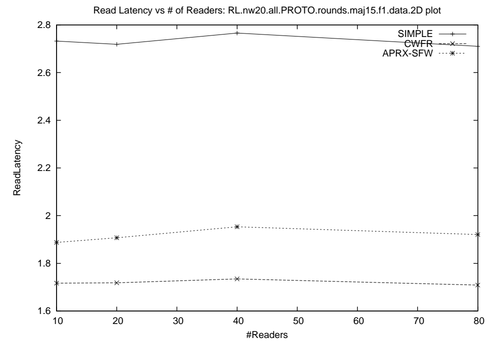
10 Writers:



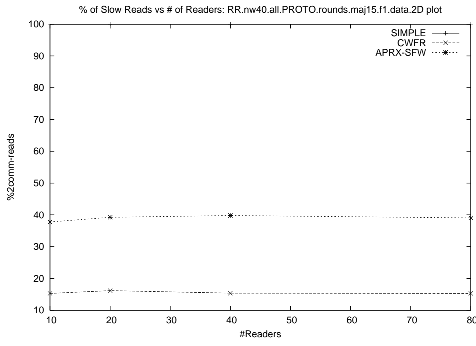


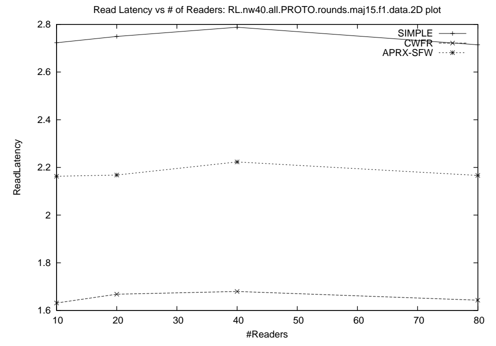
20 Writers:



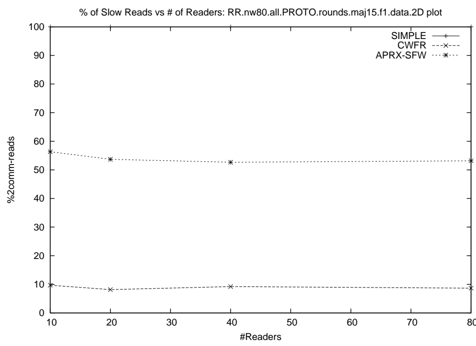


40 Writers:





80 Writers:



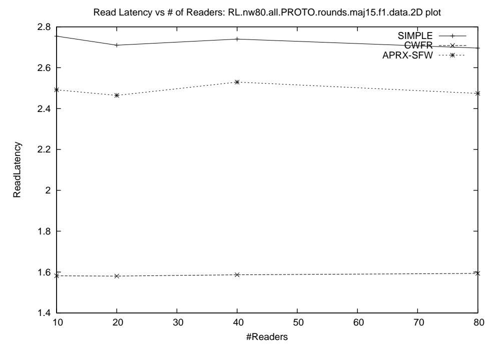
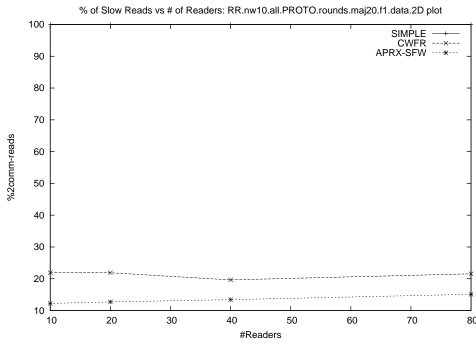
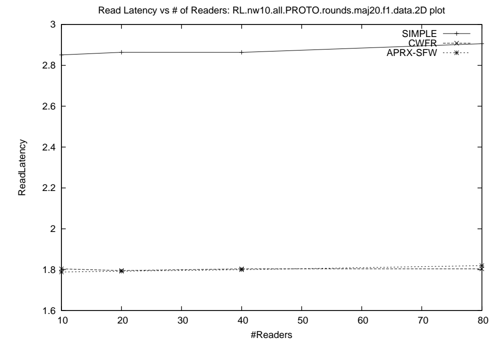


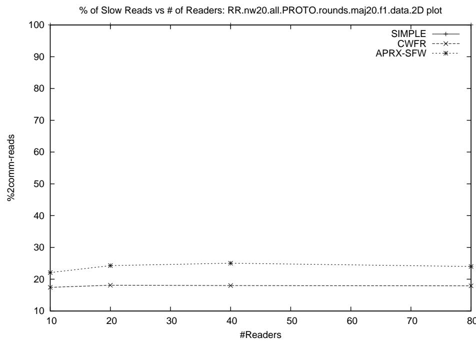
Figure 14: 14-wise quorum system ($|\mathcal{S}| = 15,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

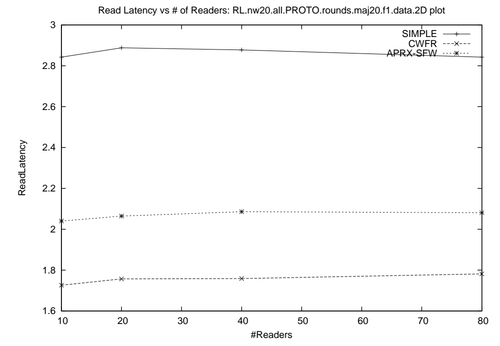
10 Writers:



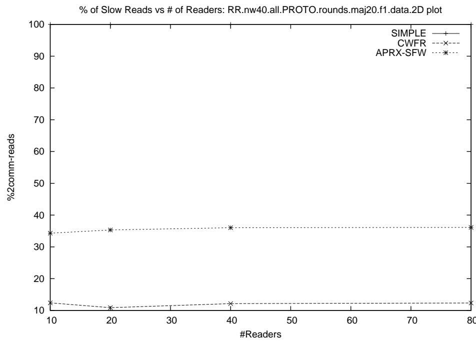


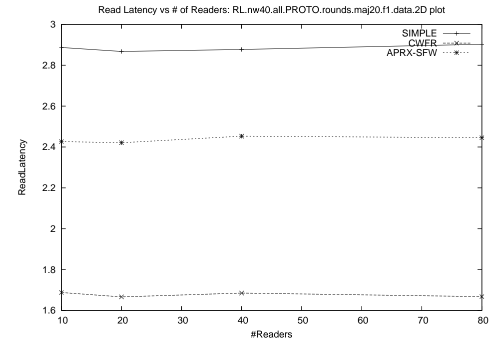
20 Writers:



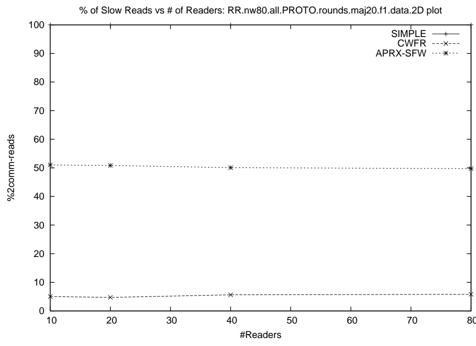


40 Writers:





80 Writers:



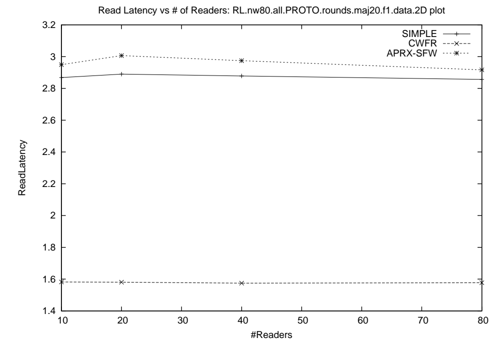
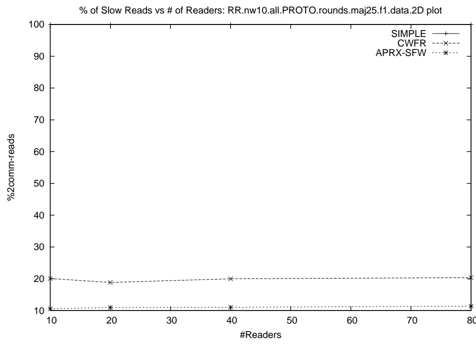
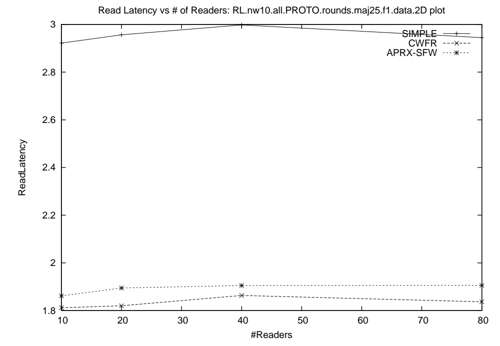


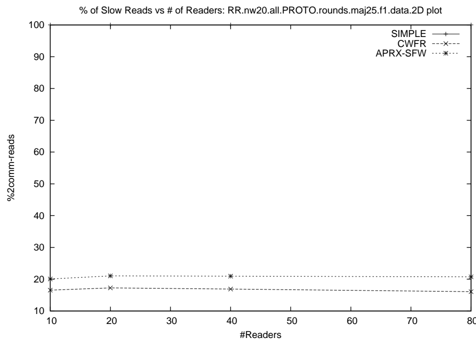
Figure 15: 19-wise quorum system ($|\mathcal{S}| = 20,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

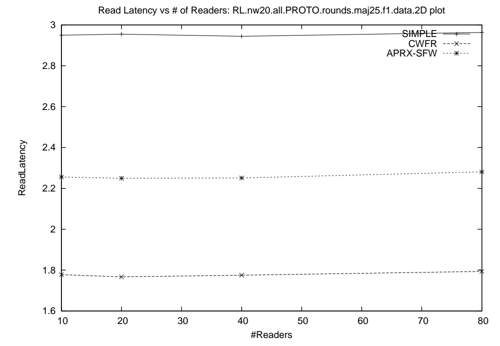
10 Writers:



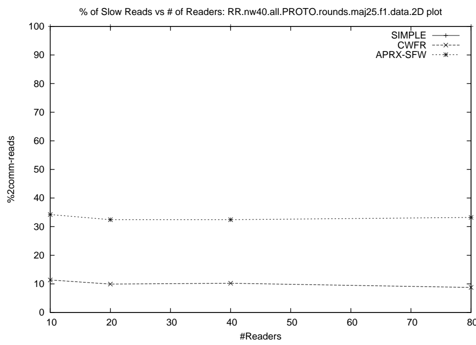


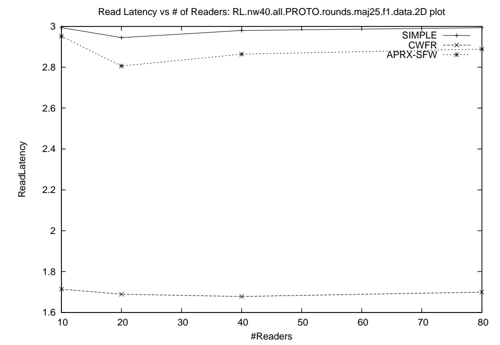
20 Writers:



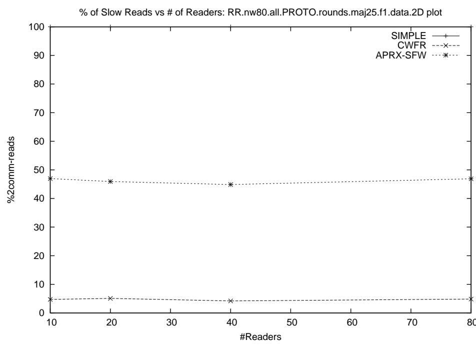


40 Writers:





80 Writers:



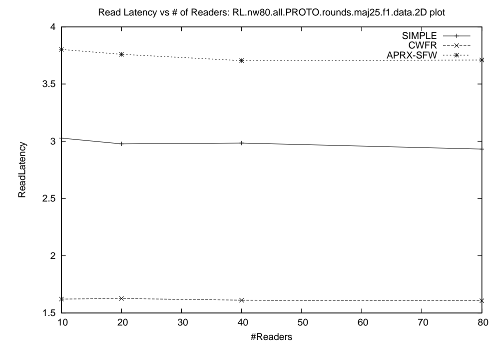
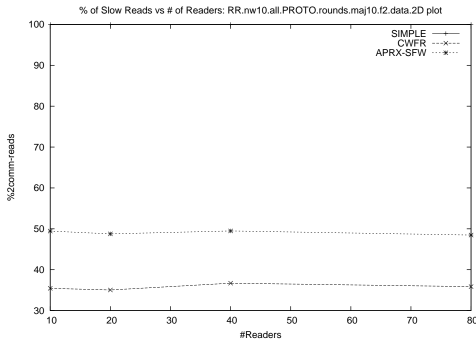
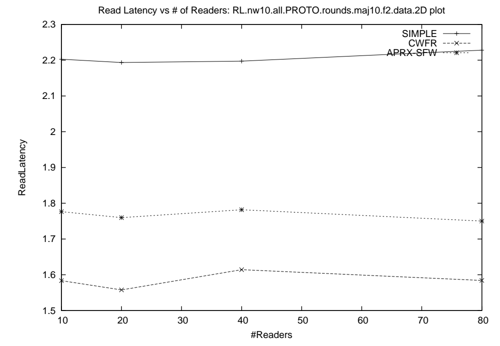


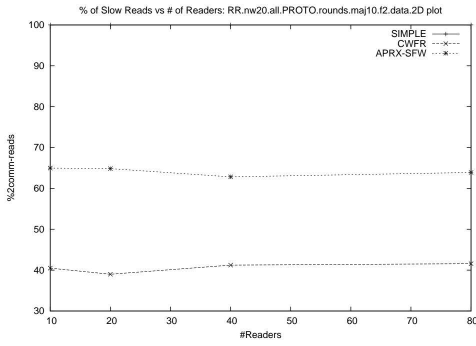
Figure 16: 24-wise quorum system ($|\mathcal{S}| = 25,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

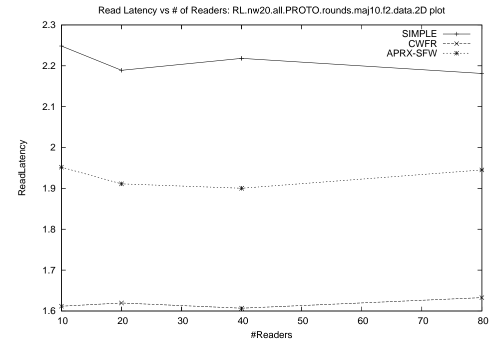
10 Writers:



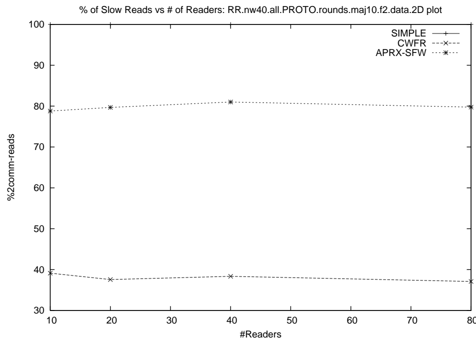


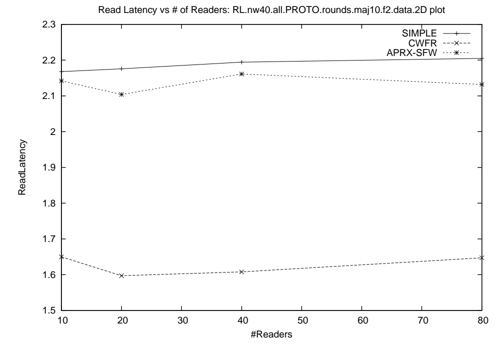
20 Writers:



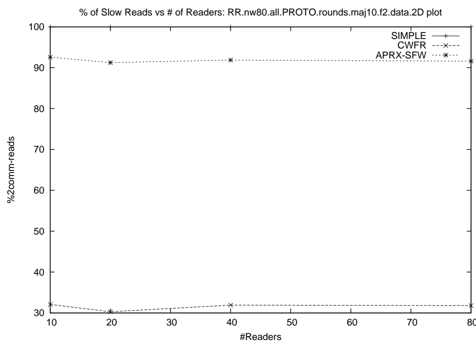


40 Writers:





80 Writers:



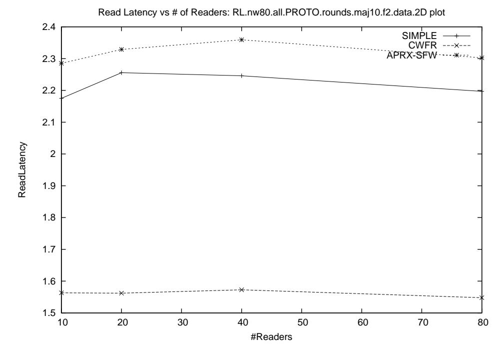
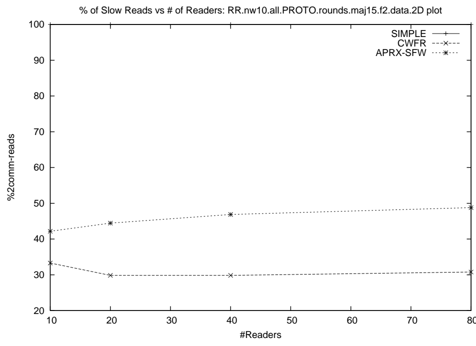
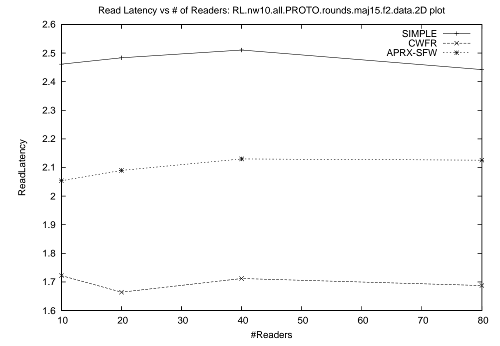


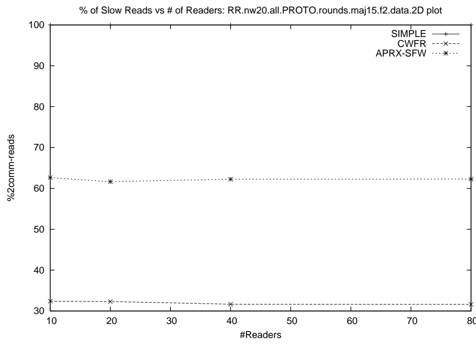
Figure 17: 4-wise quorum system ($|\mathcal{S}| = 10,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

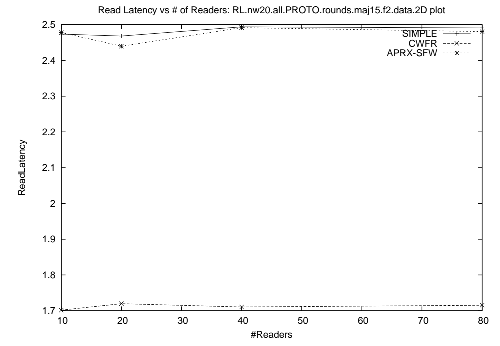
10 Writers:



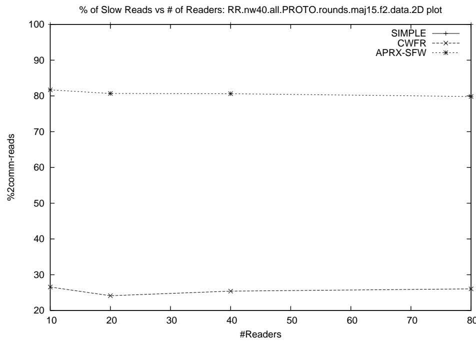


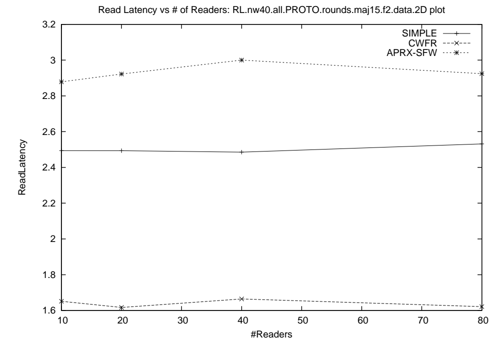
20 Writers:



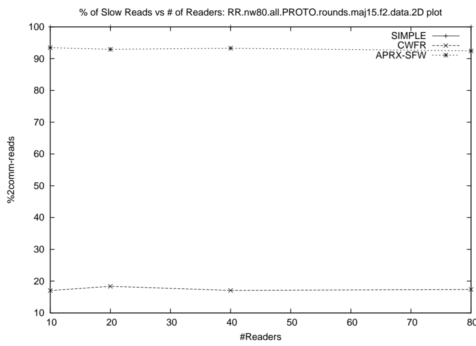


40 Writers:





80 Writers:



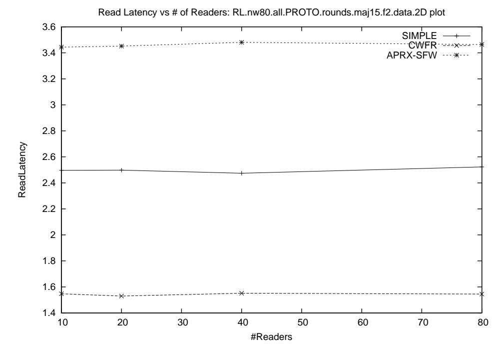
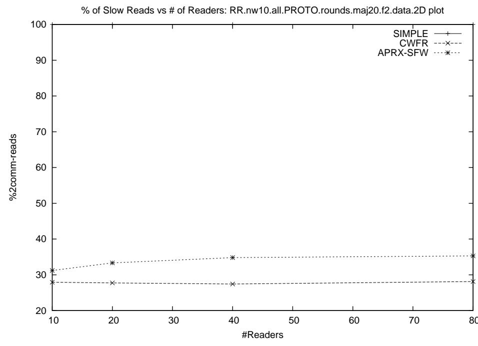
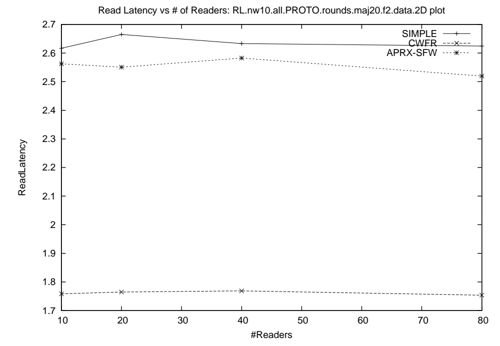


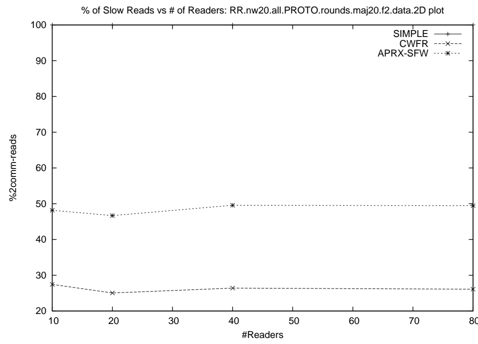
Figure 18: 6-wise quorum system ($|\mathcal{S}| = 15,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

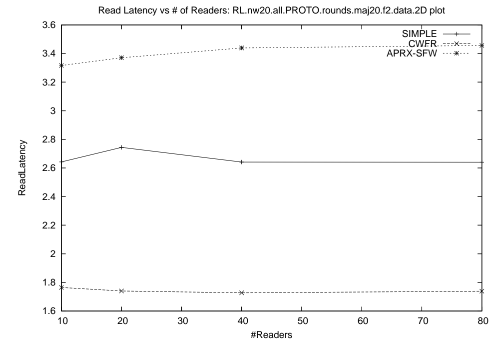
10 Writers:



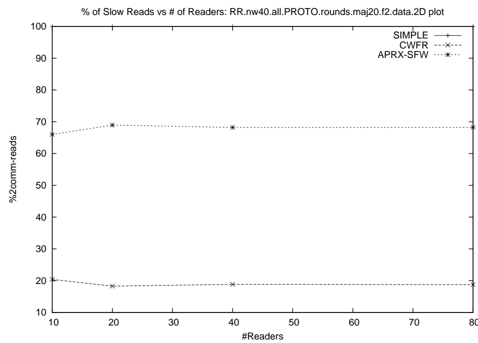


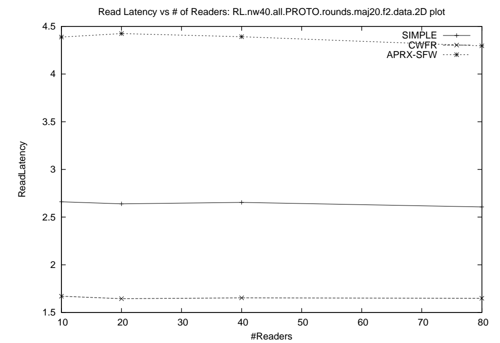
20 Writers:



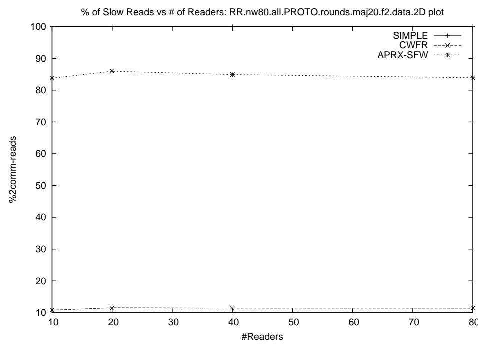


40 Writers:





80 Writers:



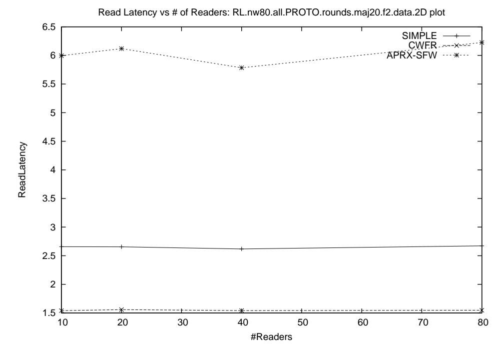
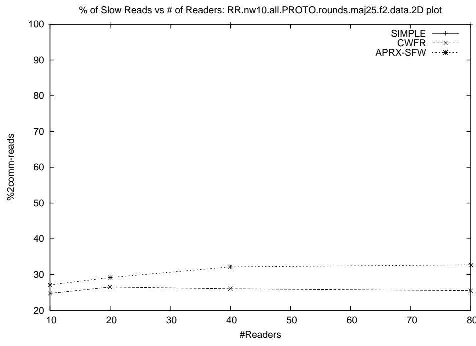
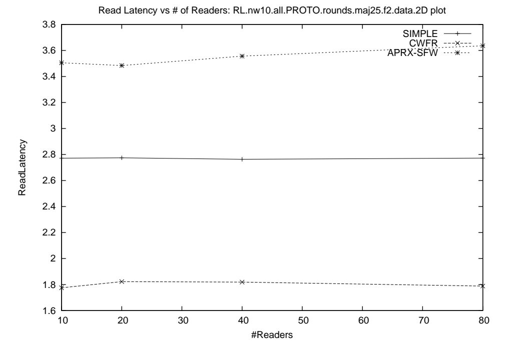


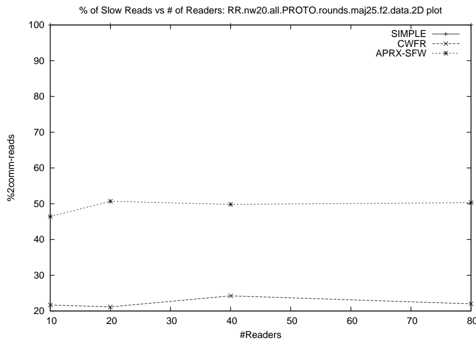
Figure 19: 9-wise quorum system ($|\mathcal{S}| = 20,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

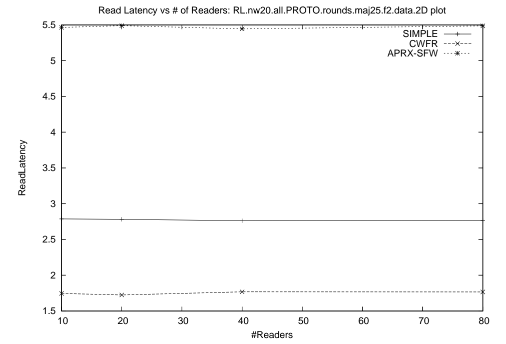
10 Writers:



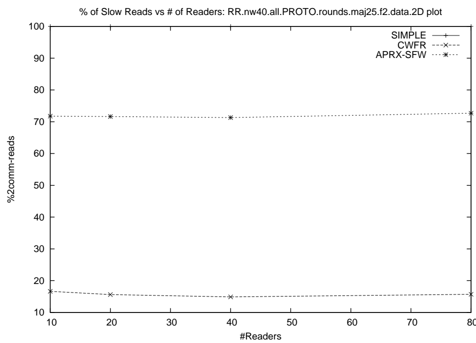


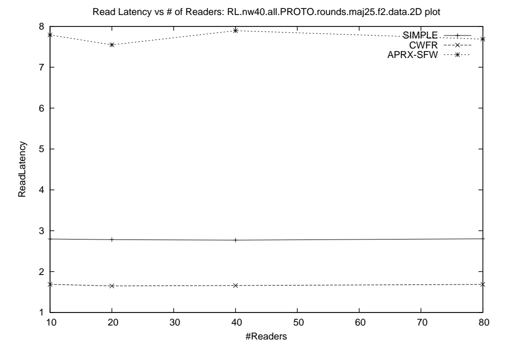
20 Writers:



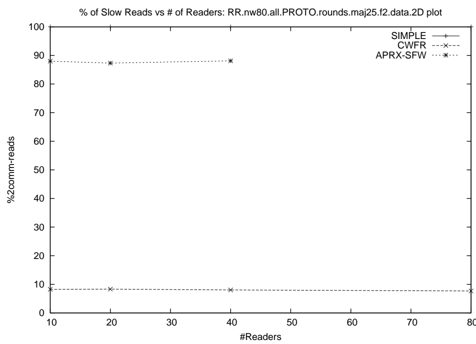


40 Writers:





80 Writers:



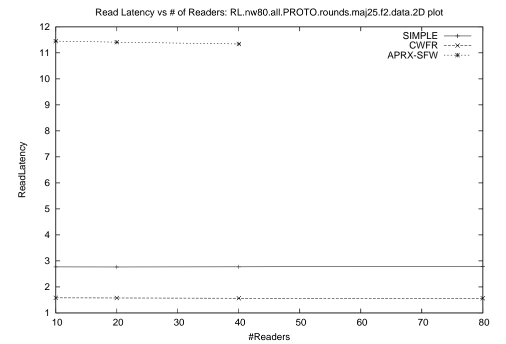
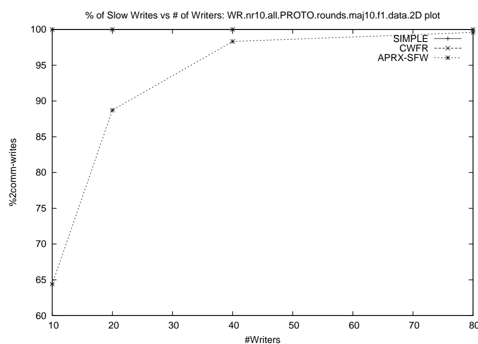


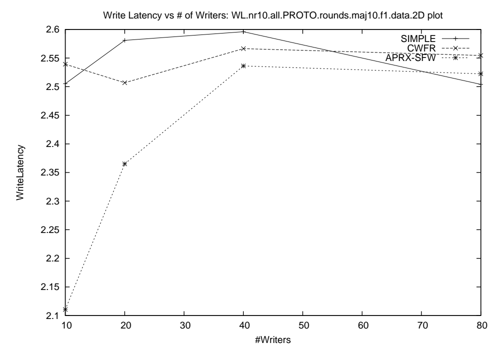
Figure 20: 11-wise quorum system ($|\mathcal{S}| = 25,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

C Write Performance

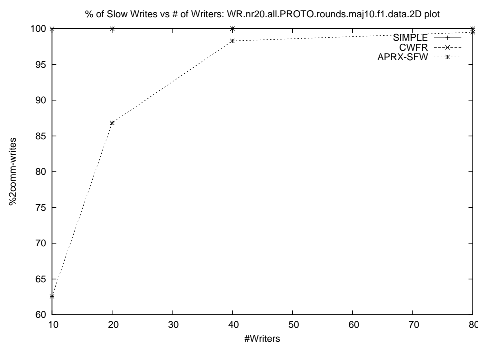
Below we present the plots regarding the write performance under variable number of readers and quorum constructions. In particular, we run the SIMPLE, CWFR, and APRX-SFW algorithms using quorum constructions with eight different intersection degrees by setting the number of servers to 10, 15, 20, and 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, 40, and 80 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.

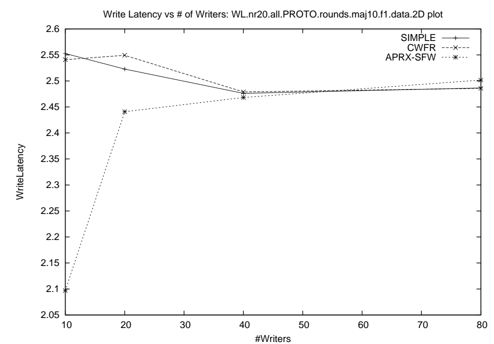
10 Readers:



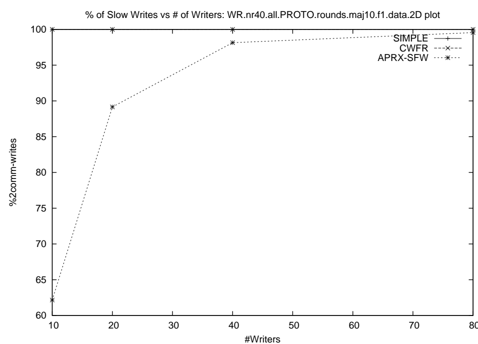


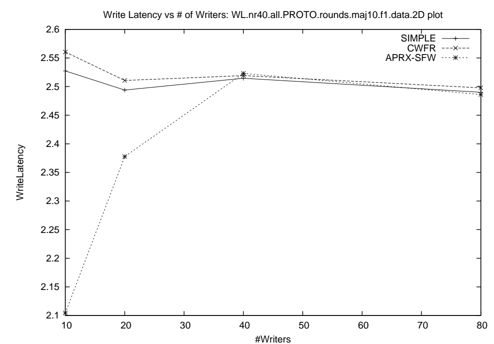
20 Readers:



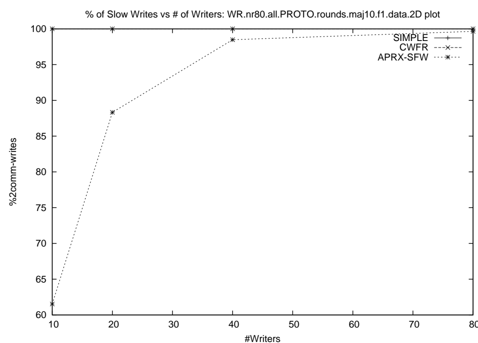


40 Readers:





80 Readers:



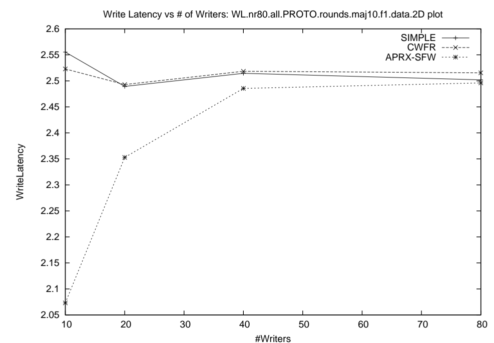
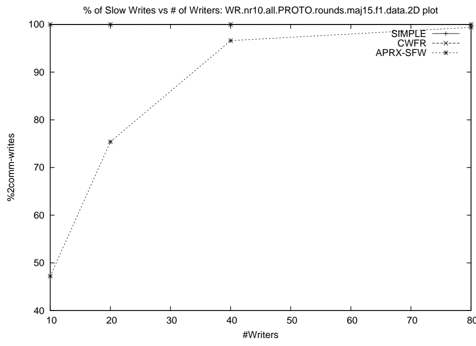
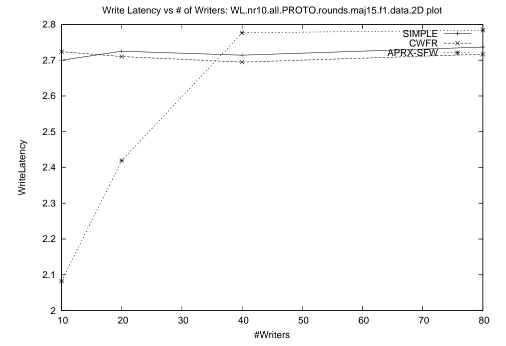


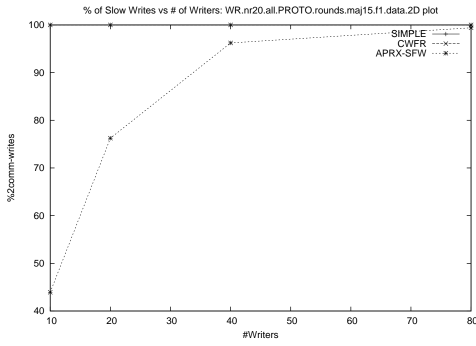
Figure 21: 9-wise quorum system ($|\mathcal{S}| = 10$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

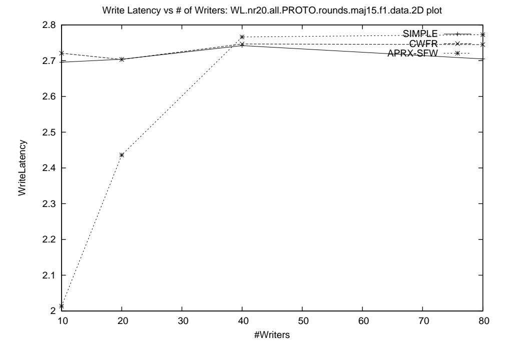
10 Readers:



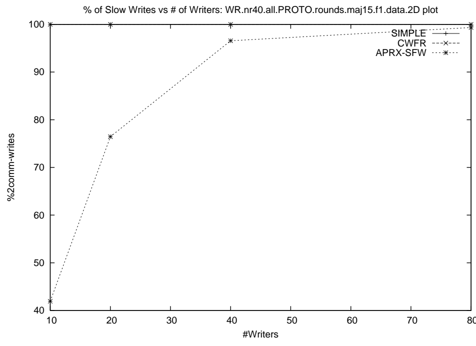


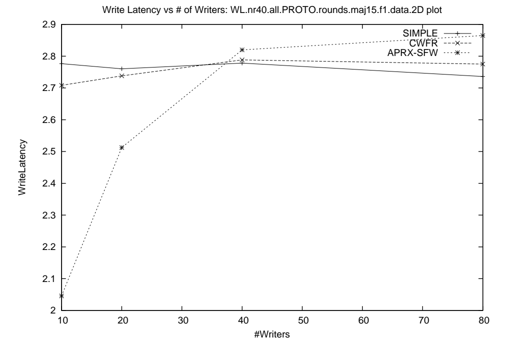
20 Readers:



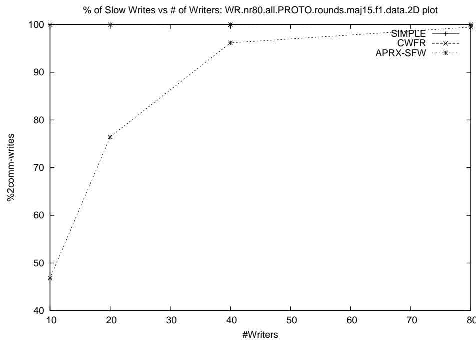


40 Readers:





80 Readers:



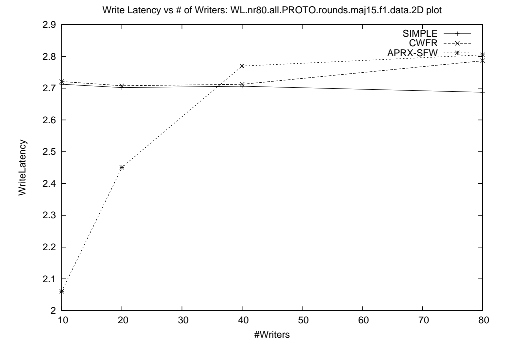
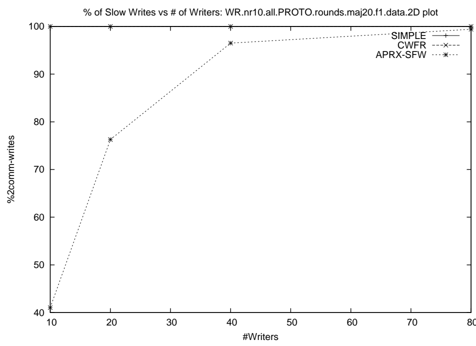
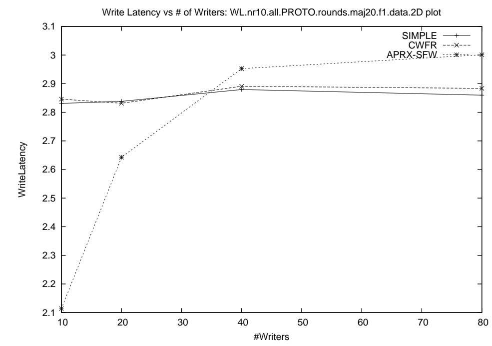


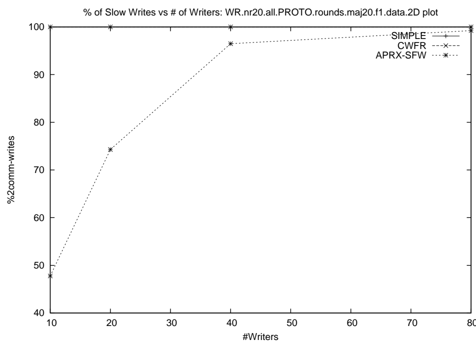
Figure 22: 14-wise quorum system ($|\mathcal{S}| = 15,\ f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

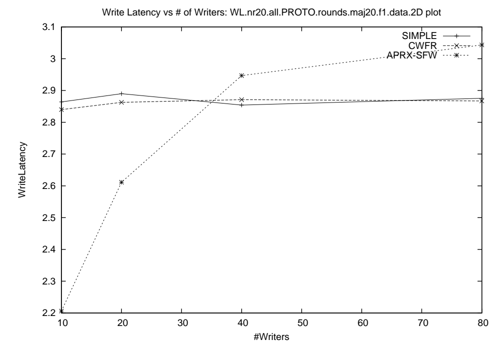
10 Readers:



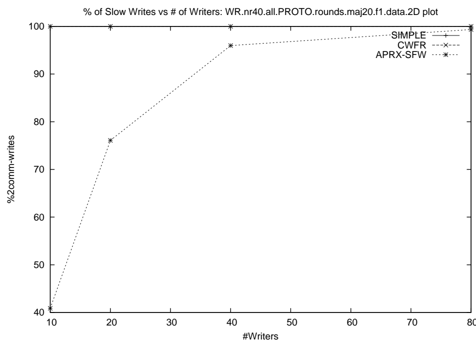


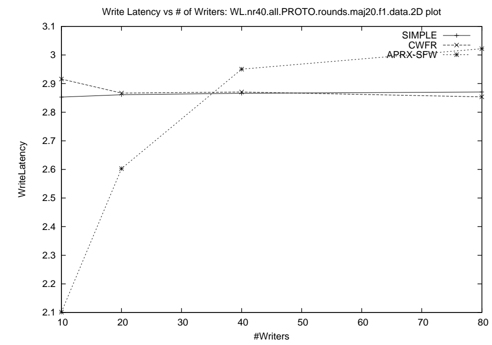
20 Readers:



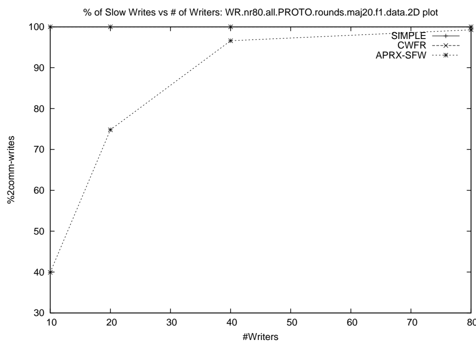


40 Readers:





80 Readers:



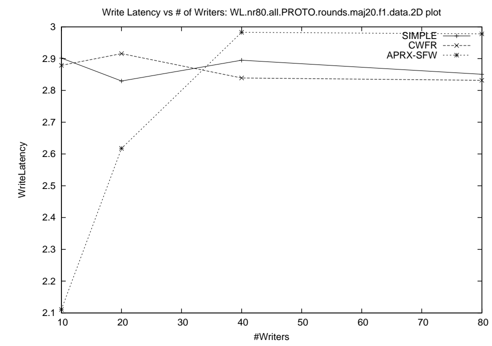
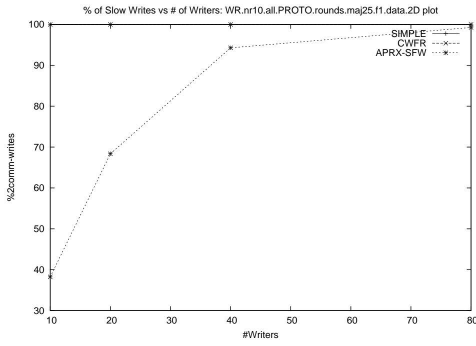
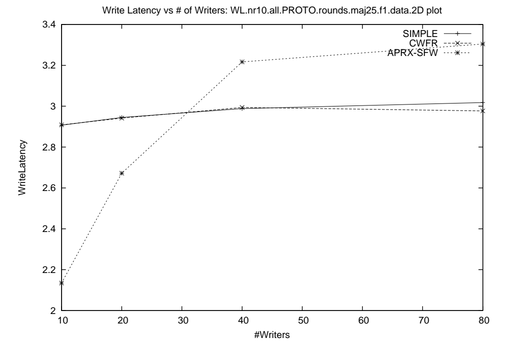


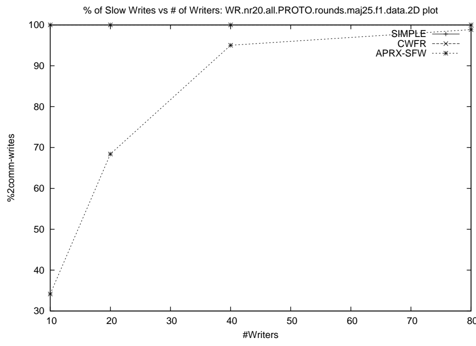
Figure 23: 19-wise quorum system ($|\mathcal{S}| = 20$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

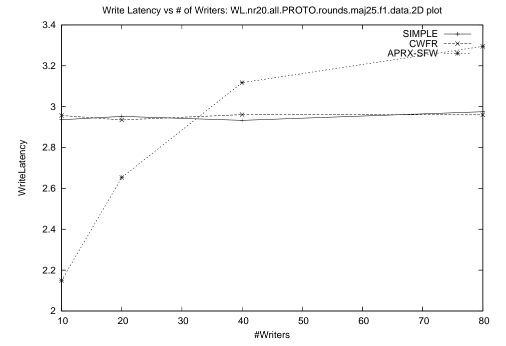
10 Readers:



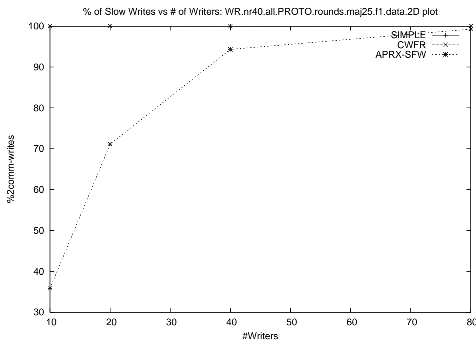


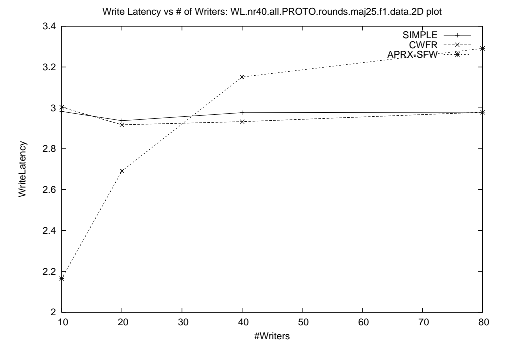
20 Readers:



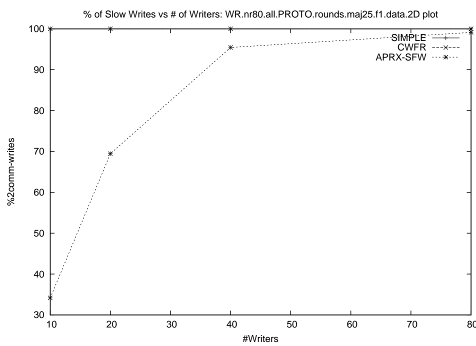


40 Readers:





80 Readers:



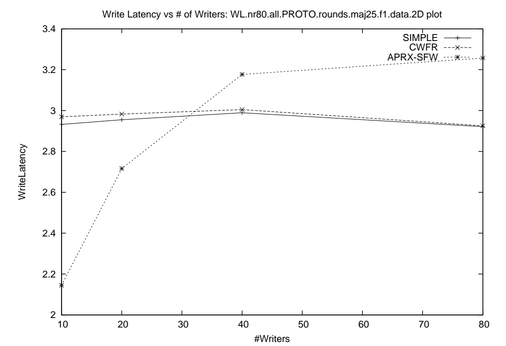
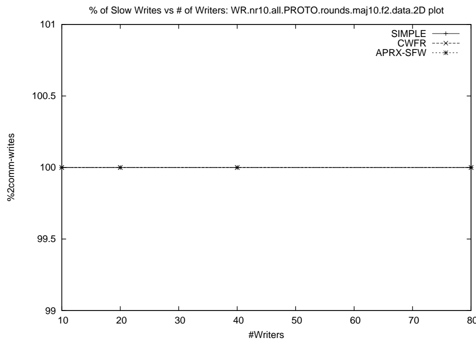
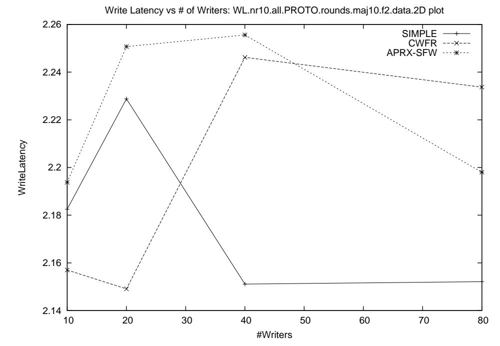


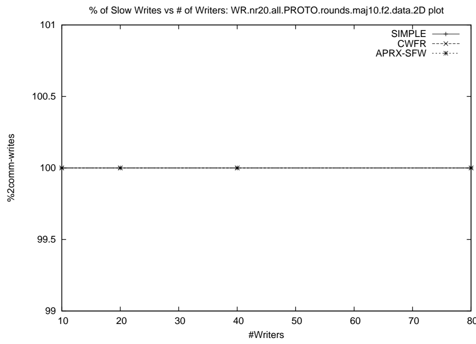
Figure 24: 24-wise quorum system ($|\mathcal{S}| = 25$, $f = 1$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

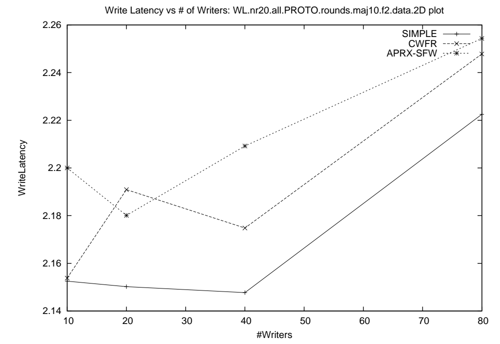
10 Readers:



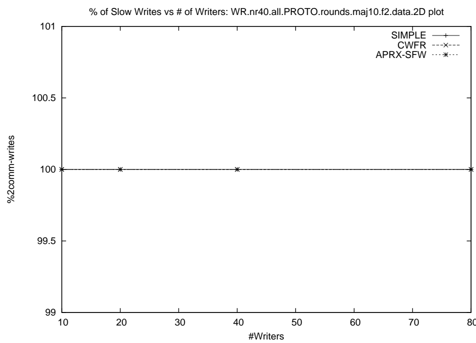


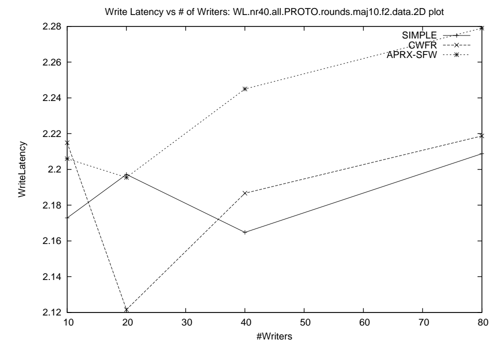
20 Readers:



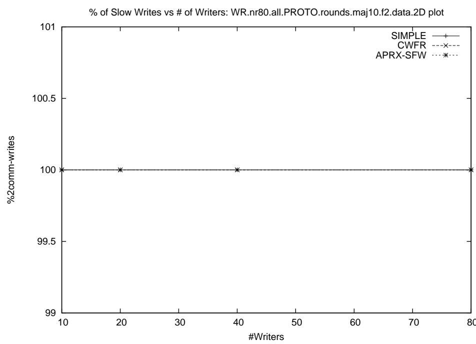


40 Readers:





80 Readers:



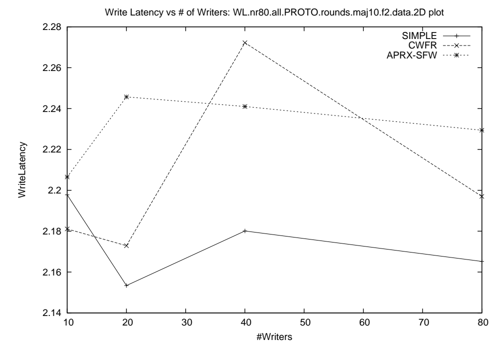
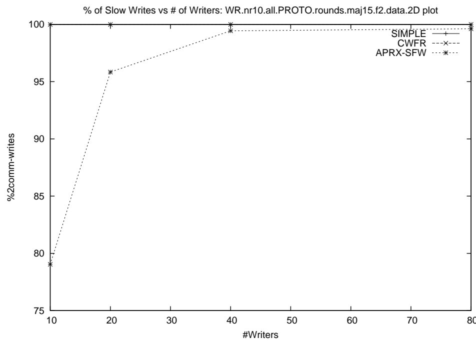
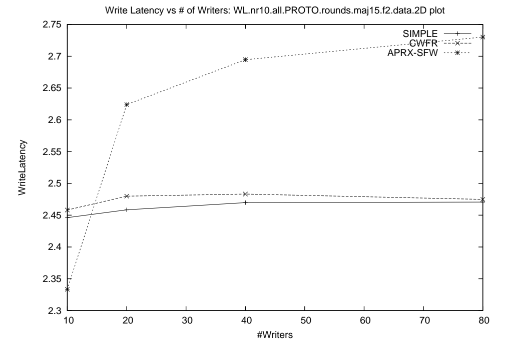


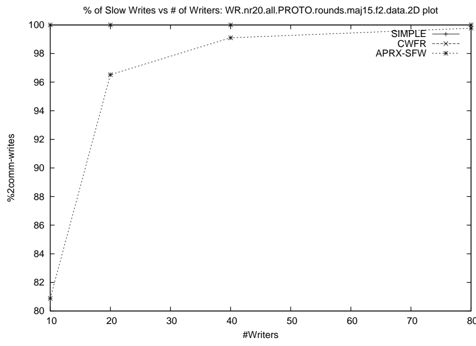
Figure 25: 4-wise quorum system ($|\mathcal{S}| = 10$, $f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

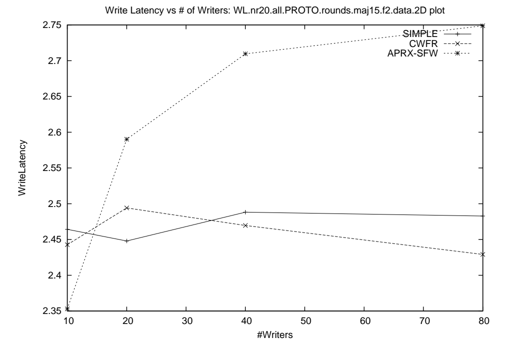
10 Readers:



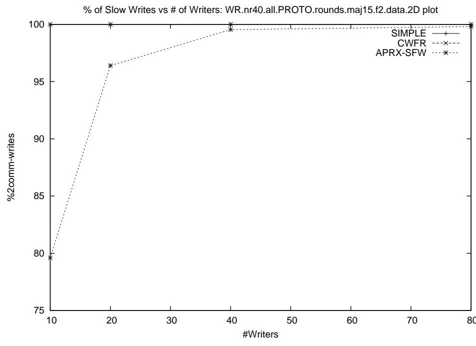


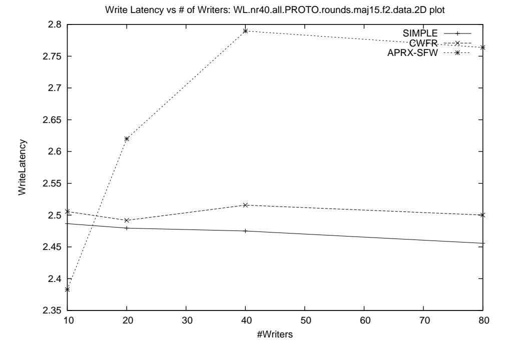
20 Readers:



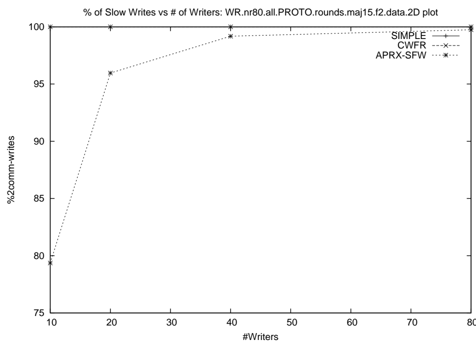


40 Readers:





80 Readers:



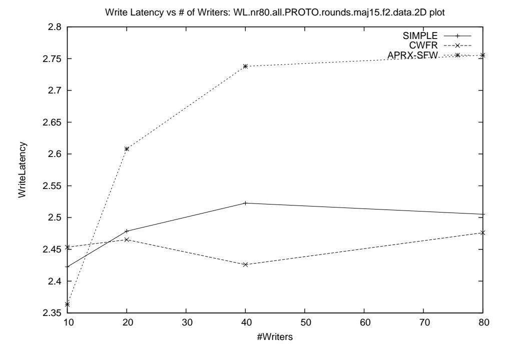
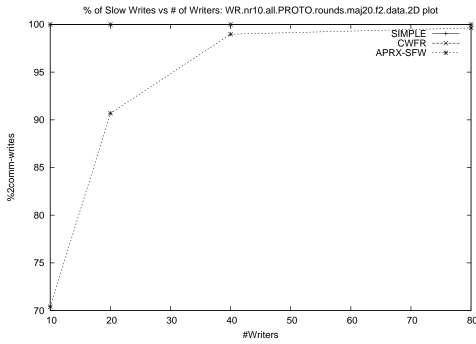
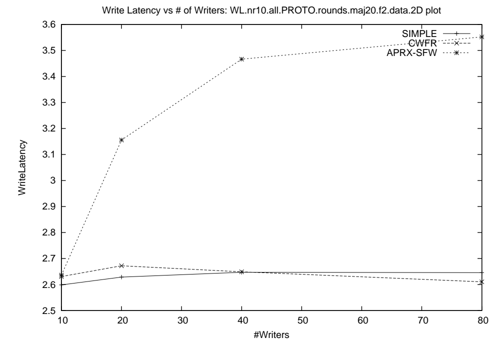


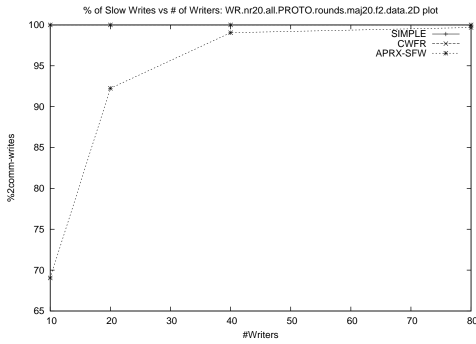
Figure 26: 6-wise quorum system ($|\mathcal{S}| = 15,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

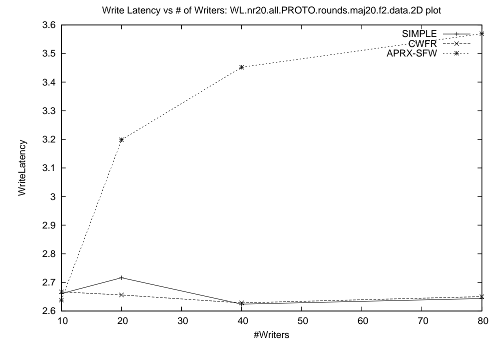
10 Readers:



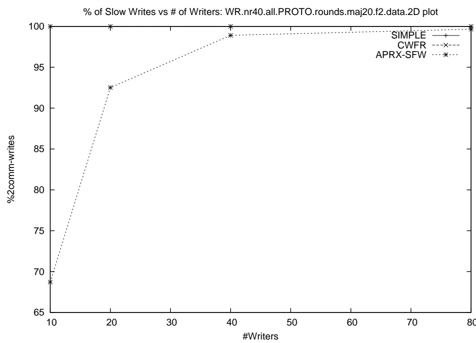


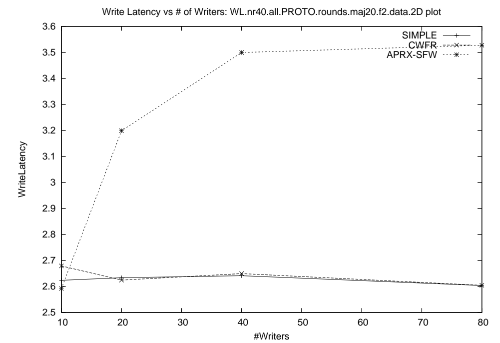
20 Readers:



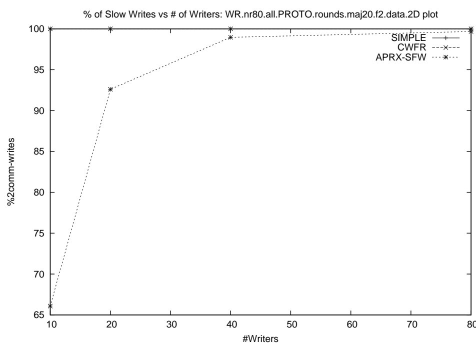


40 Readers:





80 Readers:



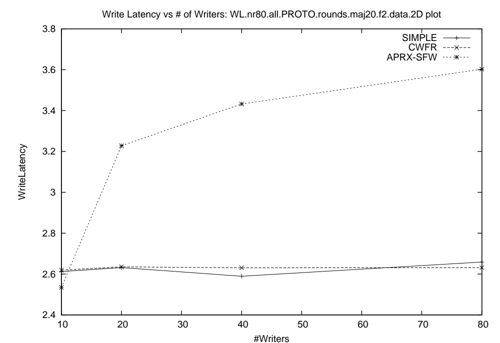
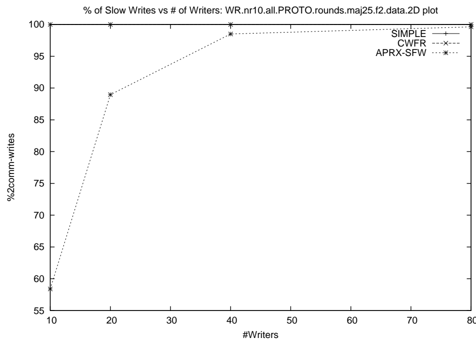
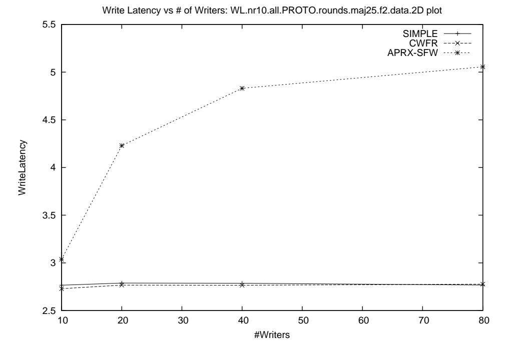


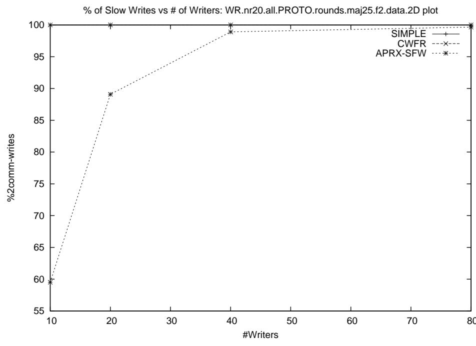
Figure 27: 9-wise quorum system ($|\mathcal{S}| = 20$, $f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

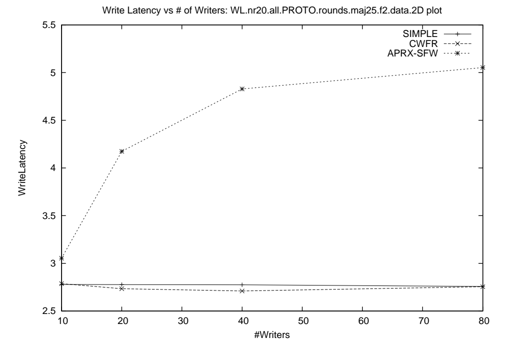
10 Readers:



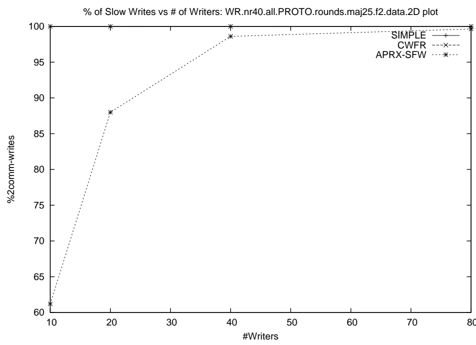


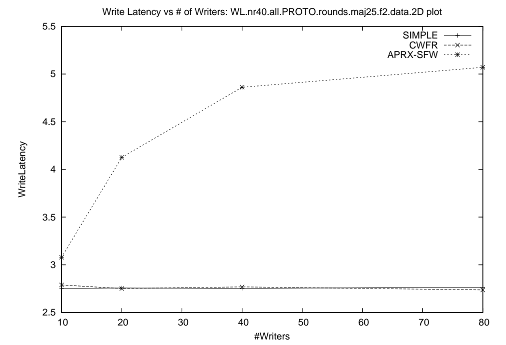
20 Readers:



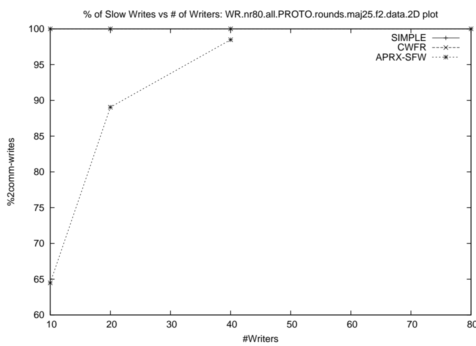


40 Readers:





80 Readers:



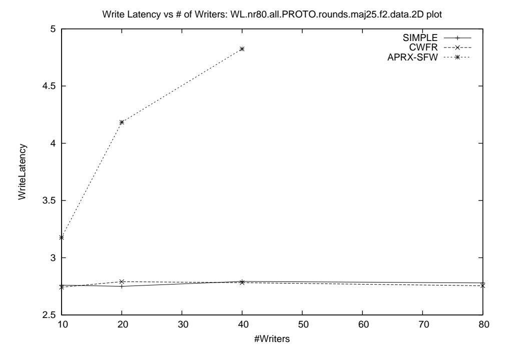
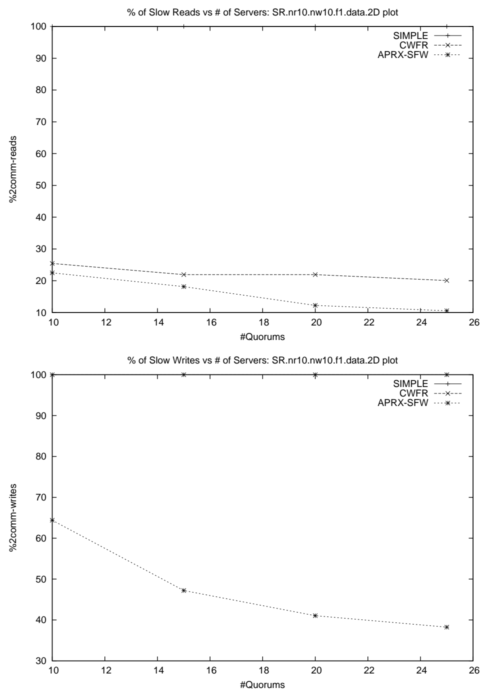


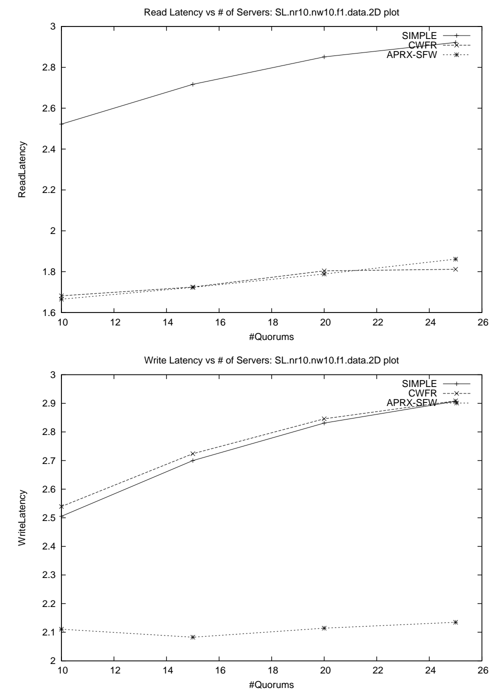
Figure 28: 11-wise quorum system ($|\mathcal{S}| = 25,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

D Operation Performance Relative to Quorum Construction

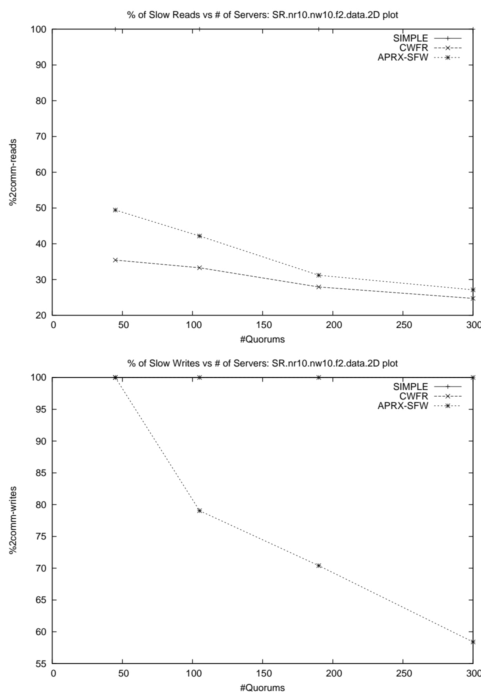
The plots below illustrate the performance of read operations over the number of quorums in the system. For the following graphs we fix the number of reader and writer participants and we change the number of servers in the system. The quorum membership is also changed when we modify the maximum number 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:



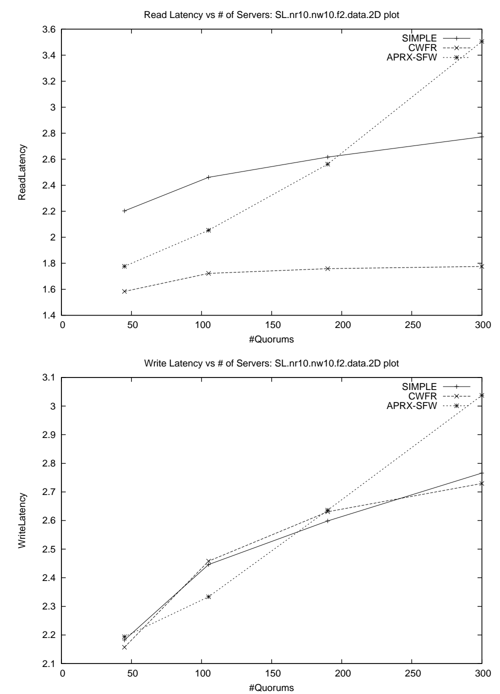
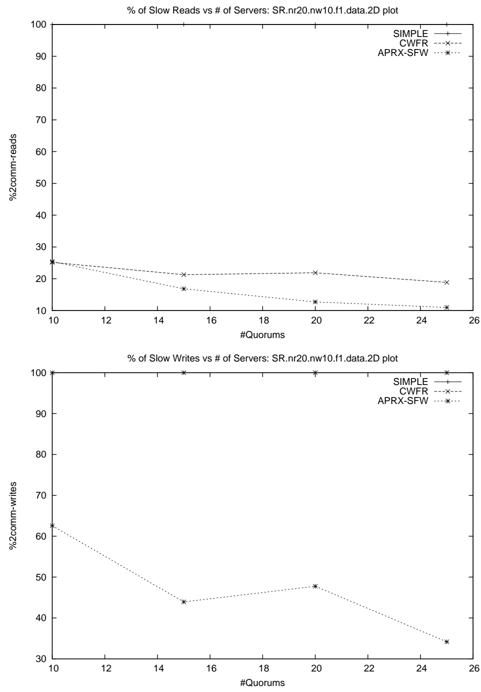
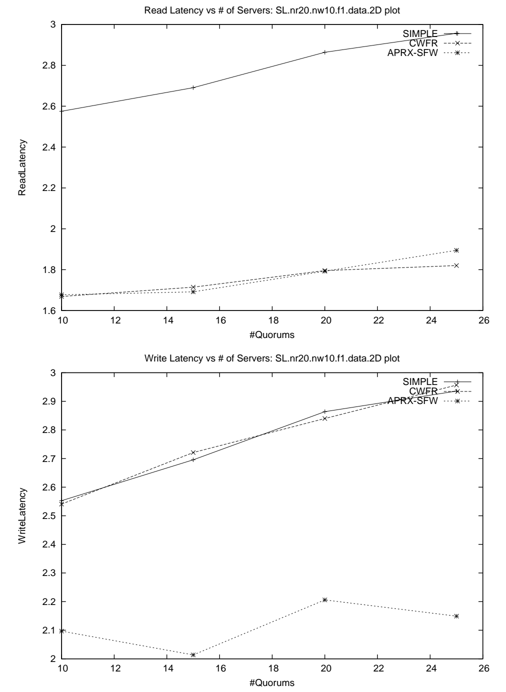


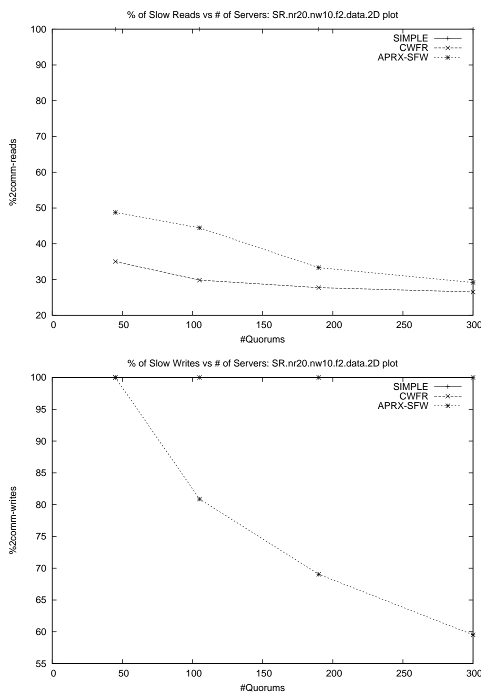
Figure 29: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 10 Writers, f=1:





20 Readers, 10 Writers, f=2:



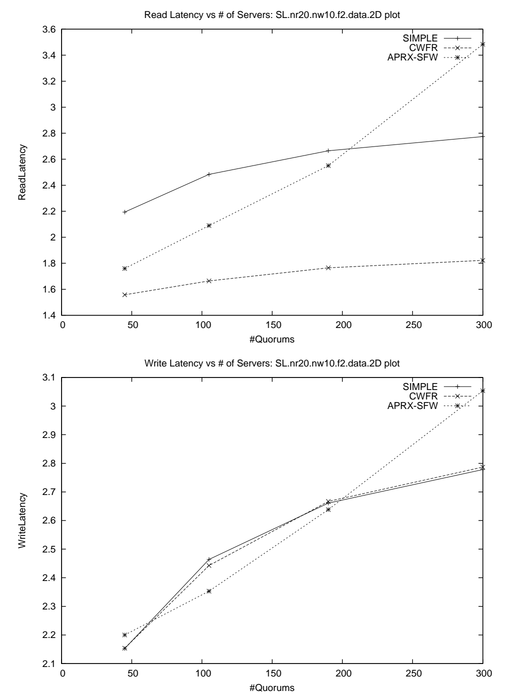
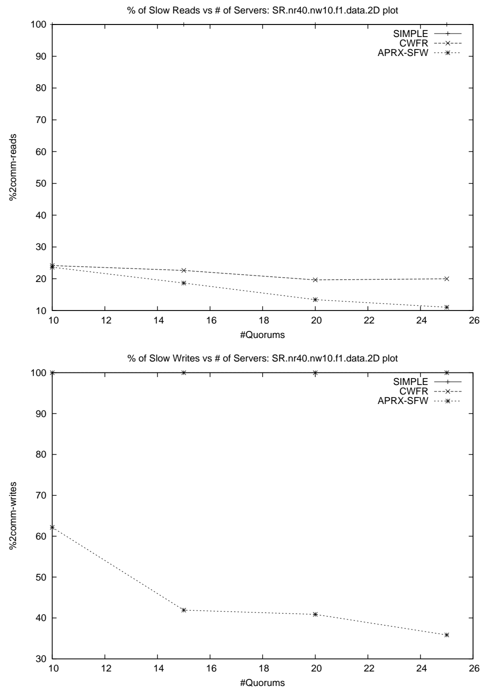
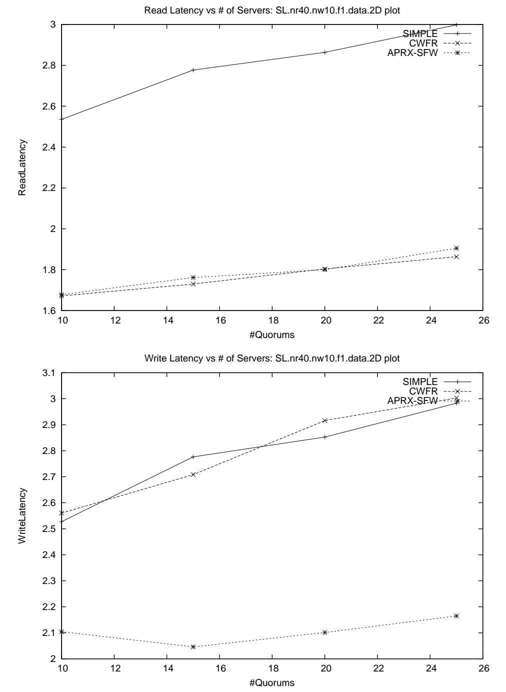


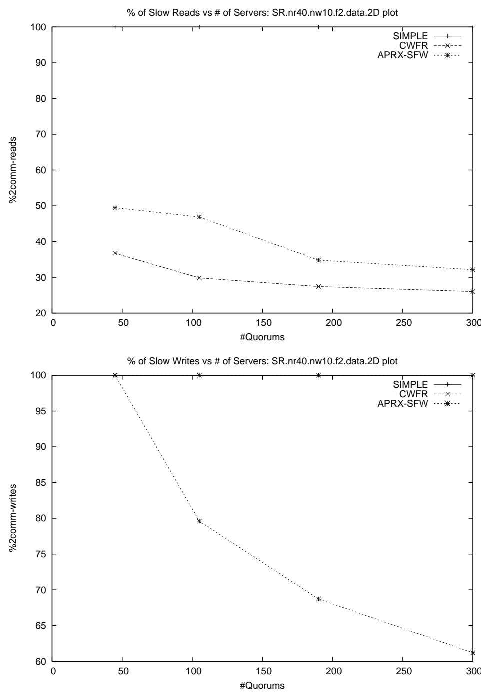
Figure 30: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 10 Writers, f=1:





40 Readers, 10 Writers, f=2:



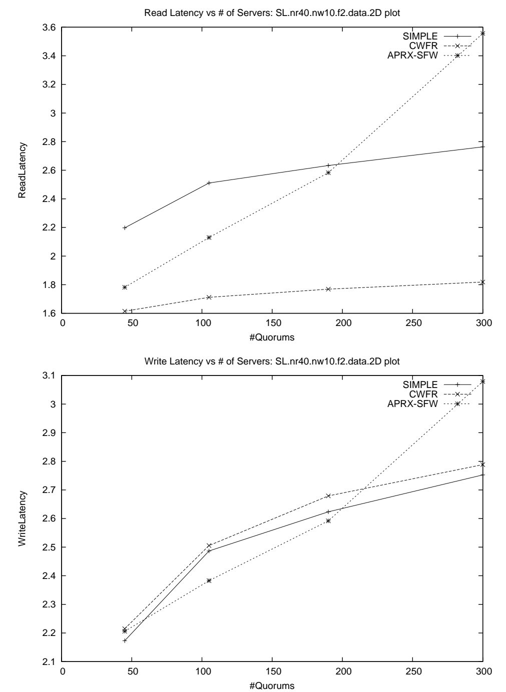
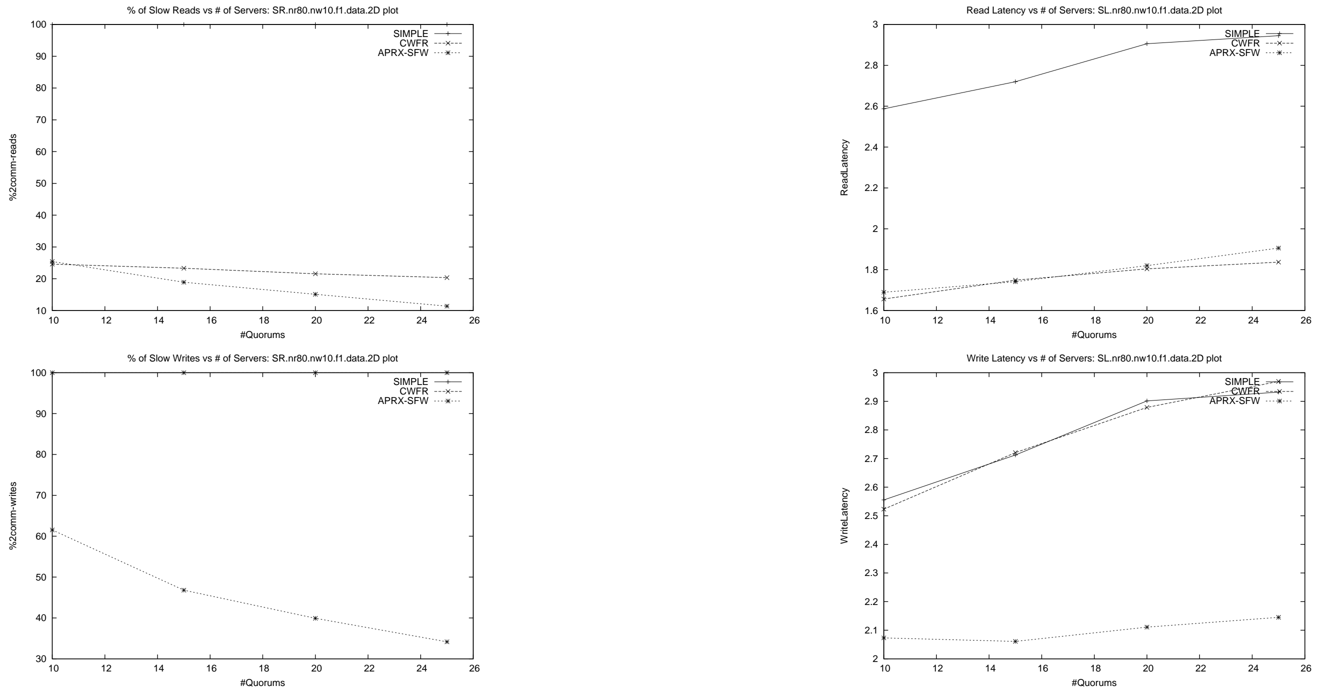


Figure 31: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

80 Readers, 10 Writers, f=1:



80 Readers, 10 Writers, f=2:

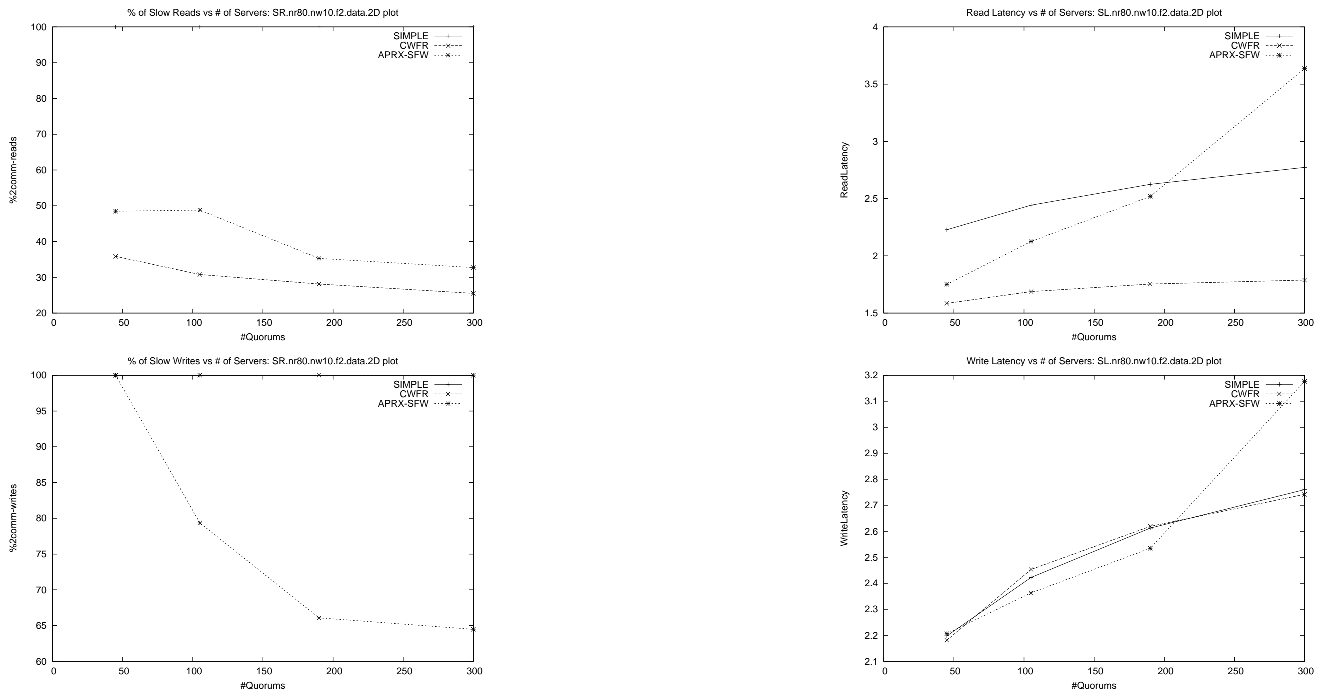
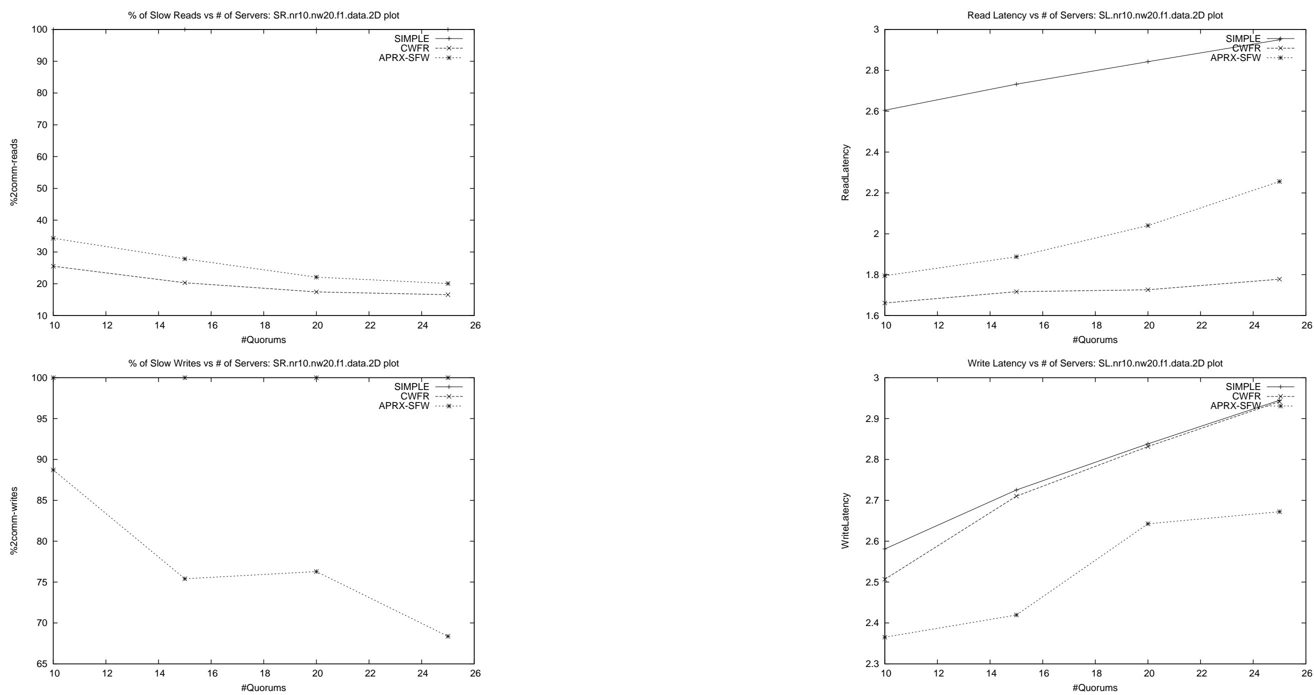


Figure 32: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

10 Readers, 20 Writers, f=1:



10 Readers, 20 Writers, f=2:

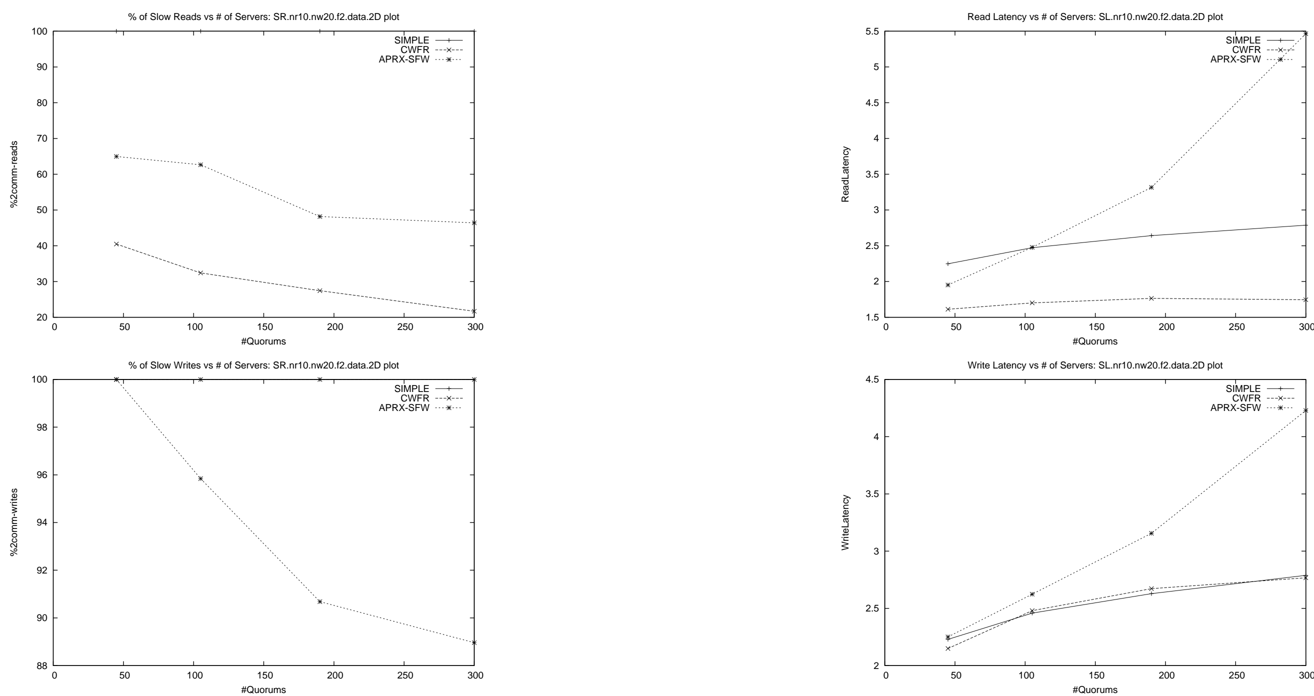
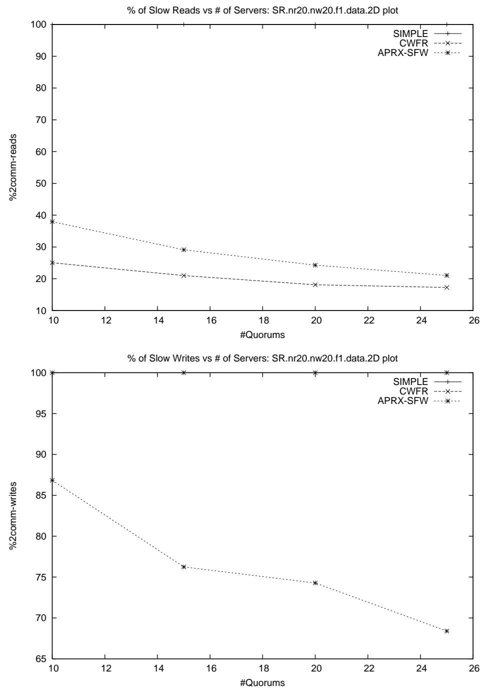
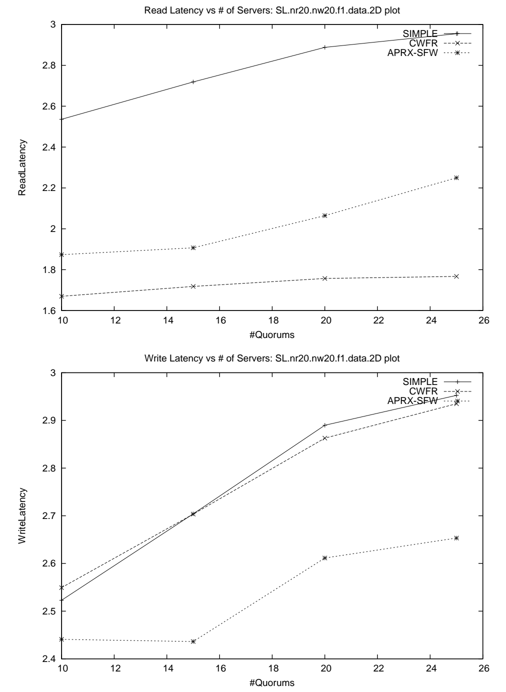


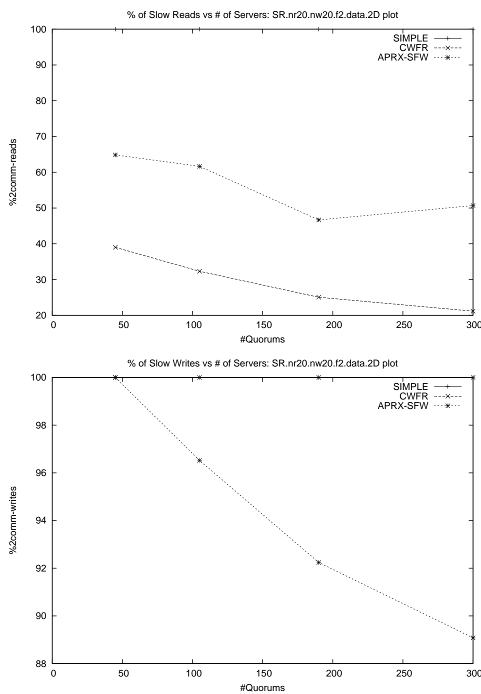
Figure 33: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 20 Writers, f=1:





20 Readers, 20 Writers, f=2:



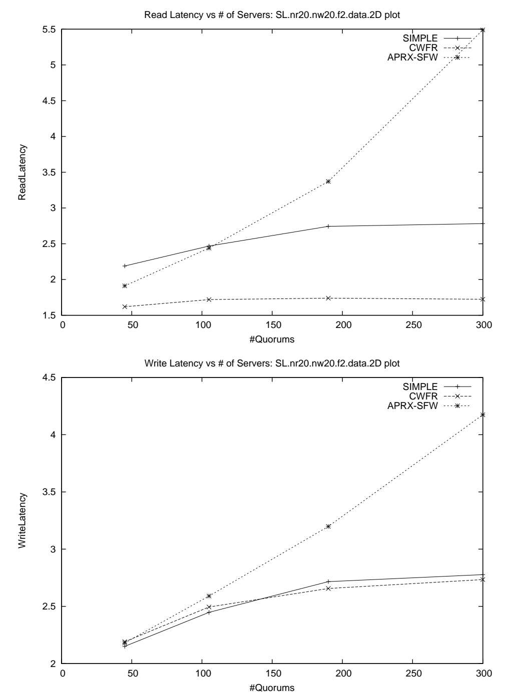
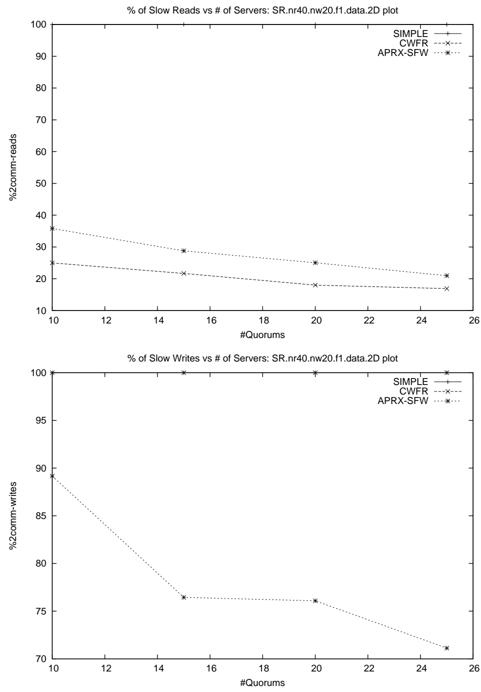
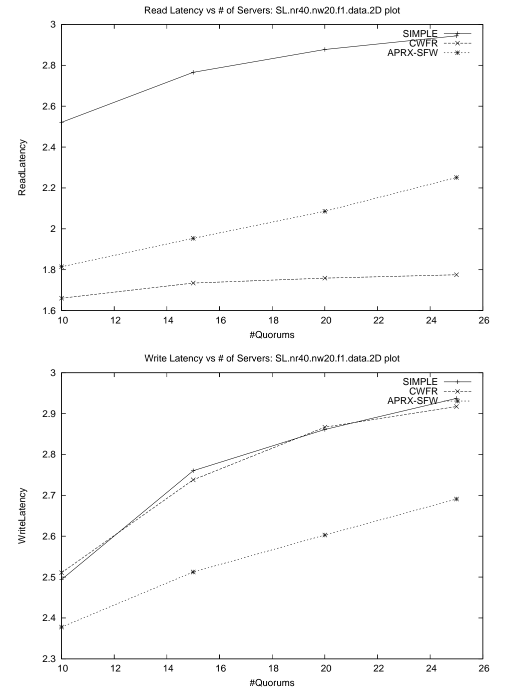


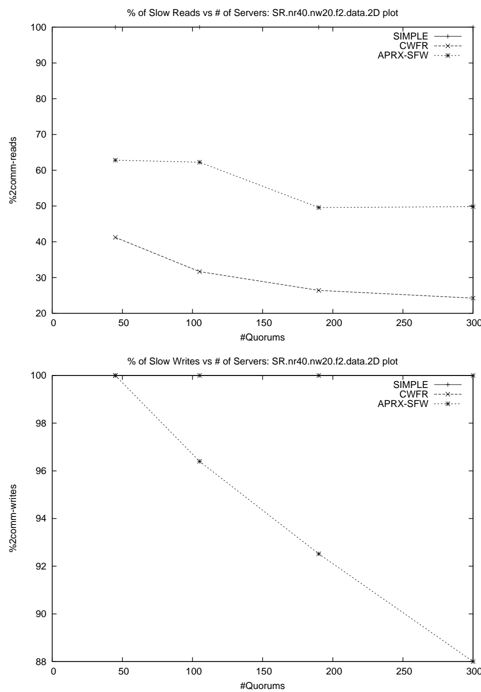
Figure 34: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 20 Writers, f=1:





40 Readers, 20 Writers, f=2:



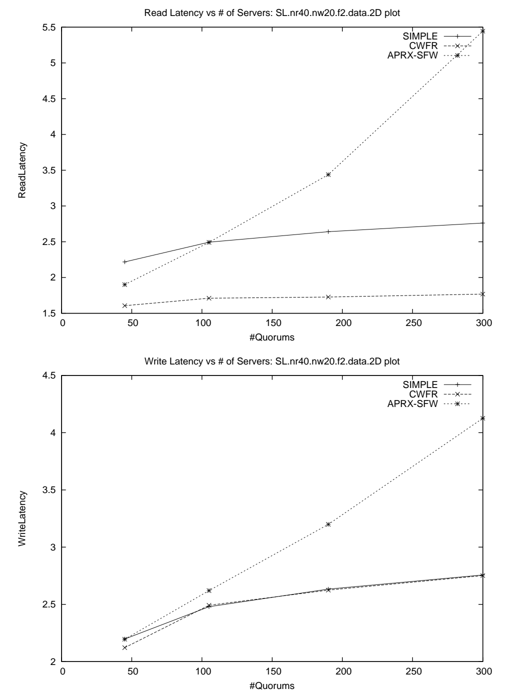
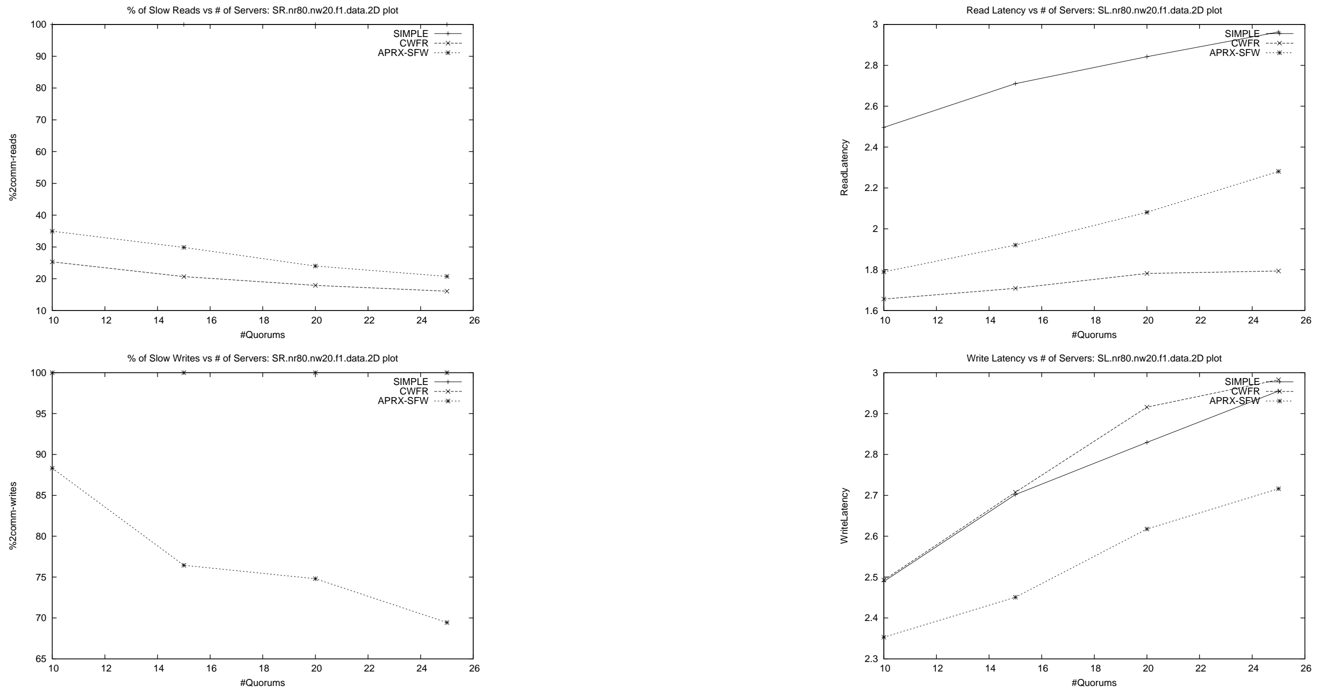


Figure 35: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

80 Readers, 20 Writers, f=1:



80 Readers, 20 Writers, f=2:

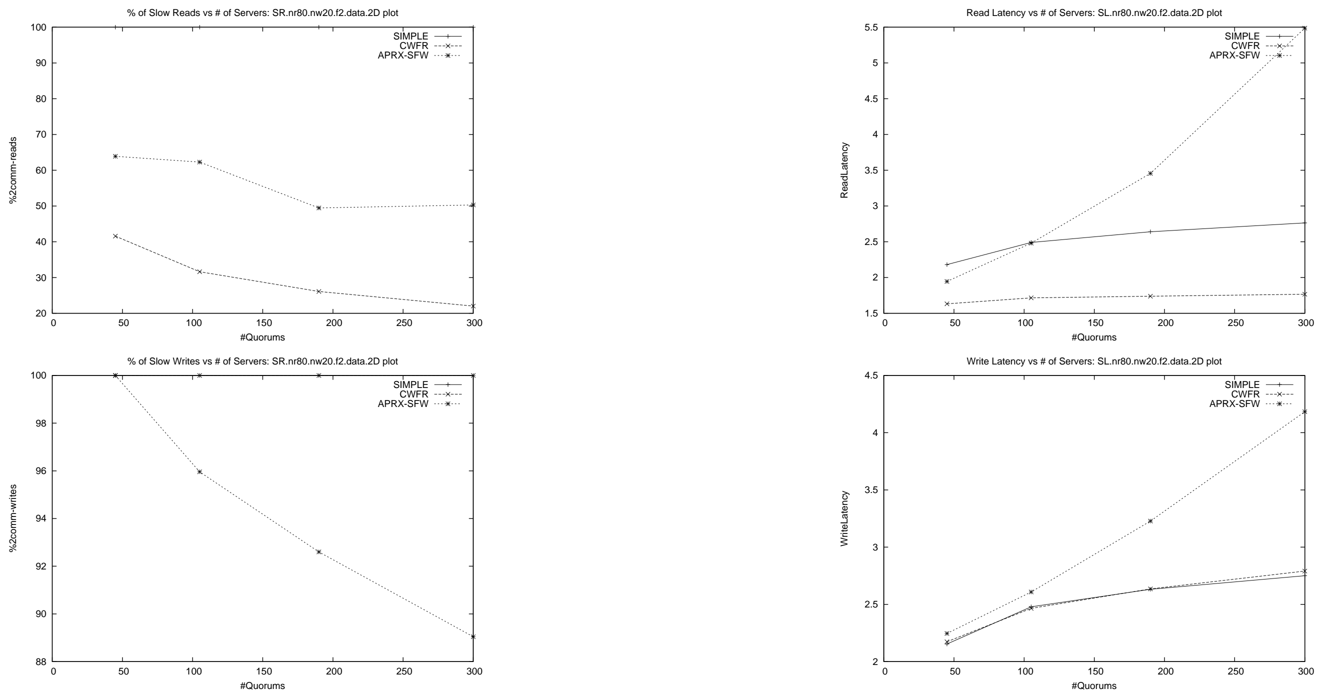
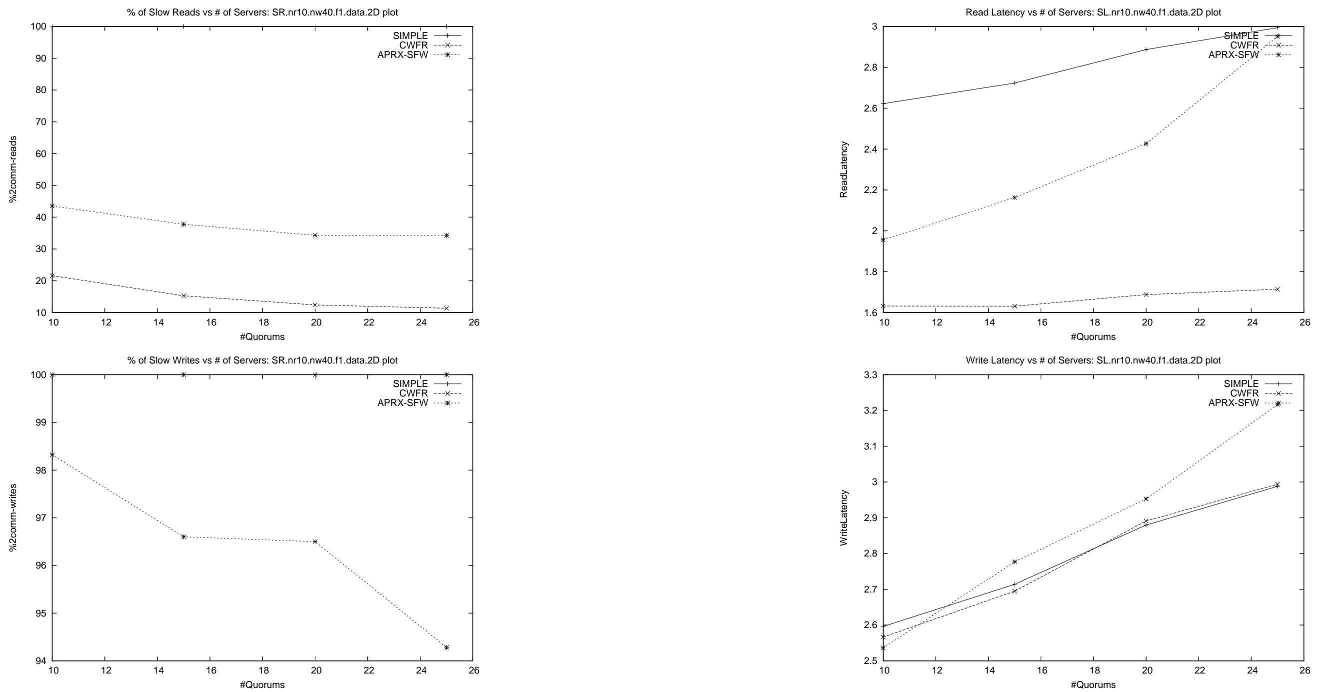


Figure 36: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

10 Readers, 40 Writers, f=1:



10 Readers, 20 Writers, f=2:

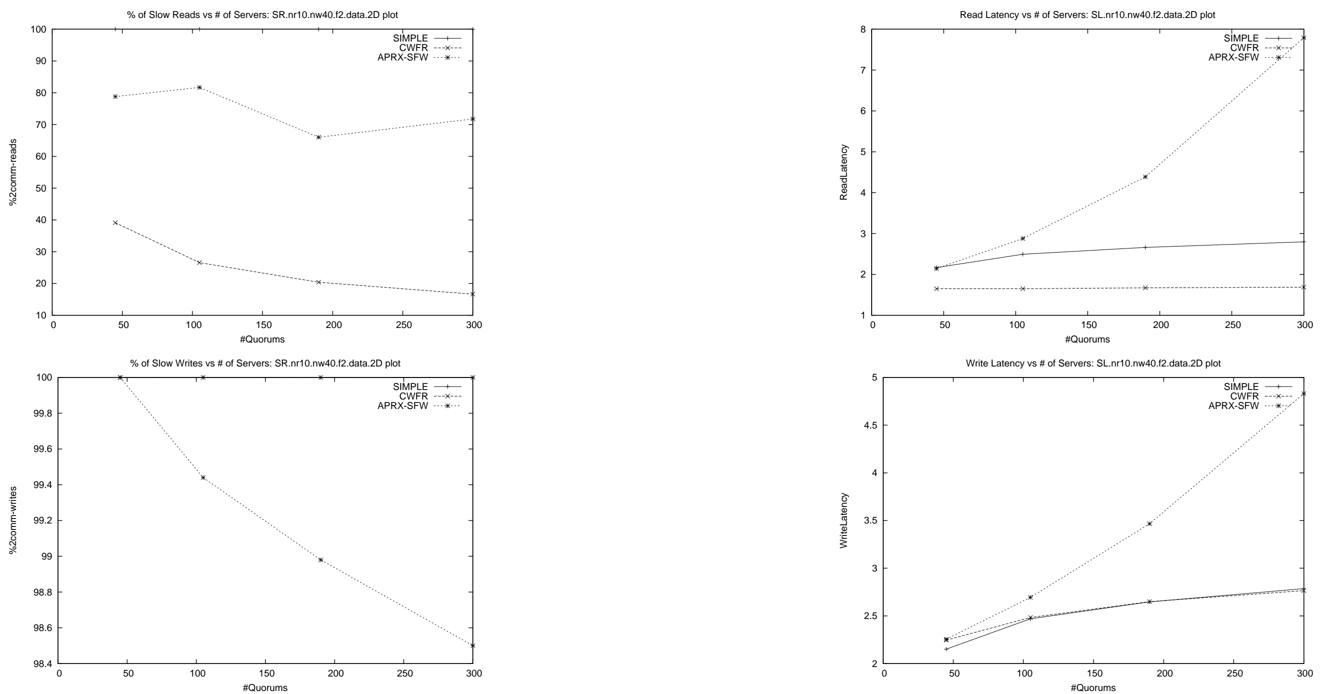
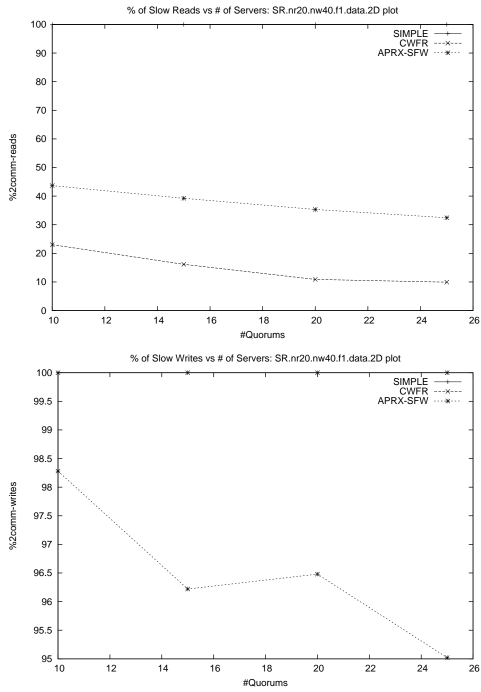
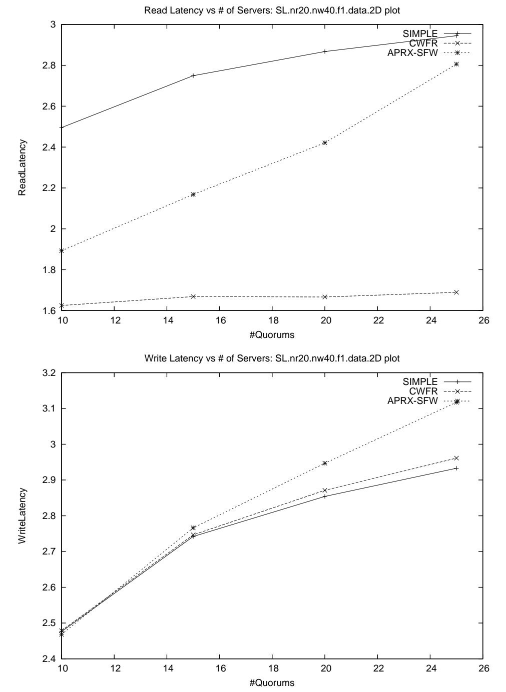


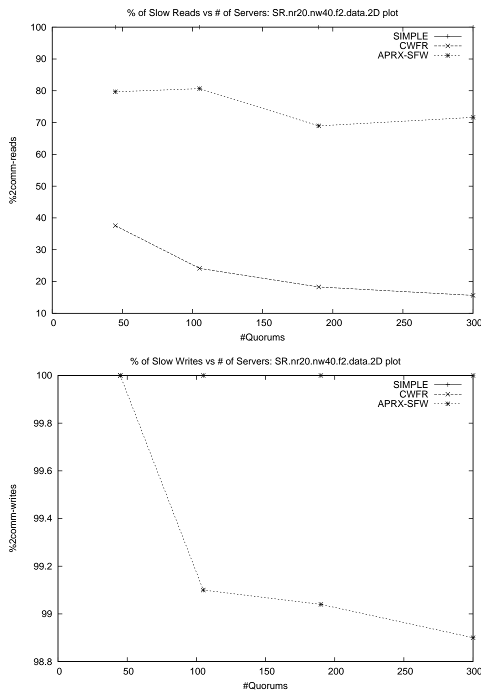
Figure 37: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 40 Writers, f=1:





20 Readers, 40 Writers, f=2:



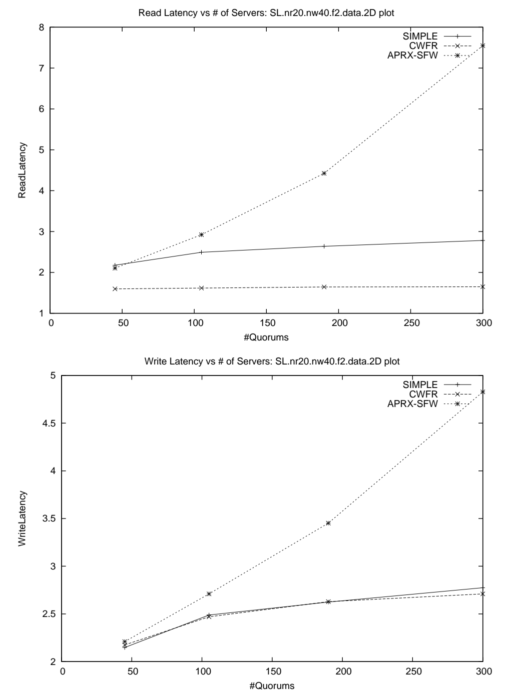
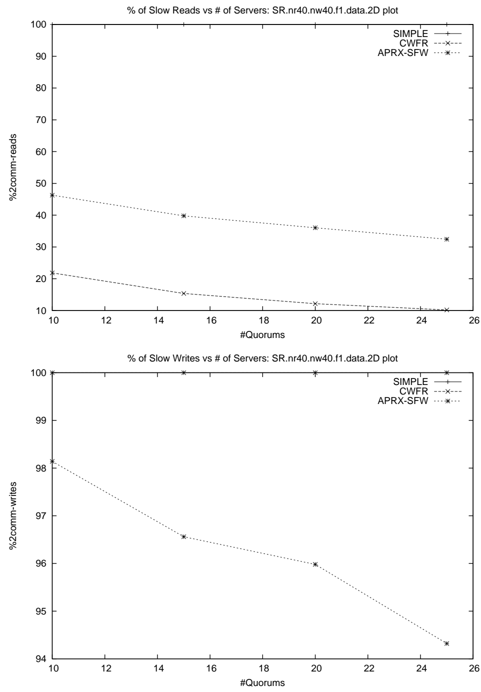
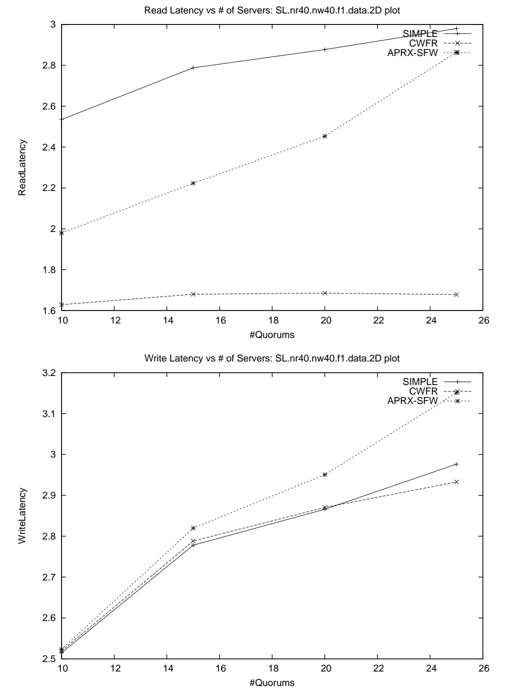


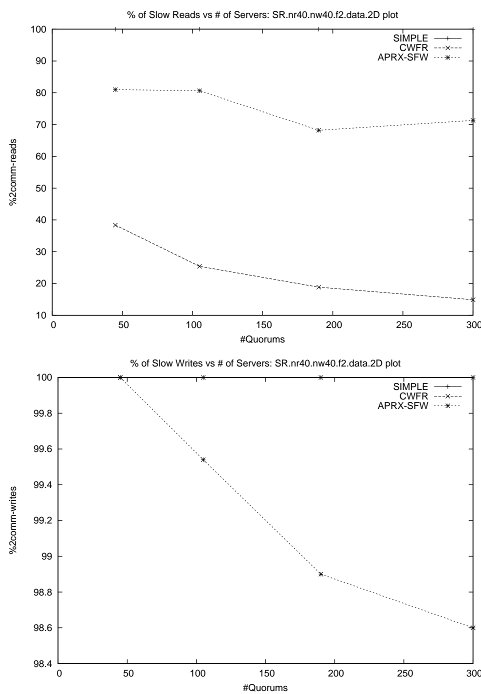
Figure 38: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 40 Writers, f=1:





40 Readers, 40 Writers, f=2:



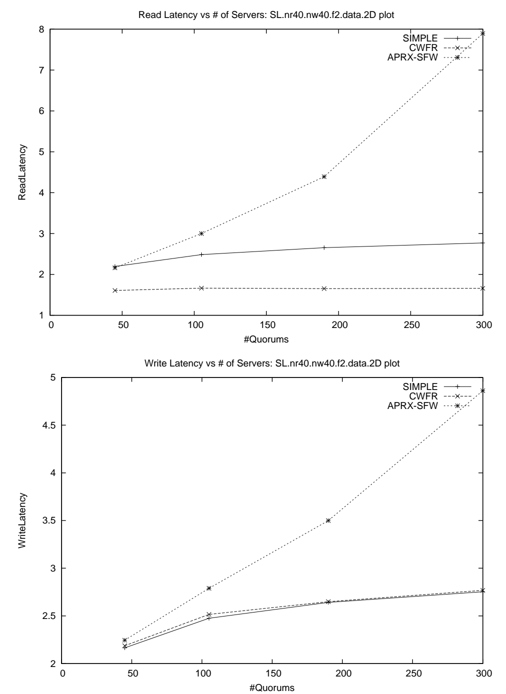
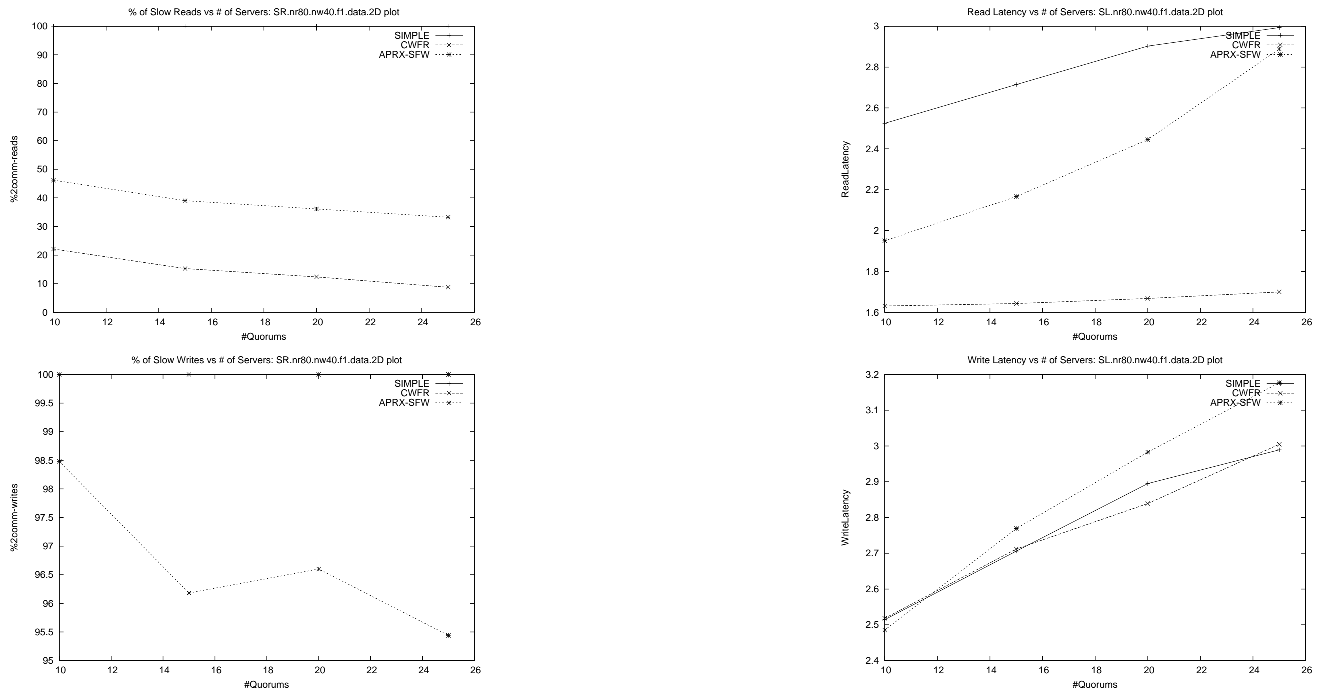


Figure 39: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

80 Readers, 40 Writers, f=1:



80 Readers, 40 Writers, f=2:

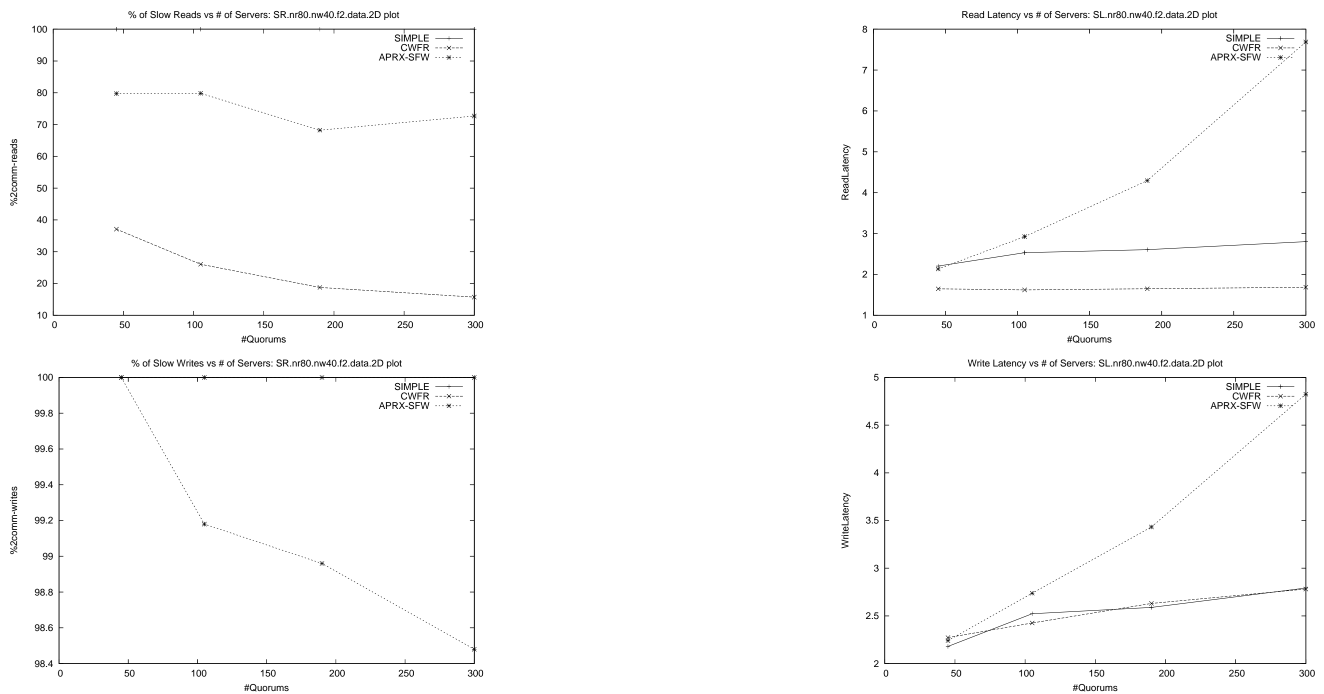
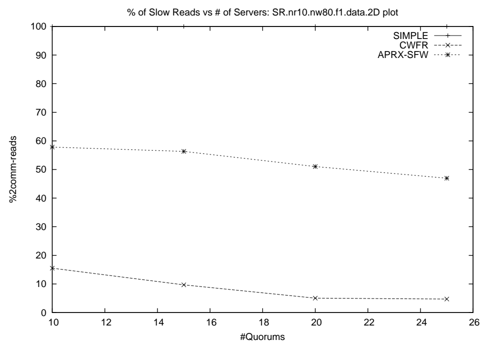
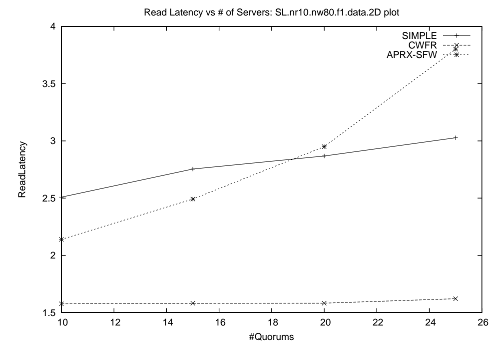
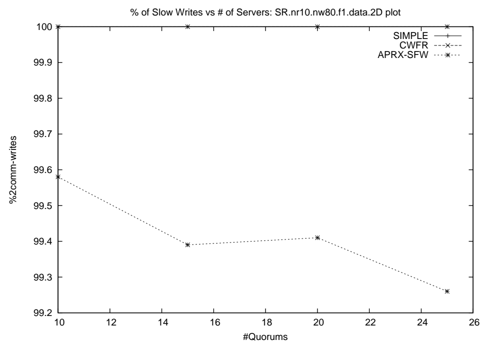


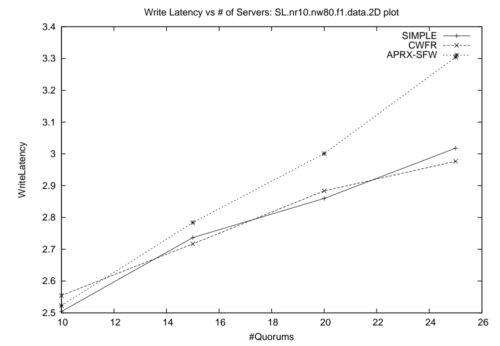
Figure 40: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

10 Readers, 80 Writers, f=1:

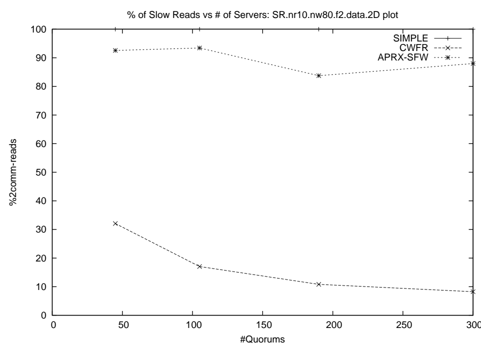


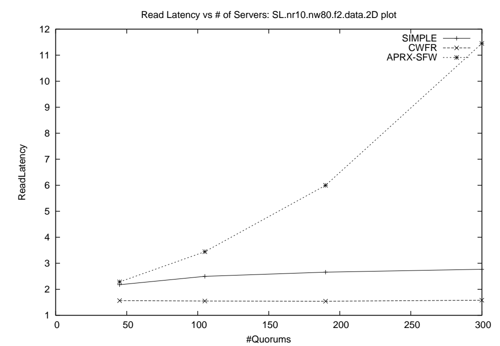


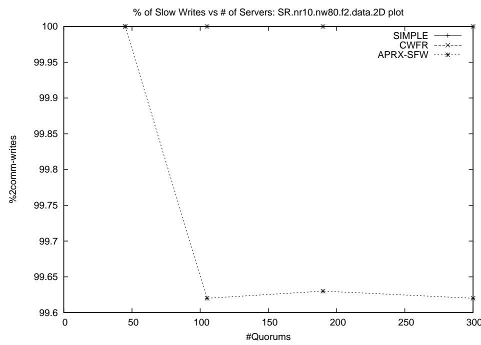




10 Readers, 80 Writers, f=2:







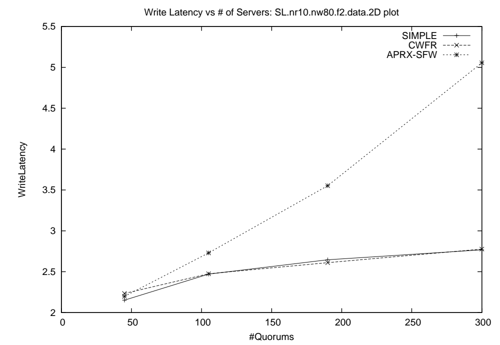
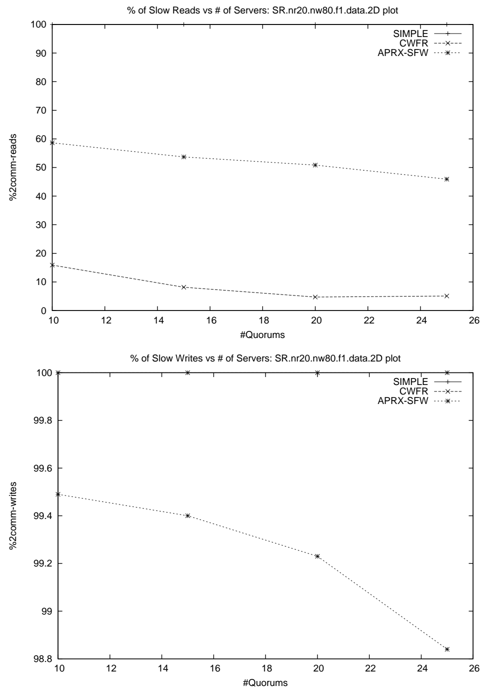
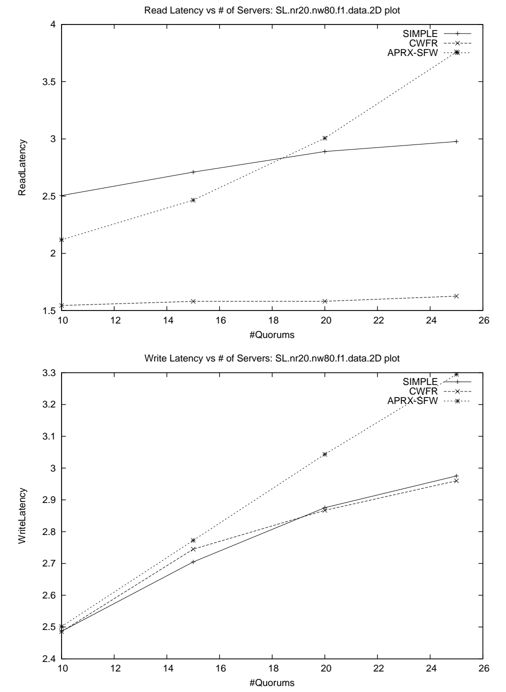


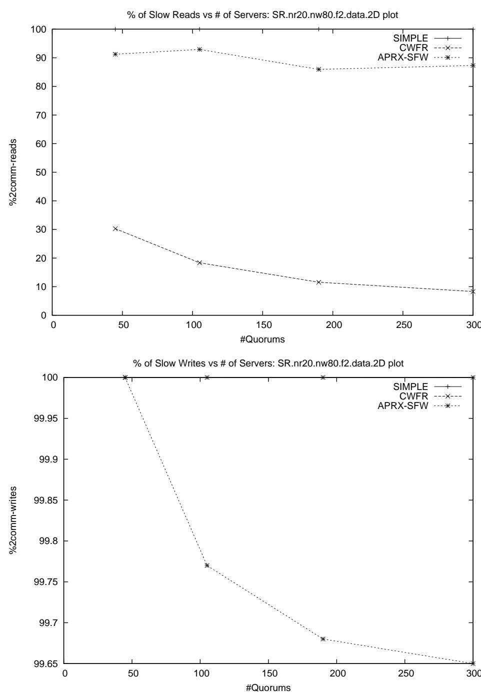
Figure 41: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

20 Readers, 80 Writers, f=1:





20 Readers, 80 Writers, f=2:



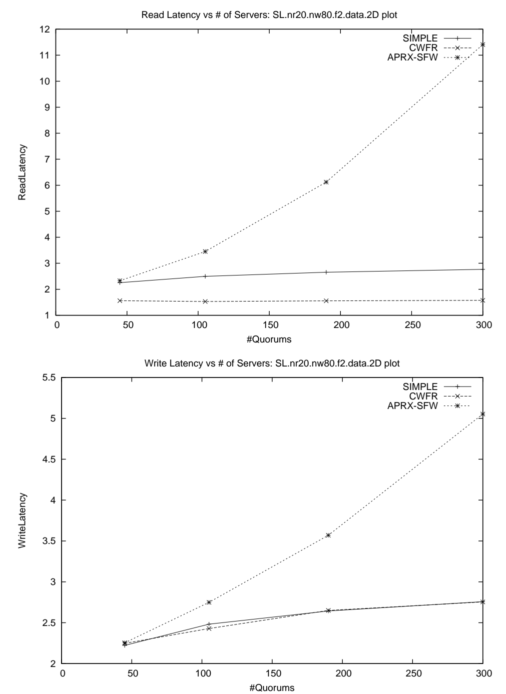
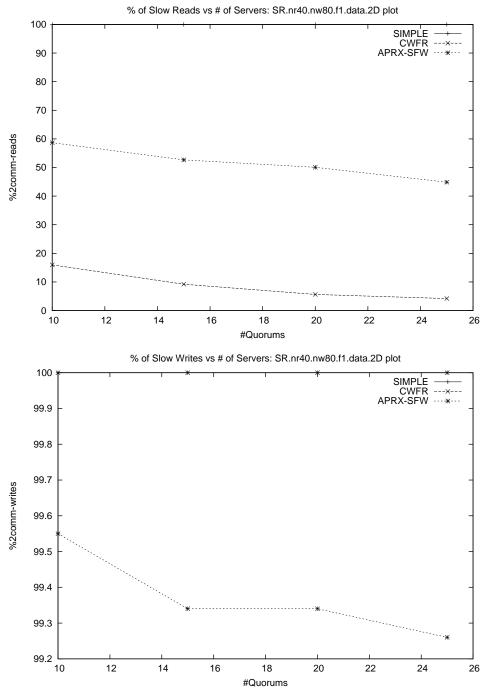
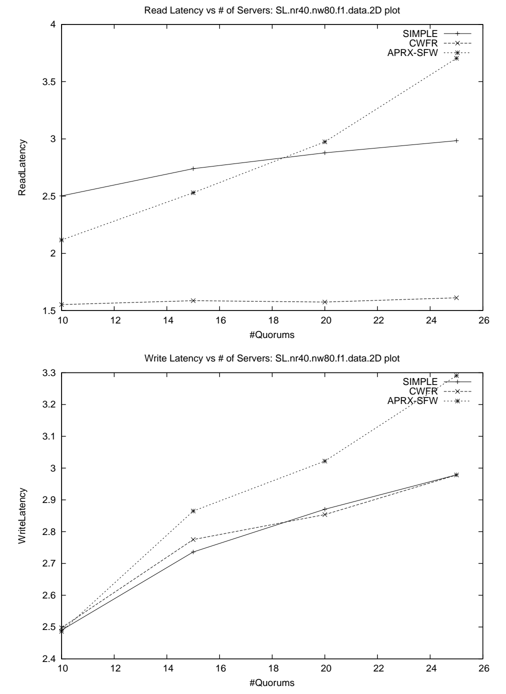


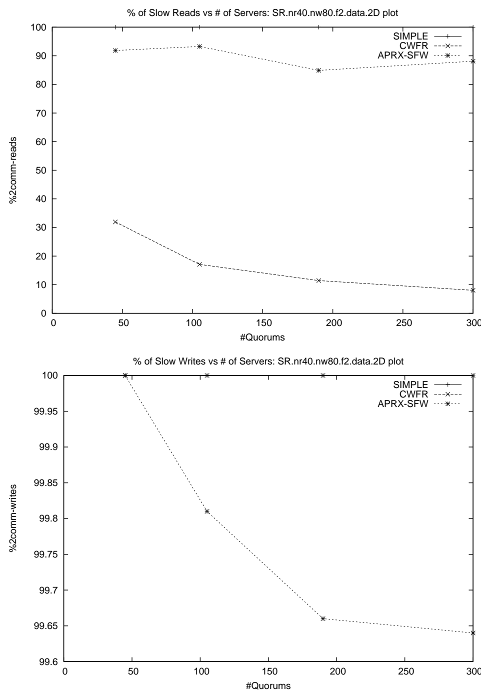
Figure 42: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

40 Readers, 80 Writers, f=1:





40 Readers, 80 Writers, f=2:



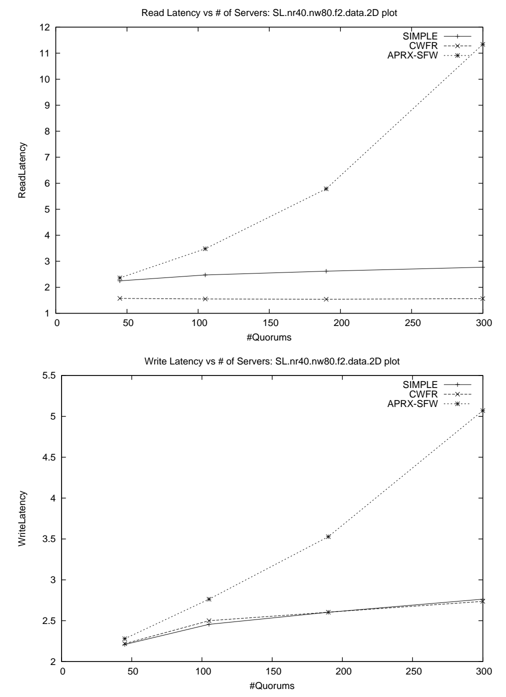
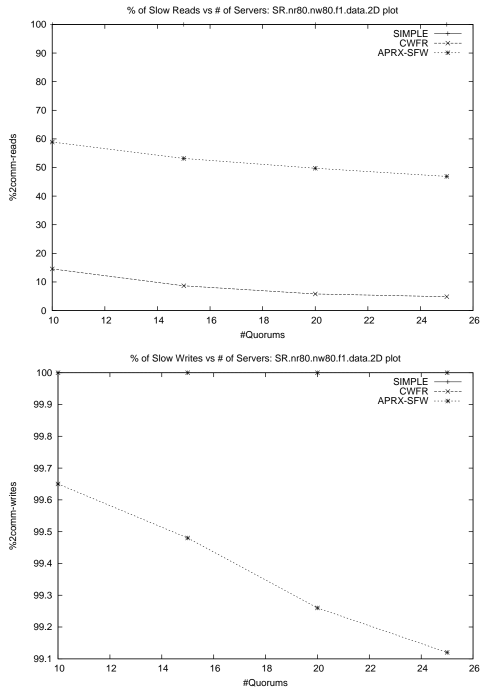
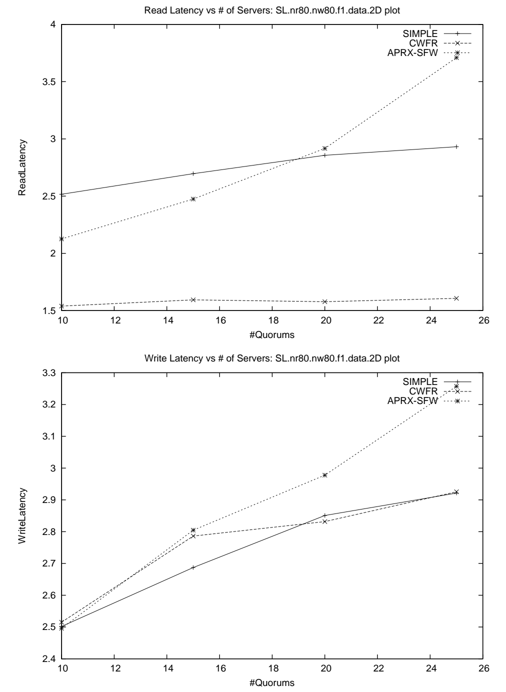


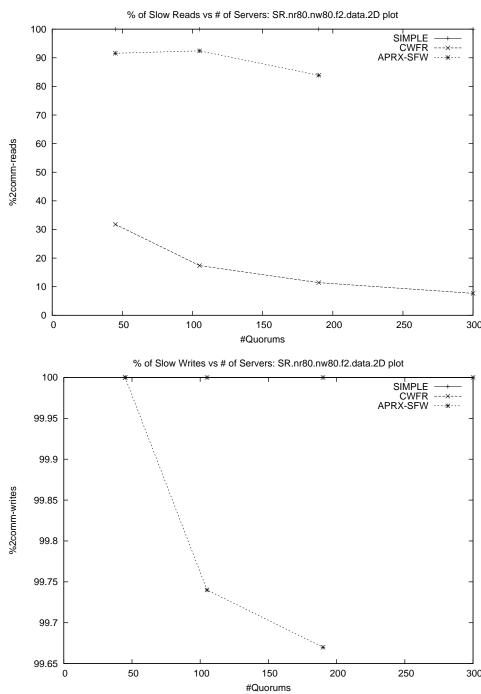
Figure 43: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

80 Readers, 80 Writers, f=1:





80 Readers, 80 Writers, f=2:



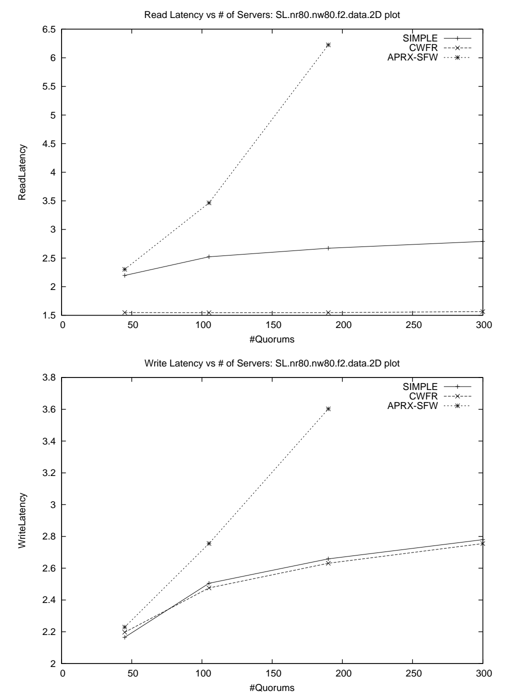
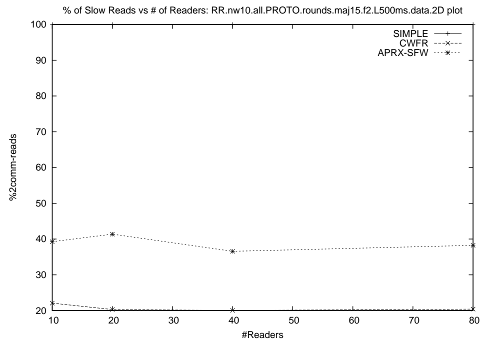


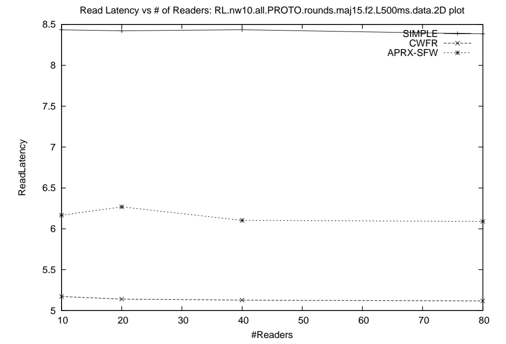
Figure 44: **Left Column:** Percentage of slow operations, **Right Column:** Latency of operations

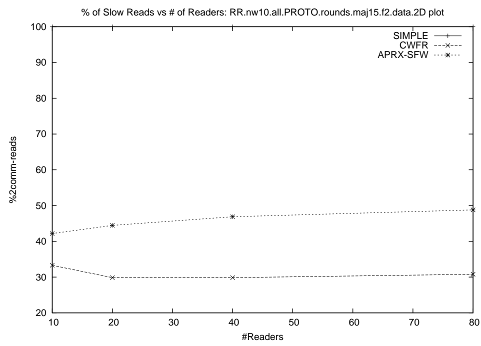
E Network Latency 500ms

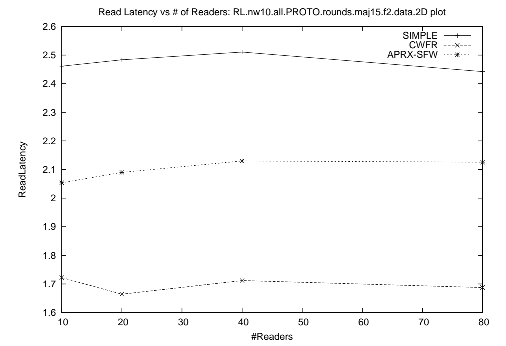
Below we present the plots regarding the read and write performance of the algorithms when the network experiences a latency of 500ms. We focused on a single quorum construction over a set of 15 servers, 2 of which may crash. Each quorum has size 13 and thus the quorum system has 105 quorum members. We run the Simple, CwFr, and Aprx-Sfw algorithms using 10,20, 40, and 80 readers and writers. By fixing the intersection degree and the number of writers (resp. readers) a plot depicts the performance of read (resp. write) operations as we increase the number of readers in the system. Such plots help us determine whether high netwrok latencies may favor algorithms that allow single round operations, even if they have higher computation demands.

10 Writers:

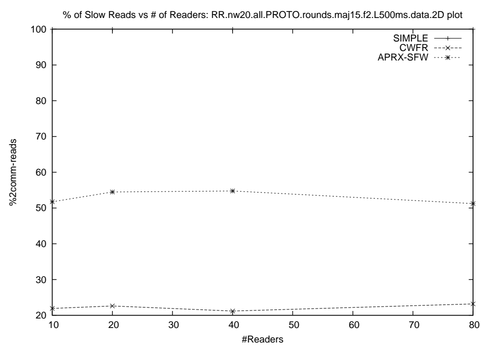


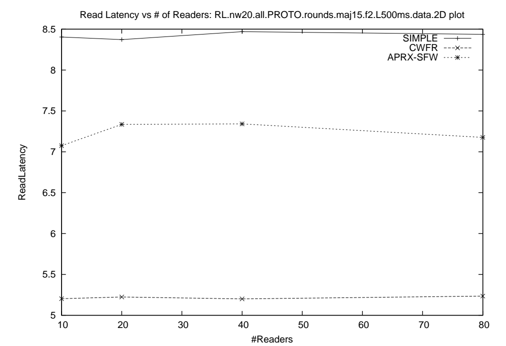


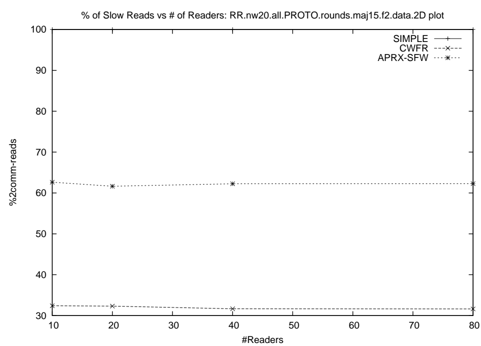




20 Writers:







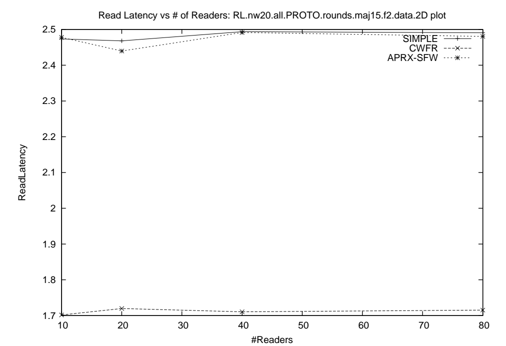
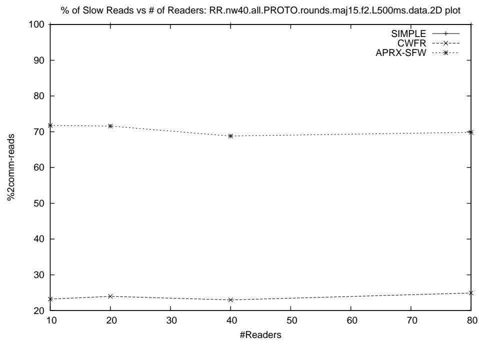
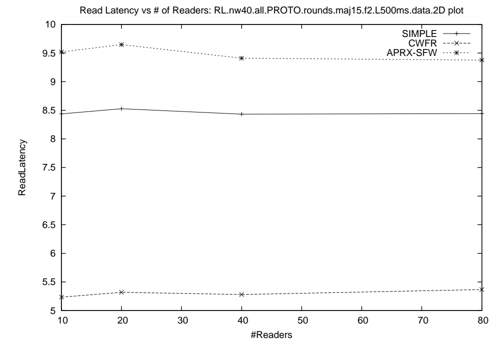
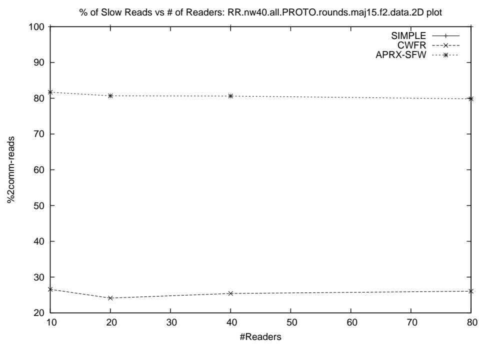


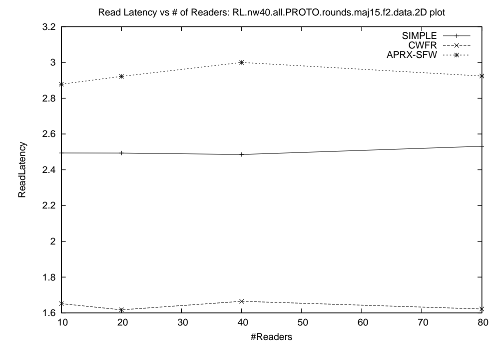
Figure 45: 6-wise quorum system ($|\mathcal{S}| = 15$, $f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

40 Writers:

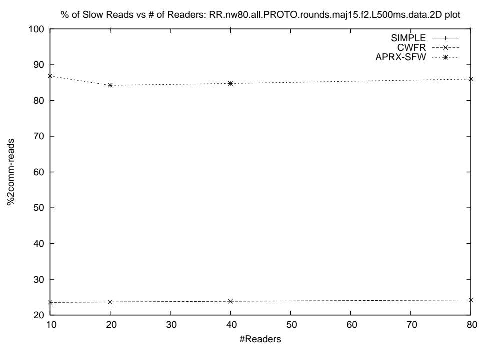


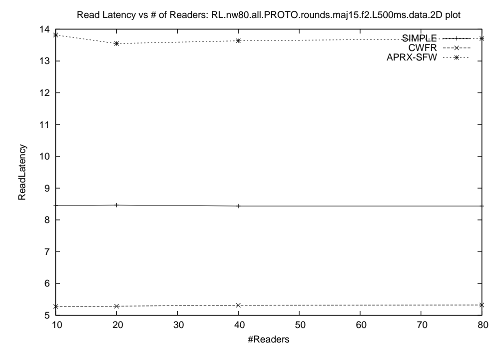


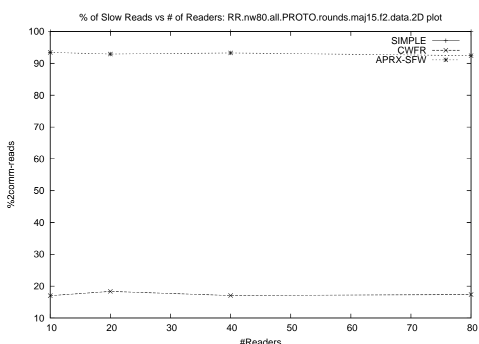




80 Writers:







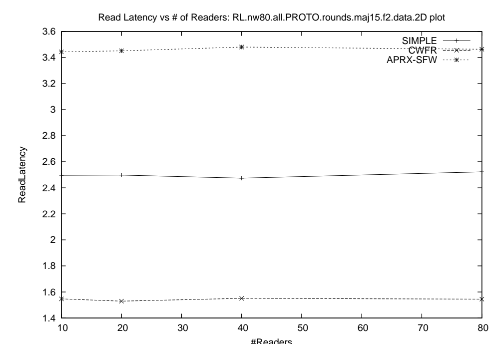
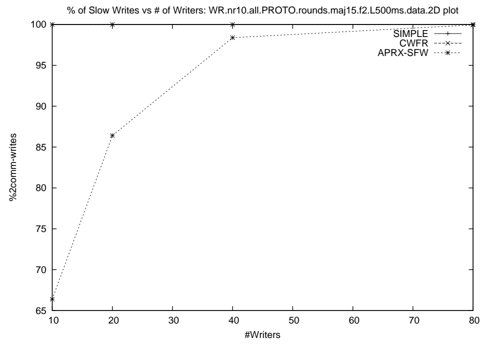
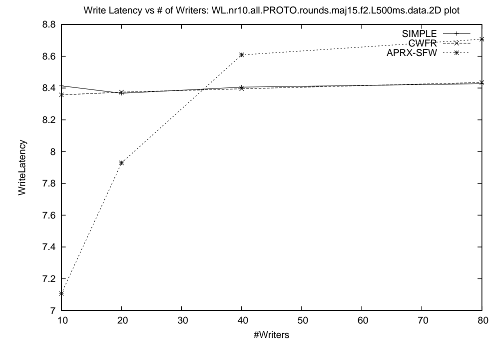
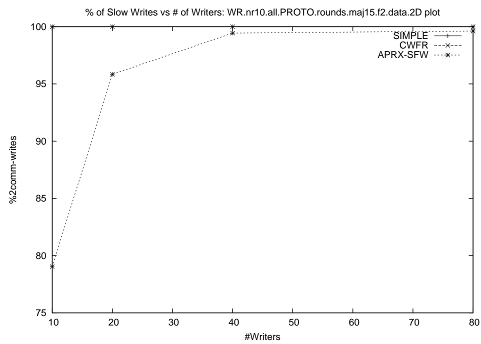


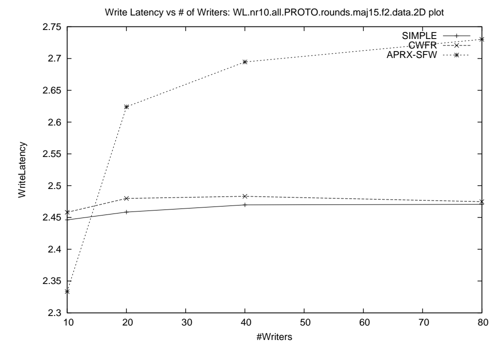
Figure 46: 6-wise quorum system ($|\mathcal{S}| = 15,\ f = 2$): **Left Column:** Percentage of slow reads, **Right Column:** Latency of read operations

10 Readers:

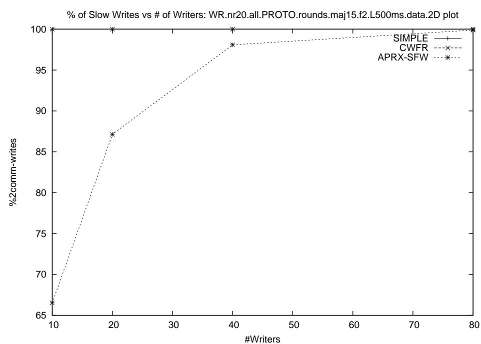


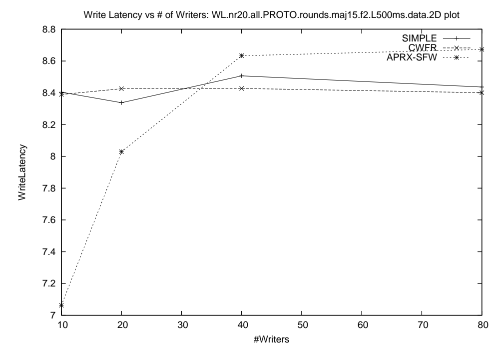


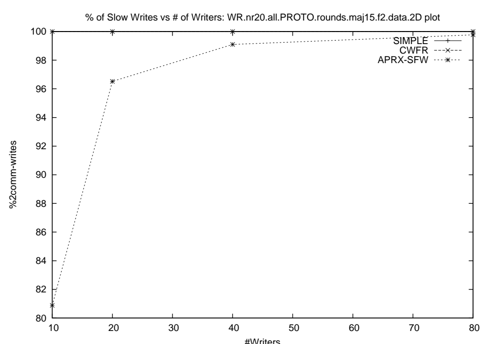




20 Readers:







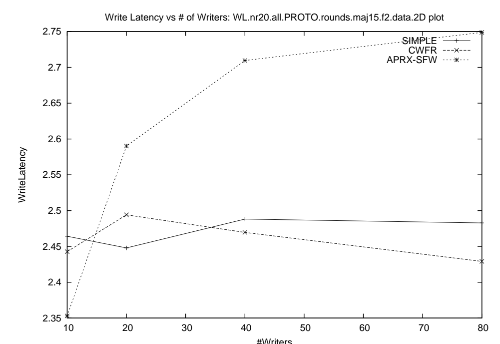
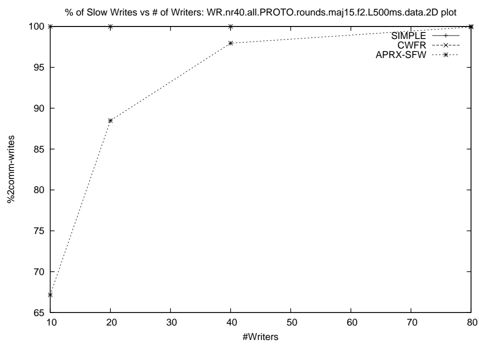
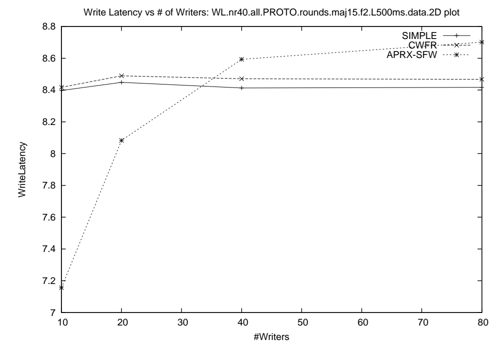
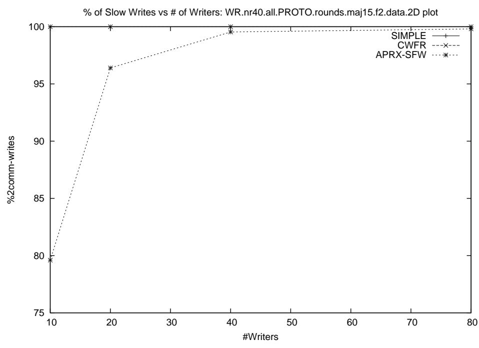


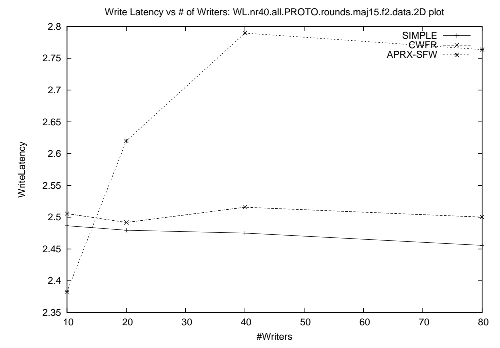
Figure 47: 6-wise quorum system ($|\mathcal{S} = 15,\ f = 2$): **Left Column:** Percentage of slow writes, **Right Column:** Latency of write operations

40 Readers:

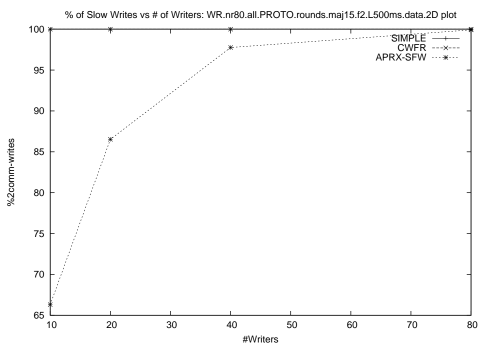


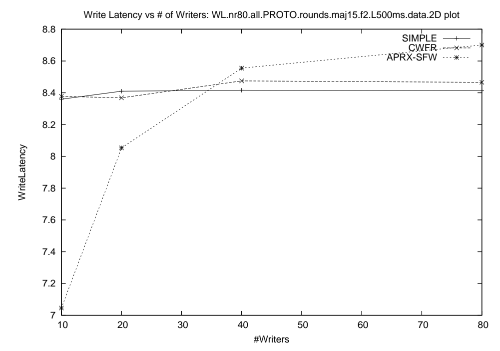


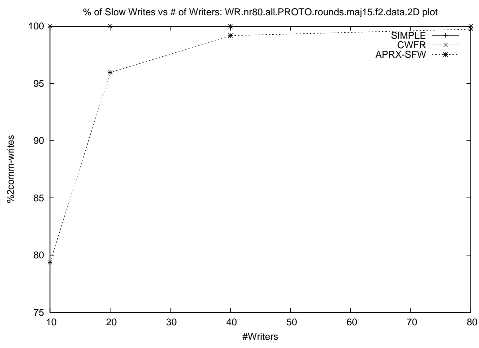




80 Readers:







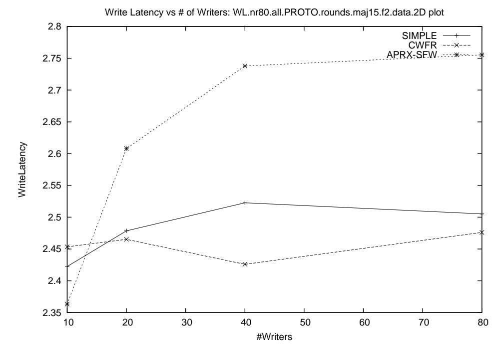


Figure 48: 6-wise quorum system ($|\mathcal{S}| = 15,\ f = 2$): **Left Column:** Percentage of slow writes, **Right Column:** Latency of write operations