Seminar Report: Opty

Ferran Delgà Fernández **Roberto Meroni** **Nicolás Zhilie Zhao**

December 19, 2023

1 Introduction

The Opty seminar is the last seminar of the Distributed Systems module and aims to enhance the students' understanding of optimistic concurrency control protocols through programming a small project in Erlang. In Section 2, we talk about the performance of the implemented protocol. In Section 3, we explain and demonstrate how we have materialised distributed execution. In Section 4, we implement forward validation and compare it with our previous backward validation version. And finally, we give our personal opinion about the seminar in the context of the Distributed Systems module.

In the remaining of this introduction section, we will illustrate the source code layout. Our source code layout consists of multiple directories:

- `backward`: contains the Erlang source code that implements backward validation of the protocol and a Python script to speedup the experiment process.
- `distributed`: contains the modified Erlang source code that implements the distributed version of the protocol.
- `forward`: contains the Erlang source code that implements backward validation of the protocol and a Python script to speedup the experiment process.

2 Performance

To demonstrate the protocol's performance across various parameter values, we conducted multiple experiments to investigate the fluctuation of mean accuracy based on factors such as the number of concurrent clients, writes per transaction, and entries. Additionally, we addressed the following questions, supporting our answers with line plots.

1. What is the impact of each of these parameters on the success rate (i.e. percentage of committed transactions with respect to the total)?

   Every parameters has some kind of negative or positive impact on the mean success rate. The details of the impact are specified in their corresponding subsection.

2. Is the success rate the same for the different clients?

   The success rate varies among clients, as depicted in the standard deviation plots in Figures 2, 4, 6, 8, 10, and 12. Although conventional practice involves adding error bars to the mean success rate line plots, we observed that, in some cases, the error bars were insignificantly small, making them ineffective for observation. Consequently, we opted for an additional plot to better illustrate the standard deviation.

2.1 Different number of concurrent clients in the system

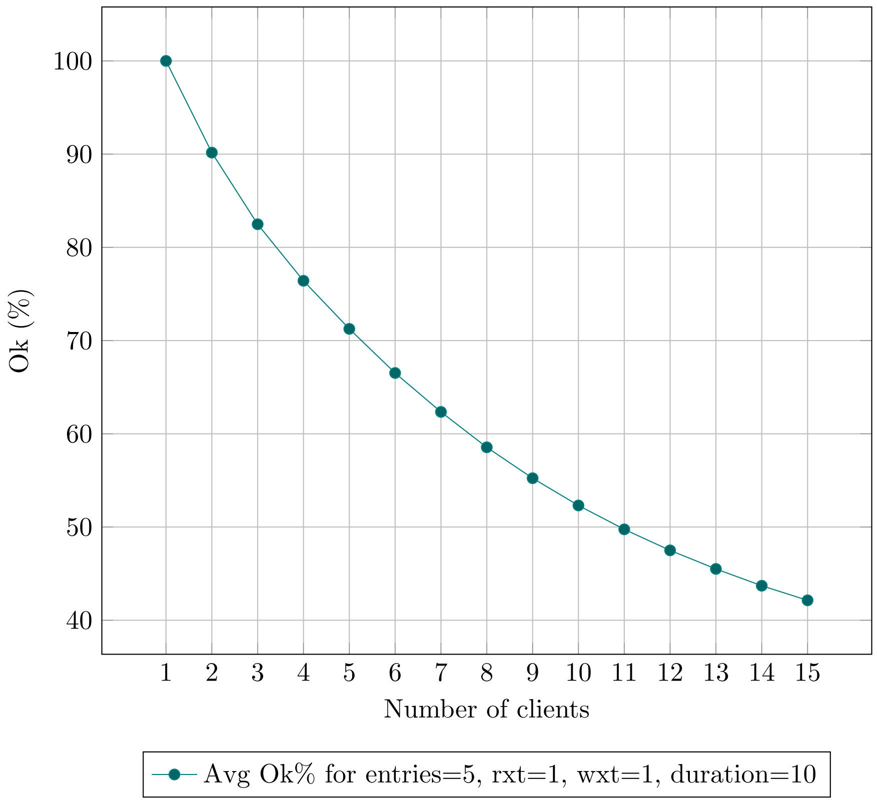


Figure 1: Number of clients vs success rate

As depicted in Figure 1, it is evident that the percentage of valid messages decreases with an increase in the number of concurrent clients. This outcome aligns with our expectations, as a higher number of clients corresponds to a increased probability of entries in the store appearing both in the write set of the committed transaction and the read set of the transaction awaiting validation. The elevation in the number of clients also incrases the concurrent transaction count, thus raising the likelihood of conflicts.

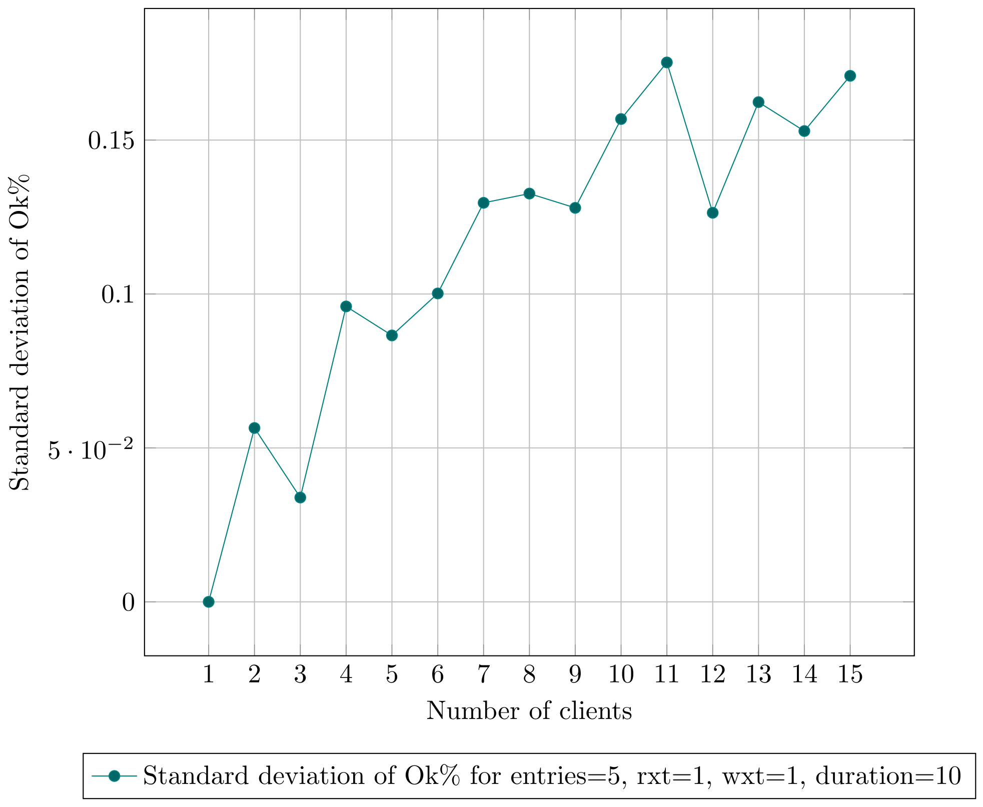


Figure 2: Number of clients vs standard deviation of the success rate

2.2 Different number of entries in the store

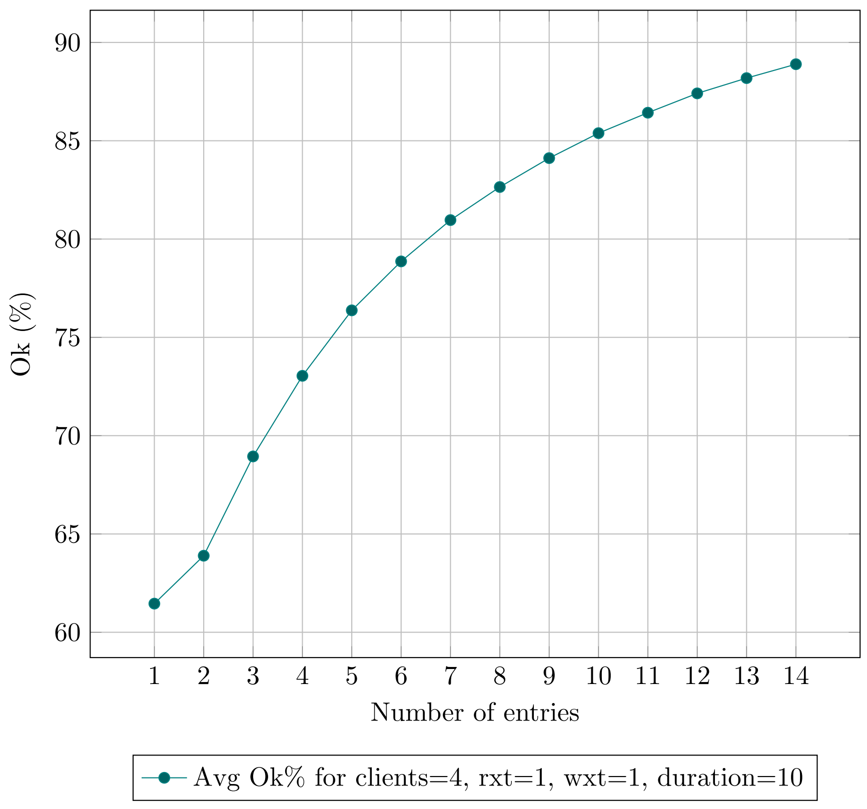


Figure 3: Number of entries vs success rate

As seen in Figure 3, when we increase the number of entries, the percentage of valid messages goes up. This happens because it lowers the chances of a write operation from a committed transaction and a later read operation from another active transaction trying to use the same entry. So generally speaking more entries mean a better success rate.

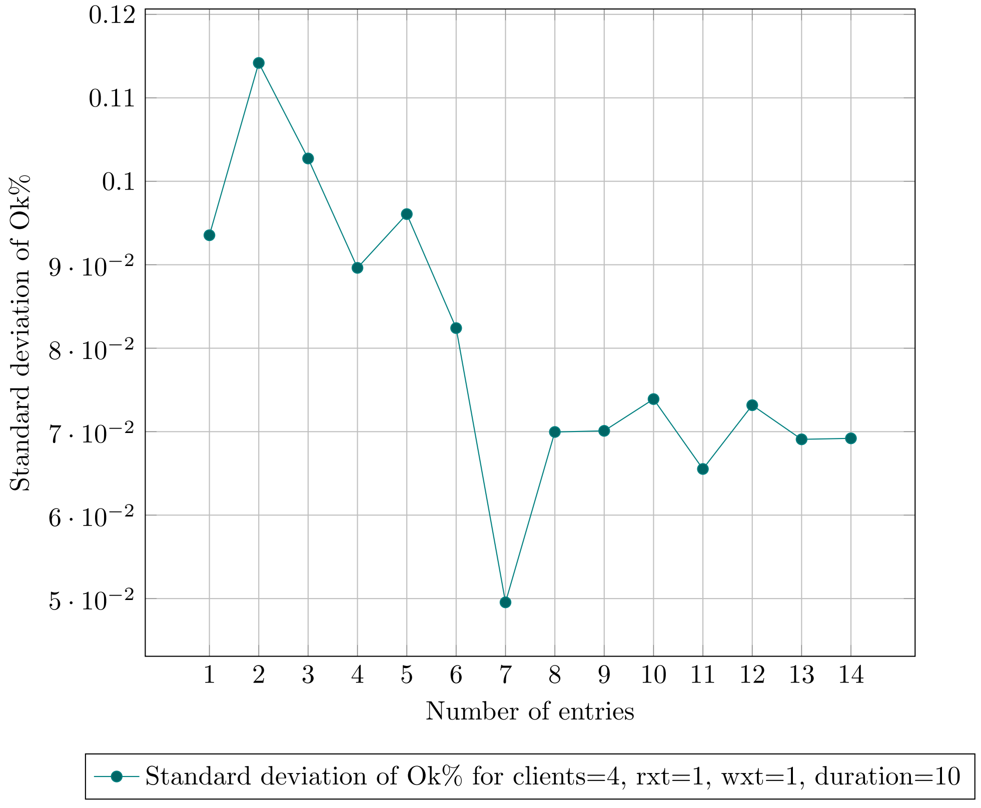


Figure 4: Number of entries vs standard deviation of the success rate

2.3 Different number of read operations per transaction

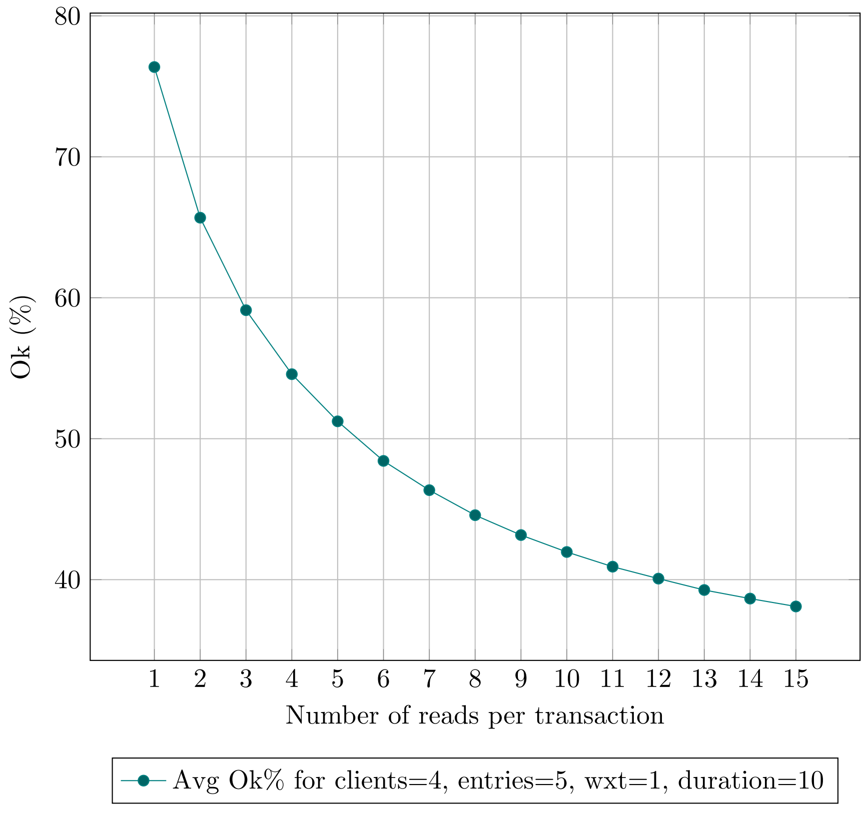


Figure 5: Number of reads per transaction vs success rate

Figure 5 suggests that as we increase the total number of operations per transaction, the likelihood of an operation affecting multiple entries in the store rises. Consequently, the probability of a subsequent write operation modifying those read entries also increases. Moreover, longer transactions result in more overlap between transactions, contributing to the observed trend.

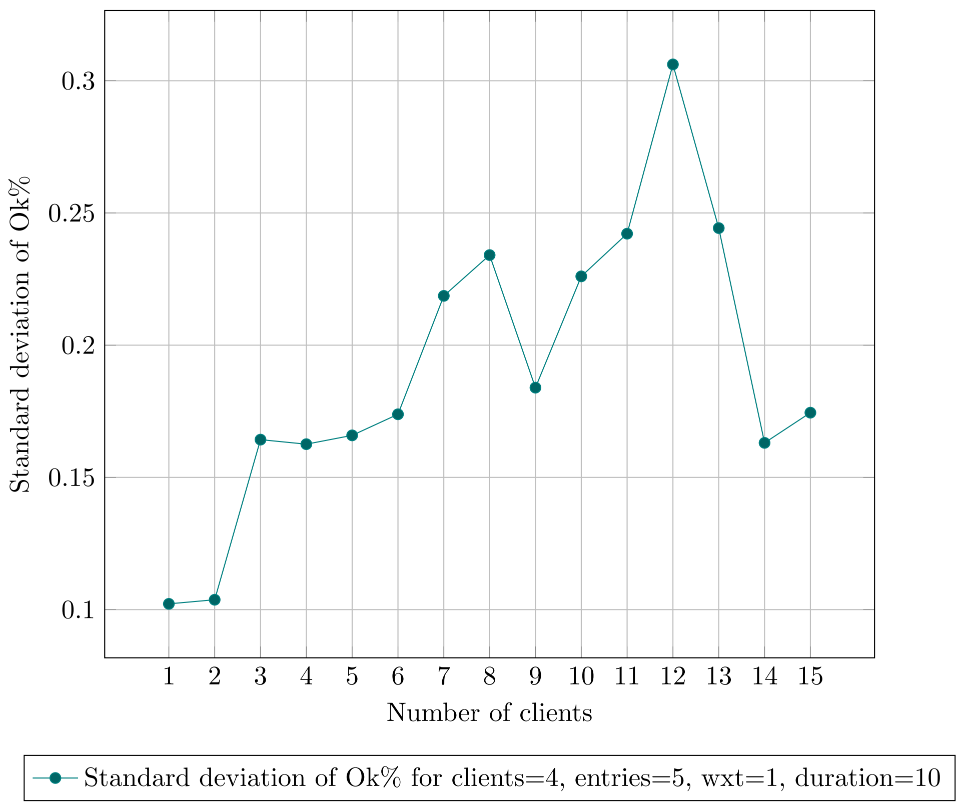


Figure 6: Number of reads per transaction vs standard deviation of the success rate

2.4 Different number of write operations per transaction

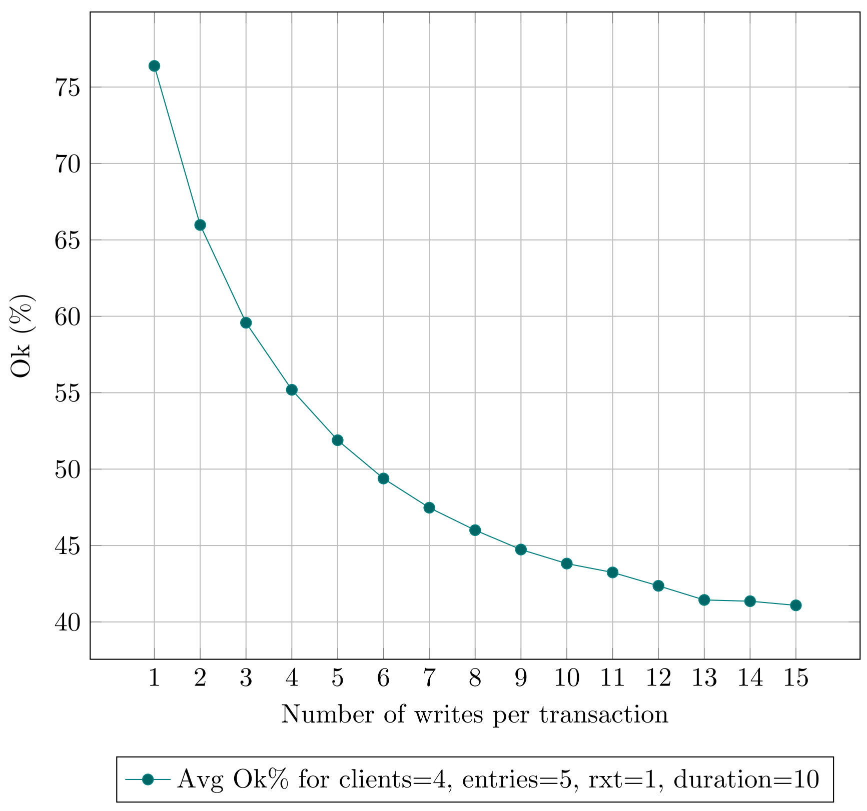


Figure 7: Number of writes per transaction vs success rate

Increasing the number of write operations per transactions and hence increasing the number of operations per transactions, decreases the mean success rate since more entries are more likely to be written and later read. This is shown in Figure 7.

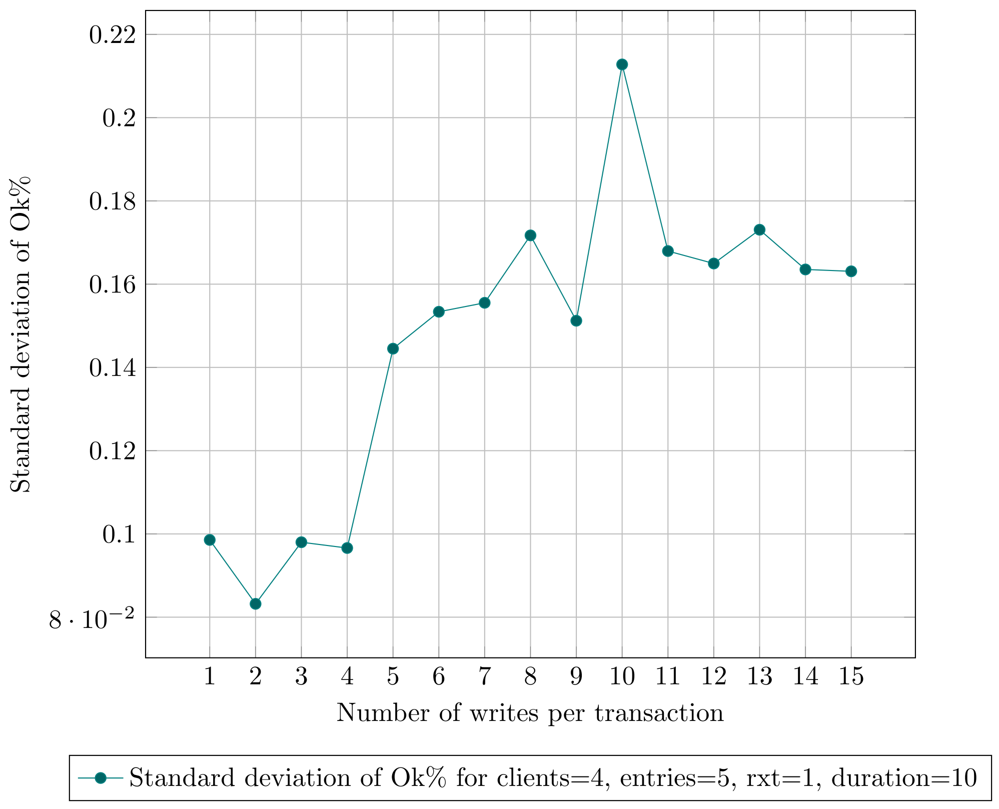


Figure 8: Number of writes per transaction vs standard deviation of success rate

2.5 Different ratio of read and write operations for a fixed amount of operations per transaction (including special cases having only read or write operations)

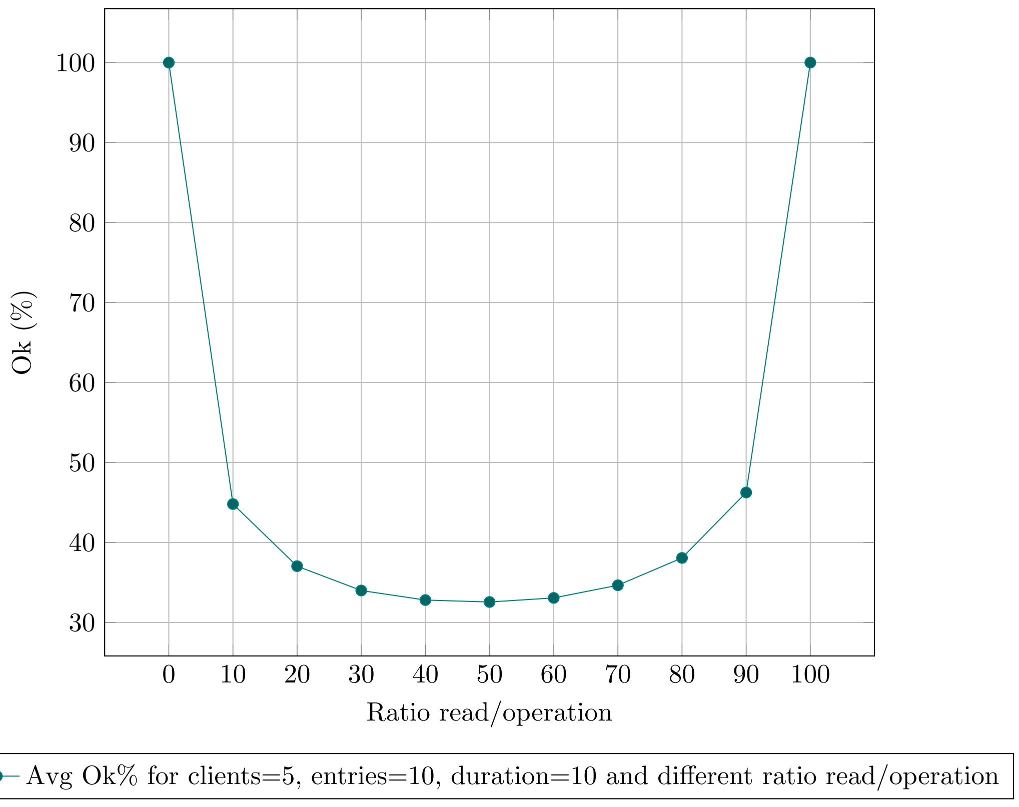


Figure 9: Ratio of read operations vs success rate.

Figure 9 suggests that having one-sided operation type results in a better percentage of successful transactions. As expected, conflicts appear whenever we have both read and write operations taking place concurrently, since multiple transactions may attempt to modify the same data. In our case, we obtained a symmetric behaviour while increasing the read operations over the fixed amount of total operations. Of course, having only write operations gives us 100% of successful transactions.

Note that in optimistic concurrency control, in the validation phase we check whether any data in the read-set has been modified by other transactions. In the case of write-only transactions, there is no read operation involved, and therefore, there's no need to check for conflicts related to the read-set.

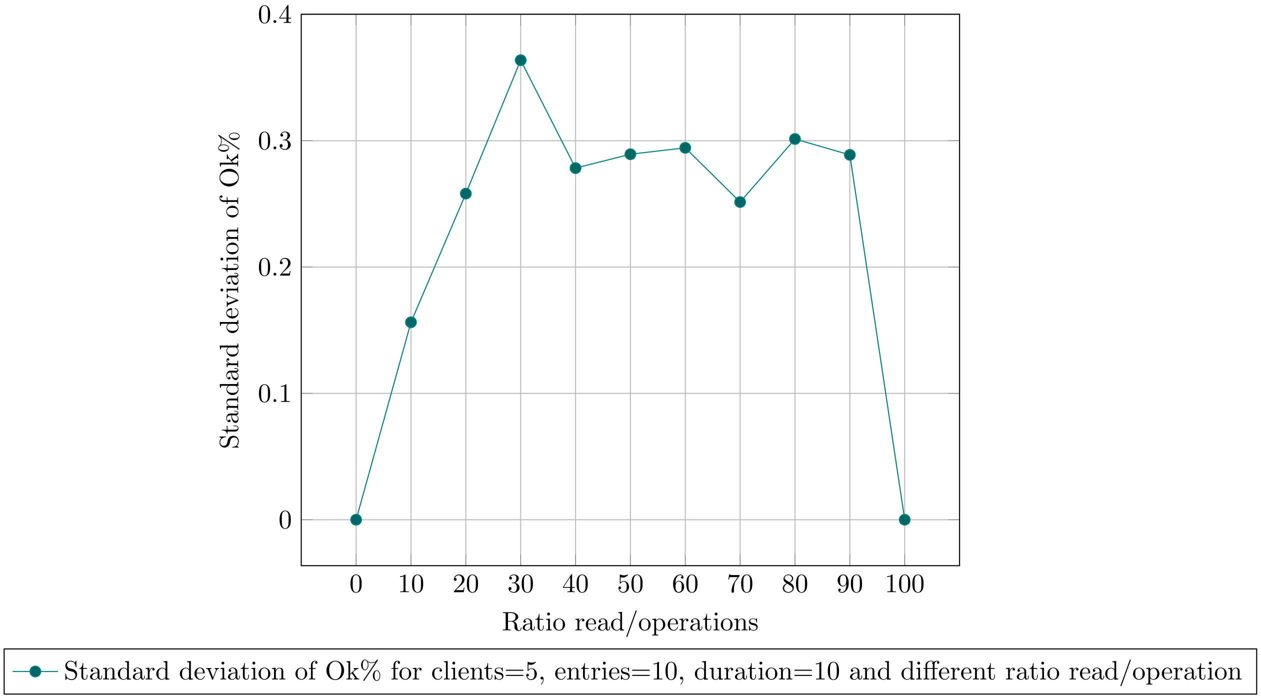


Figure 10: Ratio read/operations vs standard deviation of the success rate

2.6 Different percentage of accessed entries with respect to the total number of entries

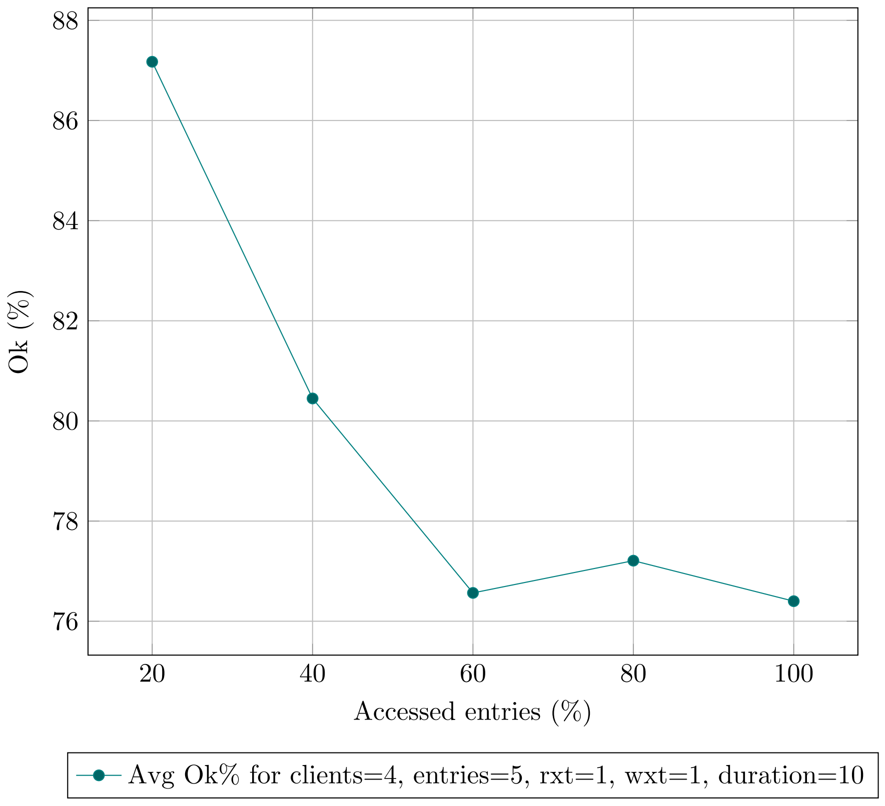


Figure 11: Percentage of accessed entries with respect to the total number of entries vs success rate

Figure 11 suggests that increasing the percentage of accessed entries with respect to the total number of entries decreases the mean success rate. Having a small subset of entries for each client makes it less likely to have multiple clients accessing the same entries. In other words, there should be less clients competing for the same entry of a store. However, if the same entry gets assigned to different clients with small subsets, the chances of conflicting transactions is higher (compared to large subsets).

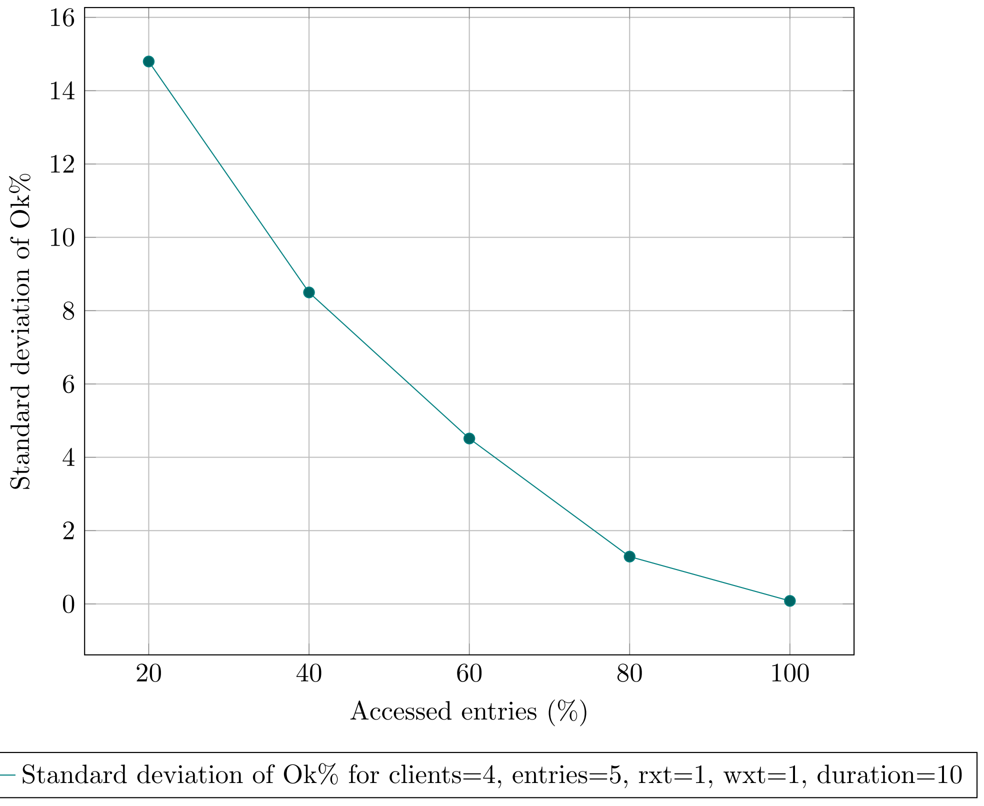


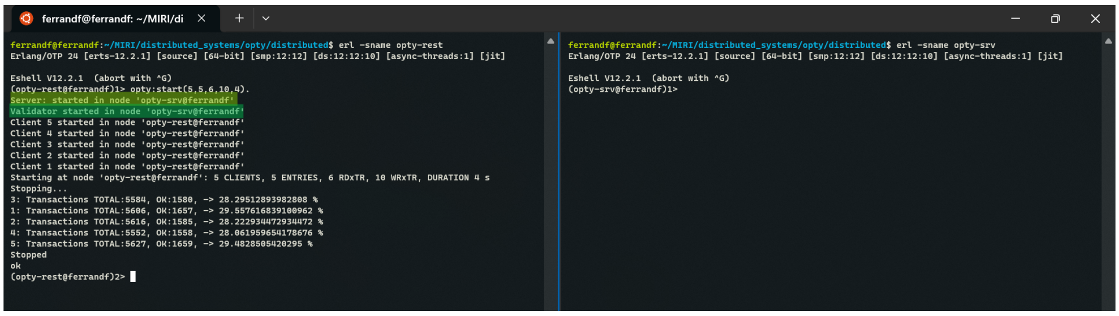
Figure 12: Percentage of accessed entries with respect to the total number of entries vs standard deviation of success rate

3 Distributed execution

We created a new erlang instance named *opty-srv@host*. In our opty file, we just have to add:

```
spawn(?ServerNode, fun() ->
    register(s, server:start(Entries)) end
),
```

to spawn the server in the new erlang instance. In the next picture we can see how are the components spawned and one *opty:start*.

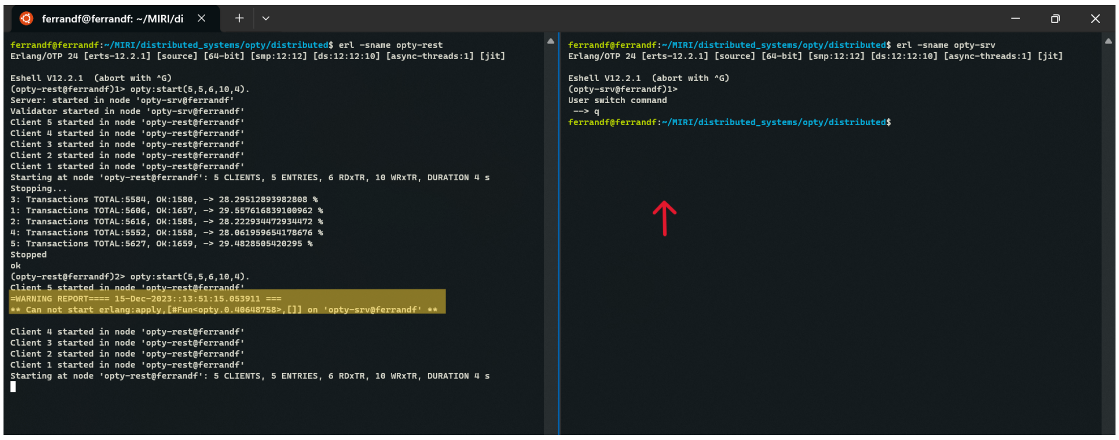


```
ferrandf@ferrandf:~/MIRI/distributed_systems/opty/distributed$ erl -sname opty-rest
Erlang/OTP 24 [erts-12.2.1] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [jit]

Eshell V12.2.1  (abort with ^G)
(opty-rest@ferrandf)1> opty:start(5,5,6,10,4).
Server: started in node 'opty-srv@ferrandf'
Validator started in node 'opty-srv@ferrandf'
Client 5 started in node 'opty-rest@ferrandf'
Client 4 started in node 'opty-rest@ferrandf'
Client 3 started in node 'opty-rest@ferrandf'
Client 2 started in node 'opty-rest@ferrandf'
Client 1 started in node 'opty-rest@ferrandf'
Starting at node 'opty-rest@ferrandf': 5 CLIENTS, 5 ENTRIES, 6 RDxTR, 10 WRxTR, DURATION 4 s
Stopping...
3: Transactions TOTAL:5584, OK:1580, -> 28.29512893982808 %
1: Transactions TOTAL:5606, OK:1657, -> 29.557616839100962 %
2: Transactions TOTAL:5616, OK:1585, -> 28.222934472934472 %
4: Transactions TOTAL:5552, OK:1558, -> 28.061959654178676 %
5: Transactions TOTAL:5627, OK:1659, -> 29.4828505420295 %
Stopped
ok
(opty-rest@ferrandf)2>
```

```
ferrandf@ferrandf:~/MIRI/distributed_systems/opty/distributed$ erl -sname opty-srv
Erlang/OTP 24 [erts-12.2.1] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [jit]

Eshell V12.2.1  (abort with ^G)
(opty-srv@ferrandf)1>
```

Figure 13: The green highlighted text shows that both the server and the validator are spawned in the opty-srv instance.

Another interesting thing is to see what happens if we abort the server instance. As expected, we get a warning message telling us that we are not able to contact it. The starting messages are also not displayed.



```
ferrandf@ferrandf:~/MIRI/distributed_systems/opty/distributed$ erl -sname opty-rest
Erlang/OTP 24 [erts-12.2.1] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [jit]

Eshell V12.2.1  (abort with ^G)
(opty-rest@ferrandf)1> opty:start(5,5,6,10,4).
Server: started in node 'opty-srv@ferrandf'
Validator started in node 'opty-srv@ferrandf'
Client 5 started in node 'opty-rest@ferrandf'
Client 4 started in node 'opty-rest@ferrandf'
Client 3 started in node 'opty-rest@ferrandf'
Client 2 started in node 'opty-rest@ferrandf'
Client 1 started in node 'opty-rest@ferrandf'
Starting at node 'opty-rest@ferrandf': 5 CLIENTS, 5 ENTRIES, 6 RDxTR, 10 WRxTR, DURATION 4 s
Stopping...
3: Transactions TOTAL:5584, OK:1580, -> 28.29512893982808 %
1: Transactions TOTAL:5606, OK:1657, -> 29.557616839100962 %
2: Transactions TOTAL:5616, OK:1585, -> 28.222934472934472 %
4: Transactions TOTAL:5552, OK:1558, -> 28.061959654178676 %
5: Transactions TOTAL:5627, OK:1659, -> 29.4828505420295 %
Stopped
ok
(opty-rest@ferrandf)2> opty:start(5,5,6,10,4).
Client 5 started in node 'opty-rest@ferrandf'
=WARNING REPORT==== 15-Dec-2023::13:51:15.053911 ===
** Can not start erlang:apply,[#Fun<opty.0.40648758>,[]] on 'opty-srv@ferrandf' **

Client 4 started in node 'opty-rest@ferrandf'
Client 3 started in node 'opty-rest@ferrandf'
Client 2 started in node 'opty-rest@ferrandf'
Client 1 started in node 'opty-rest@ferrandf'
Starting at node 'opty-rest@ferrandf': 5 CLIENTS, 5 ENTRIES, 6 RDxTR, 10 WRxTR, DURATION 4 s
```

```
ferrandf@ferrandf:~/MIRI/distributed_systems/opty/distributed$ erl -sname opty-srv
Erlang/OTP 24 [erts-12.2.1] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [jit]

Eshell V12.2.1  (abort with ^G)
(opty-srv@ferrandf)1>
User switch command
 --> q
ferrandf@ferrandf:~/MIRI/distributed_systems/opty/distributed$
```

Figure 14: The yellow highlighted text notes that the Erlang instance is down and for that it cannot start the server.

Note that here we didn't show where is the handler spawning. However, the one that starts the handler is the client and consequently, it is spawned in the same node as the client. That means that in our case, the handler is running in the Erlang instance called *opty-rest@host*.

4 Forward Validation

To implement forward validation, we have made several adjustments to our backward validation code, specifically in the *Handler*, *Entry*, and *Validator* modules. This section will delve into the details of the modifications made to each module to realize forward validation. The subsequent sections will cover the performance of forward validation compared to our previous backward validation implementation.

4.1 Handler

Two notable changes have been made to the Handler module. First, we now include the ID of the handler in the message sent to the validator. Second, the timestamp of each element in the read set is no longer necessary, and its removal is justified in the next section.

4.2 Entry

In the context of forward validation, where we compare the write set with the read set of active transactions, the timestamp in the entry is no longer needed. Additionally, we introduce a list of active transactions that have read the entry, called `ActiveReads`, which we will utilize for validation. To consistently check the read set of active transactions, a new parameter called *Owner* is defined in the Entry process. An entry is assigned an owner at the start of the validation phase, and once assigned, no process can read or write to the entry. The ownership is only released when the entry commits its write set[1]. Two new message types are introduced: {`block, NewOwner`} and {`unblock, Owner`}, which are used to block/unblock the entry during validation. Additionally, a {`clean, Handler`} message is introduced to delete the Handler ID from the active reads list.

4.3 Validator

The Validator ensures that a {`block, Handler`} message is sent to the entries in the write set of the transaction to be validated. This message blocks the entry and assigns ownership to the handler, allowing consistent validation and updating of entries without interference from other clients. It is noteworthy that the entry now receives the write set of the handler, as opposed to the read set. This change is logical since the validation process involves checking the intersection between the write set of the transaction to be validated and the read set of uncommitted transactions. After receiving validation results from all relevant entries, the handler's write set is committed into the entries.

To maintain the integrity of the `ActiveReads` list in entries, the validated handler is removed from the list by sending {`clean, Handler`} messages to entries present in the handler's read set. Finally, after completing the validation and updating process, entries that were locked are released by sending them {`unblock, Handler`} messages.

4.4 Performance

In Figure 15, 16, 17 and 18, we show the comparisons between backward and forward validation by showing how the success rate fluctuates depending on some parameters. As we can clearly observe from the previously mentioned plots, the forward validation implementation performs worse than our previous backward validation code. For backward validation, a transaction X gets invalidated if an entry in its read set is in the write set of another concurrent transaction Y, and if Y gets validated. For forward validation, a transaction X gets invalidated if an entry in its write set is in the read set of another concurrent transaction Y, even if Y doesn't get validated. Given that we suppose equal expected transaction times, probability of an entry being part of a set equally distributed, probability of a transaction getting validated ≤ 1, the forward validation implementation is expected to invalidate more transactions.

[1]Notice that even though we use the *ownership* terminology, it is basically the same fundamental idea as the *mutex* that we saw in lectures.

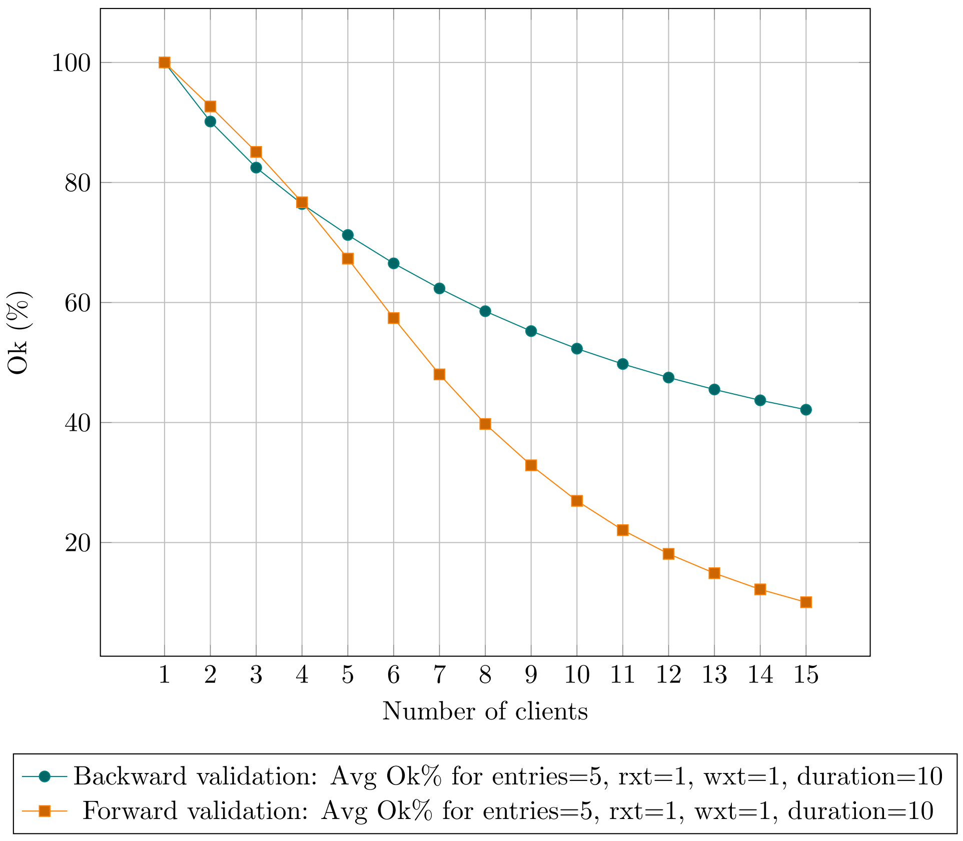


Figure 15: Number of clients vs success rate

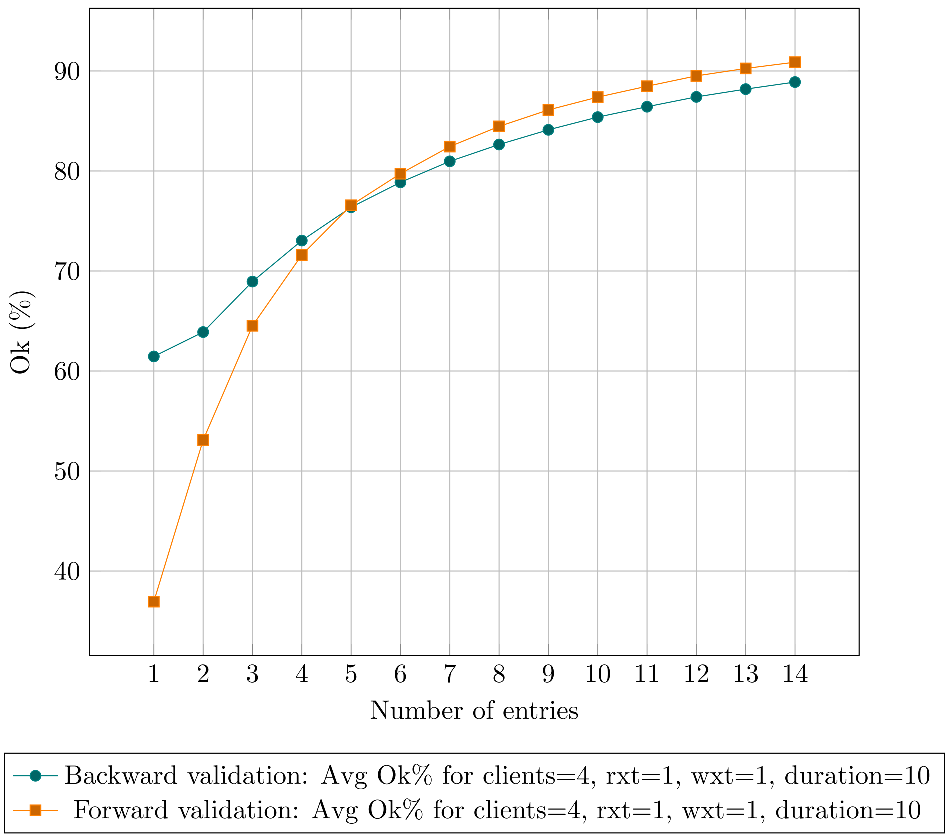


Figure 16: Number of entries vs success rate

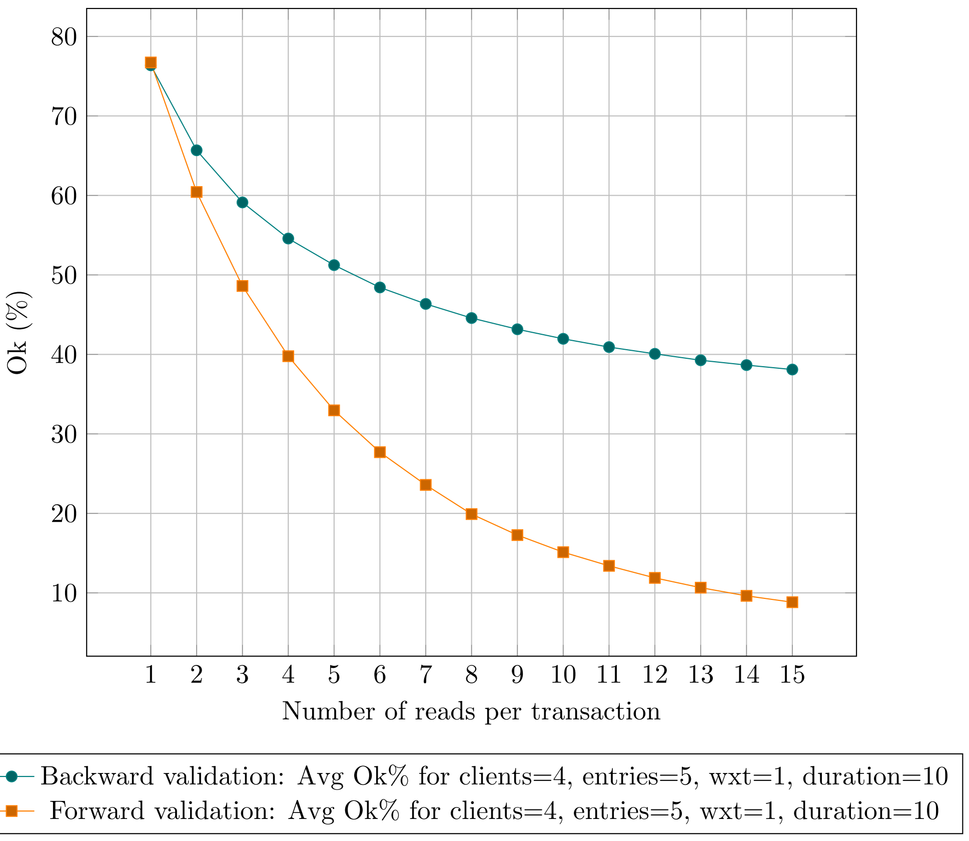


Figure 17: Number of reads per transaction vs success rate

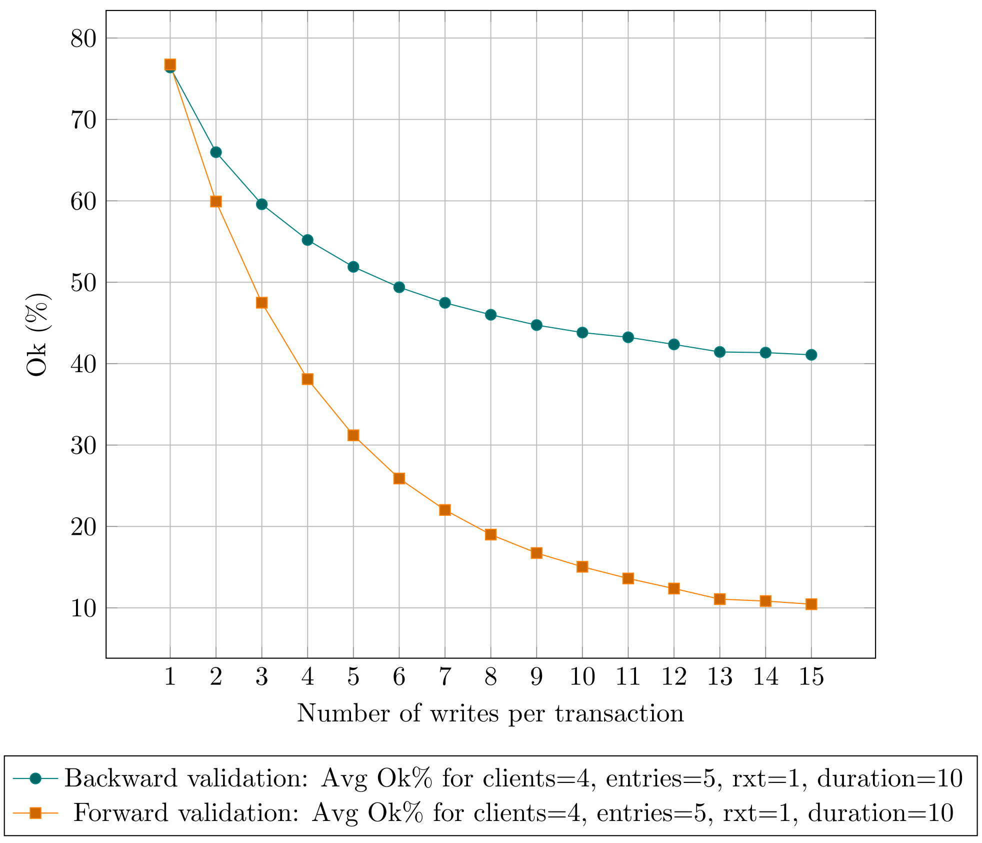


Figure 18: Number of writes per transaction vs success rate

5 Personal opinion

5.1 Ferran Delgà Fernández

I found the seminar helpful to better understand the implementation of optimistic concurrency control. I believe is accurate to do the implementations taking into account what has been taugh in lectures. It should definetly be included in the next year's course. On my end, I realize that there's room for me to delve deeper into the finer points of the implementation.

5.2 Roberto Meroni

Overall, the seminar looks fine as it is, I don't think there is much room for improvement considering the time limitations. I'd have appreciated more effective resources to face this seminar. For example, the paper for Paxos was perfect, containing exactly everything that was needed and nothing more.

5.3 Nicolás Zhilie Zhao

This seminar was useful to clarify the strength and weaknesses of the optimistic concurrency control. However, maybe having a previous seminar or assignment implementing the pessimistic concurrency control would further benefit the student to understand the pros and cons of the optimistic approach.