Runtime Consistency Adaptation

Vasco António de Aguiar e Morganho

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. João Coelho Garcia

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. João Coelho Garcia
Member of the Committee: Prof. João Carlos Antunes Leitão

October 2021
Acknowledgements

In this section I would like to write a short paragraph expressing my gratitude to the people that made it possible for me to complete this thesis. First and foremost I would like to thank my parents that provided me with everything necessary to ensure I was never lacking the resources to make this work possible, as well as the motivation they continuously gave me throughout the last year. I would like to thank my supervisor Professor João Garcia and João Loff, who were very reliable when it came to ensuring the quality of this work. I really appreciate their effort, availability and enthusiasm, as I could always seek them for guidance. Lastly I would like to thank all my friends and other family members that somehow contributed to keep me motivated and work hard in this thesis.
Abstract

Distributed systems are an essential component of Internet services, as they give us the ability to access data in different machines in different locations. This is possible through replication, but without the implementation of a replication protocol in the underlying system, this replication can be unsound and cause dissimilarity between replicas due to delays in the network and failures. Most systems use one single consistency model to solve these problems, which is embedded in their own implementation. If the consistency model needs to be changed or updated, the system has to be deeply rewritten or replaced by a different one. Variable Consistency Messaging Layer [4] solves this problem by abstracting the implementation of the consistency model into a set of modules. This abstraction eases the switching of replication protocols but the developer still has to restart the system in order to change consistency models.

We propose a framework that makes the dynamic change of consistency models possible at runtime according to rules defined by the programmer, through the use of VCML. Our new version of the framework takes metrics like latency, throughput, etc, and computes them in a pre-set of rules given by the developer, in order to dynamically change the models at runtime. Our experiments show that our solution is able to increase the performance of a system by up to 13.6% in a scenario where the state of the system gradually transfers from mostly read state to a write heavy state.

Keywords: Consistency, Replication, Distributed Systems, Dynamic Adaptation
Resumo

Os sistemas distribuídos são um componente essencial dos serviços de Internet, pois permitem-nos aceder a dados em diferentes máquinas em diferentes locais. Isto é possível devido a estratégias de replicação, mas sem a implementação de um protocolo de consistência no sistema subjacente, esta replicação pode ser incorreta e causar dissimilaridade entre as réplicas devido a atrasos na rede e falhas. A maioria dos sistemas usa um único modelo de consistência para resolver esses problemas, que está embutido na sua própria implementação. Se o modelo de consistência precisa de ser alterado ou atualizado, o sistema tem de ser profundamente reescrito ou substituído por um diferente. Variable Consistency Messaging Layer [4] resolve esse problema abstraindo a implementação do modelo de consistência num conjunto de módulos. Essa abstração facilita a troca de protocolos de consistência, mas o programador ainda precisa reiniciar o sistema para alterar os modelos de consistência.

Propomos um framework que possibilita a mudança dinâmica de modelos de consistência em tempo de execução de acordo com regras definidas pelo programador, através do uso de VCML. Esta nova versão da framework usa métricas como latência, produtividade, etc, e calcula-as num pré-conjunto de regras fornecidas pelo programador com o objetivo de alterar dinamicamente os modelos em tempo de execução. As experiências de desempenho realizadas, mostram que a nossa solução é capaz de aumentar o desempenho de um sistema em até 13,6% num cenário onde o estado do sistema é alterado gradualmente, de um estado maioritariamente de leitura para um estado maioritariamente de escrita.

**Palavras-chave:** Consistência, Replicação, Sistemas Distribuídos, Adaptação Dinâmica
# Contents

1 Introduction 19

2 Related Work 22

2.1 Consistency 22

2.2 Consistency-Performance tradeoff 25

2.2.1 Consistency vs Performance in different consistency levels 25

   I. Strong consistency models 25
   II. Available consistency models 26

2.2.2 Consistency model performance metrics 26

2.3 Variable Consistency Messaging Layer 27

2.3.1 Framework API 28

2.3.2 Group Membership 30

2.3.3 Ordering 30

2.3.4 Replication 31

2.3.5 Delivery Condition 31

2.3.6 Quorum 32

2.3.7 Communication 32

2.4 Motivation for Dynamic Adaptation 32

   2.4.1 Handling transient state in dynamic adaptation - Consistent cuts 34

3 Architecture 36

3.1 Tracker role 38

3.2 Tracking metrics 38

   I. Tracker node: 38
   II. System node: 39

   3.2.2 Latency and throughput monitoring 40

   3.2.3 Stale read ratio monitoring 40

3.3 Evaluation module and Configurations 41

   3.3.1 Configurations 42

3.4 Reconfiguring the framework 42
I. Tracker node

II. System node

4 Evaluation

4.1 Methodology

4.2 Experimental Setup

4.3 Experimental Results

4.3.1 Overhead caused by the monitoring of metrics

4.3.2 Potential gain by reconfiguring the framework

4.4 Discussion

5 Conclusion

5.1 Future Work

References
List of Figures

Fig. 1. Example of an inconsistent system 24
Fig. 2. Consistency models tree 24
Fig. 3. Stale read situation 28
Fig. 4. New message flow 30
Fig. 5. Response time 34
Fig. 6. Cost per transaction 34
Fig. 7. Consistent vs inconsistent cut 35
Fig. 8. Overview of dynamicVCML 38
Fig. 9. Metrics tracking and evaluation 40
Fig. 10. System reconfiguration 44
Fig. 11. staticVCML vs dynamicVCML throughput with a gradual increase of the write ratio 50
List of Tables

Table 4.1. staticVCML vs dynamicVCML
(with no configuration change) overhead in COPS - 95:05 operations ratio 48

Table 4.2. staticVCML vs dynamicVCML
(with no configuration change) overhead in COPS - 50:50 operations ratio 48

Table 4.3. staticVCML vs dynamicVCML
(with a reconfiguration mid experiment) gains in COPS - 95:05 operations ratio 49

Table 4.4. staticVCML vs dynamicVCML
(with a reconfiguration mid experiment) gains in COPS - 50:50 operations ratio 49
List of Listings

Listing 3.2.1. Stale reads algorithm 42
Listing 3.3.1. Example of a set of rules in the Evaluation module 42
Listing 3.3.2. Example of two configuration text files 43
### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCML</td>
<td>Variable Consistency Messaging Layer</td>
</tr>
<tr>
<td>CAP</td>
<td>Consistency, Availability, Partition-Tolerance</td>
</tr>
<tr>
<td>COPS</td>
<td>Clusters of Order-Preserving Servers</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Process Unit</td>
</tr>
<tr>
<td>DKVF</td>
<td>Distributed Key-Value Store</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>YCSB</td>
<td>Yahoo! Cloud Serving Benchmark</td>
</tr>
<tr>
<td>BASE</td>
<td>Basically Available, Soft state, Eventually consistent</td>
</tr>
<tr>
<td>ACID</td>
<td>Atomicity, Consistency, Isolation, and Durability</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
</tbody>
</table>
1 Introduction

Distributed systems are collections of computing elements (or nodes) each being able to behave independently of each other. To enable the availability of consistent replicas of data across multiple system nodes at all times, a replication model has to be implemented in the system. Data replication is used to enhance the availability, reliability and throughput of data between replicas (or nodes). But data replication across nodes brings some challenges, such as: availability of the entire system, dynamic group membership, consistency and coordination between nodes spread throughout the system. To solve these problems the system developers implement a consistency model in the system when developing it. However, there are multiple consistency models (Sec. 2.1) and some are better than others depending on the circumstances. For example, for a bank’s transaction service it is important for the system to be strongly consistent, as seen in section 2.1, Fig. 2, whereas in a social media service having a lower level of consistency is not as impactful.

An optimal system would be consistent, have high availability and have network partition-tolerance, but unfortunately the CAP [1] theorem states that having these three properties simultaneously is impossible. Ensuring a high level of consistency at all times in terms of synchronous replication means a loss of performance due to high operation latencies. This becomes even more apparent when data is being replicated over geographically distant areas, since the distance factor affects latency. Consequently, many services and cloud providers use storage systems with eventual consistency, which ensures that all replicas will eventually become consistent even though the system can return stale data at some points in time. Yet, the use of this more available form of consistency comes with its own drawbacks. As shown in [2], under heavy client access by means of reads and writes, some available consistency systems may return up to 66.61% stale reads, meaning that probably two out of three reads are inconsistent. This can also lead to increased conflict handling, which can result in a higher overhead. When under heavy access, available consistency solutions can prove to be too costly in terms of inconsistency. On the other hand, systems that use strong consistency models may become too unavailable during heavy access periods, leading to sub optimized user experience.

When implementing a new system the developers have to be very careful about which consistency model to implement when building a distributed system, since the consistency model is normally deeply embedded in the system making it almost unchangeable after the system is completed. Depending on the current state of a system in terms of the amount of reads and writes operation being made at a point in time, network latency, etc. different levels of consistency can prove to be more efficient than others. Having the right level of consistency for the current state of the system can solve issues like loss of performance during high access times, or low consistency when the amount of stale reads becomes too high. If for some reason the consistency model does not meet the needed requirements and has to be replaced, the system will have to be deeply rewritten, or even replaced in its totality. Being able to abstract the
implementation of the consistency model from the implementation of the system would solve this issue. Luckily there is a framework that does this abstraction. Variable Consistency Messaging Layer [3] uses a set of well-defined modules, aiming to frame the most common replication protocols within these modules, as well as, to ease the switching of replication protocols in the target system. This solves the problem of having to rewrite the system in order to change the consistency model, by simply reconfiguring the framework.

Sometimes an event can occur in the server or client side of a system, like a sudden increase of requests in a given interval, that affects the efficiency and effectiveness of said system, and if the efficiency and effectiveness of the system are badly compromised, this can call for a sudden need to reconfigure the system. However, with Variable Consistency Messaging Layer, this reconfiguration can only be done during build time, meaning that if we wish to reconfigure the framework, the service will have to stop in order to apply the changes made. This leads to a loss of availability, since the whole system needs to stop while the new changes are being applied. Having the whole system unavailable, even if only for a couple of minutes, is critically nonoptimal, especially during high access times, where a lot of work would have to be delayed or even lost. For example, let's say an online store sets up an exclusive sale event during a short period. If in that short period the system becomes overloaded with requests, resulting in bad performance, i.e. bad user experience, the event would be compromised. Changing the configuration of the system, in order to amend this issue, most of the time means that the system will have to become unavailable while these changes occur. Having the ability to swiftly change the configuration of the system during runtime could be a solution to prevent issues of this nature.

A system that dynamically changes consistency models during runtime, adapting to the state of the system, in order to ensure good performance as well as good consistency at all times would be ideal, but also challenging. How do we monitor the performance of the system? What are the challenges of changing consistency models during runtime?

In this work we will address the tradeoff between performance and consistency, as well as which metrics should be taken into consideration in order to calculate this tradeoff for a system during runtime. We will add the functionality of dynamic adaptation of the consistency protocol in runtime to Variable Consistency Messaging Layer. We will show how we monitor the system's performance and how we made the reconfiguration of consistency models possible during runtime, as well as an easy way for the programmer to create his own configurations and set of rules for when reconfigurations should happen.
2 Related Work

2.1 Consistency

In this section we will look at why consistency protocols are important in distributed systems, as well as explore some of the more known protocols and how they work. Consistency models are essentially a contract between processes, clients or users and the data store and the replicas that store the data [4]. They determine the rules for visibility and apparent order of updates to the system’s objects. If you successfully write a value to a key in a strongly consistent system, the next successful read of that key is guaranteed to show that write. A client will never see out-of-date values [5]. But to have strong consistency, the scalability and performance of the system must be compromised. Network delays, acquisition of locks, waiting for replication are problems that have to be dealt with by the system resulting in poor system performance, specifically high latency and low throughput. That is the reason why most non critical applications tend to implement more available forms of consistency in their systems [6], [7], [8], [9], in order to provide high throughput and low latency. But it is evident that available consistency models have their own drawbacks. One of these drawbacks can be the exposure of anomalies to the user [6]–[8], for example: a user can have their permissions revoked in a file sharing application, but still be able to read files that were uploaded after the revocation [8].

Strong consistency is critical in some systems, such as distributed banking systems, for example, where any data anomaly is unacceptable. Let us analyze an inconsistent distributed banking system (Fig. 1.): We have a shared bank account, owned by user X and user Y, with 100 credits. The system consists of replicas A and B. User X is connected to replica A and user Y is connected to replica B. User X withdraws 80 credits sending a message to replica A, and user Y also performs a withdrawal operation of 100 credits in replica B just after X makes its withdrawal. Both users are able to perform the withdrawal operations, getting more money than they had in their shared account, because the system allows inconsistent states.

An optimal system would be consistent, have high availability and have network partition-tolerance, but unfortunately the CAP [1] theorem states that having these three properties simultaneously is impossible. CAP is an acronym for Consistency, Availability and Partition tolerance, and the theorem states that a system can only strongly support two of these properties simultaneously. For a system to be strongly consistent, a read operation always returns the same value independently of the node where it is being performed at a given point in time. This value is also the value of the most recent write operation. If every request gets a response regardless of the state of any individual node in the system, the system is said to have availability. If the system continues to run without the loss of data or messages despite the number of messages being delayed by the network between nodes or even in the event of nodes failing, the system is said to have partition-tolerance. Availability is achieved by replicating the data across different nodes, and by having the system always return a value independently of the read operation, but this value can be outdated due to the network partitioning as well as other circumstances. Therefore, by achieving availability, consistency is sacrificed. When consistency is guaranteed, the system will return an error or a time-out if particular information cannot
be guaranteed to be up-to-date, therefore sacrificing availability.

![Diagram of an inconsistent system]

**Fig. 1.** Example of a inconsistent system

There are several consistency models with different levels of consistency guarantees that we will explore in this section. In this proposal we will not be studying transactional models and session guarantees, as they are not compatible with Variable Consistency Messaging Layer. In Fig. 2 we can see the relationships between common consistency models for concurrent systems, arrows show the relationship between consistency models. For instance, strict serializable implies both serializability and linearizability, linearizability implies sequential consistency, and so on.

![Consistency models tree diagram]

**Fig. 2.** Consistency models tree, adapted from [10]
• **Causal Consistency**  
This consistency model ensures that causally-related operations should appear in the same order on all processes. If an operation A requires an operation B to be executed to be correct, then A is causally-related to B. If operations are not causally-related, they can be immediately applied without being ordered by causal consistency [9], [11]. It guarantees that if an operation depends on a subsequent operation to be correct, the system will only return the result of the second operation when the first operation’s result is available.

We can further analyze these dependencies with Lamport’s [12] notion of potential causality: happens-before (→).

1. The causal order between two operations A and B is A→B, if A was executed before B in the same process.
2. If an operation A is a write and operation B is a read, then A→B if B returns the value written by A even if the operations are executed at different processes.
3. If two operations A and B have the causal order A→B and an operation C has the causal order with B of B→C, then A→C.

• **Sequential Consistency**  
Total order is guaranteed in this model. The result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program [13]. It is possible for one replica to be ahead or behind another replica, but when it returns an object, it is not possible for that replica to return a previous version of that object that it had already returned.

• **Linearizability**  
Also known as atomic consistency [6], [14]. This consistency model guarantees total order but, differing from Sequential Consistency, with real-time ordering operations. It provides a real-time guarantee on the behavior of a set of single operations on a single object. A read operation will always return the value of the latest write. Nonetheless, it supports concurrent operations. Given an operation A and an operation B, in which A is a write and B is a read and these operations are concurrent, there is no guarantee of what value B will return. It can return the value written by A or the outdated value. This is because what this model guarantees is if an operation is complete, a following operation returns the latest value.

Hybrid models, like RedBlue Consistency for example, are becoming increasingly more popular. These models use more than one consistency model. This increases efficiency and effectiveness of systems that have operations that require stronger consistency and others that are required to be executed fast. In the case of RedBlue [15], it uses eventual consistency for one type of operation (Red) and linearizability for other types of operations (Blue).
2.2 Consistency-Performance tradeoff

As previously mentioned, the CAP theorem states that only two out of the three following properties can be guaranteed simultaneously: Consistency, Availability and Partition tolerance. Partition tolerance is crucial for networking reliant scalable distributed systems, so the real tradeoff that has to be defined is between consistency and availability. In [16], Eric Brewer argued that the consistency-performance tradeoff is more important than the consistency-availability tradeoff. Brewer argues that since partitions are rare, the obvious solution is to predict their occurrences and the consistency-availability tradeoff should only be considered during the partitions. Contrastingly, the consistency-performance tradeoff is a permanent one.

With a lot of systems being deployed on a wide area scale with data replicated over distant geographical areas, traditional storage systems that ensure strong consistency will experience higher latencies. For this reason eventual consistency [17] was introduced as an alternative. Many storage systems nowadays opt for BASE properties (basically available, soft state, eventually consistent) instead of ACID properties (Atomicity, Consistency, Isolation, and Durability) in order to increase performance and availability by relaxing the consistency rules.

2.2.1 Consistency vs Performance in different consistency levels

As addressed in the previous section, there is a tradeoff between consistency and performance, meaning that lower or more available levels of consistency will increase performance, and higher or stronger levels of consistency will decrease performance. This is because of how differently the replication system works depending on the level of consistency that is being used.

I. Strong consistency models

Implementing strong consistency can be very expensive, for example, the duration of reads or writes in sequentially consistent systems has to be linear with the latency of the network [18]. This results in a loss of performance, as a consequence of this loss of performance, availability is reduced as well [19]. While high availability and high throughput have great value, there are also many cases where strong data consistency is more important than availability and high throughput [5].

In other words, there are cases where performance has to be sacrificed in order to guarantee the level of consistency required by the system. Consistency models like Sequential Consistency and Linearizability (Sec. 2.1) are well composed but restrictive in performance since they severely restrict possible optimizations such as pipelining write accesses [20]. Fortunately, strong consistency is not needed in most cases.
II. Available consistency models

Many available consistency models have been developed to allow the host to relax consistency while still guaranteeing that the consistency required by the system is fulfilled. Google's BigTable [21], for example, uses available consistency models like eventual consistency since it accomplishes the consistency requirements and isn’t as expensive as stronger consistency models are. Because available consistency models are not strict in guaranteeing program order, these models allow for performance optimizations. At the cost of guaranteeing consistent data, available consistency improves performance [22].

But when the system is under heavy access, i.e. under heavy reads and writes the system can become too costly in terms of inconsistency. With higher inconsistency can also come increased conflict handling, and if this conflict handling becomes frequent it can result in higher overhead, leading to lower user experience. One example of this is Facebook, which uses a cloud system called Cassandra [23]. Cassandra uses eventual consistency, providing high availability, scalability and performance. A study made in 2011 showed that with 890 million people using Facebook daily [24], i.e. being under heavy access at all times, its stale read rate was higher than 60% [2]. This is not ideal for such a big company, therefore methods like Static Consistency were implemented, where different groups of data are treated differently in terms of consistency level [2]. Even in a social network-type setting, where more available consistency models are preferred, this amount of inconsistency is hardly desirable. We can conclude that even available consistency models can reach a saturation point. Some available consistency models, mainly eventual consistency, can become very inconsistent under heavy access.

2.2.2 Consistency model performance metrics

The goal of our proposal is to make Variable Consistency Messaging Layer handle consistency dynamically at runtime, in order to offer the best tradeoff between consistency and performance. To achieve this, metrics have to be tracked and taken into consideration when calculating this tradeoff.

I. Performance

A way to evaluate performance is to take these two metrics into consideration: read and write operations latency and throughput. These are the two most common metrics used to evaluate performance in cloud systems [2], [38]. These operations latency will generally be higher when using strong consistency since, for example, in case of quorum-based models, the reads have to wait for the replies of other replicas.

Throughput is directly affected by the read operation time. Slower reads make the number of possible operations per second smaller.
II. **Consistency**

Analyzing the stale reads rate of reads during execution is a way to evaluate consistency during execution. As seen in (Sec. 2.2.1.II), systems that use available consistency get increasingly inconsistent the higher the access rate they are under at a given time. For a social media application, for example, high stale read rate can be tolerable, but if an online shop application has a high stale read rate during heavy access, anomalies can occur. For critical applications that require strong consistency, this rate should be 0%. In Figure 3 we can see what are the conditions that lead to a stale read. Each write done within the Tp timeframe could be a stale read, if it is done in a replica where the original write has not been yet propagated to.

![Stale read situation](image)

*Fig. 3. Stale read situation*

### 2.3 Variable Consistency Messaging Layer

Variable Consistency Messaging Layer abstracts the implementation of a system's consistency underlying model, making consistency adaptations an easier task. It extracts the consistency implementation to a modular layer under the systems. These modules can be independently changed in order to allow switching consistency models when needed.
Not every consistency model is adequate for all system execution conditions. The VCML framework explored how different consistency models could be easily exchanged in a modular fashion. The aim of this framework is to ease the switching of consistency protocols in systems, alleviating the programmers job of having to rewrite the system when the current consistency protocol is no longer adequate. This can happen when, for example, the consistency guarantees of the protocol that was initially implemented in the core of the system are no longer appropriate for the business requirements of a company. This can happen, for example, when there is a sudden increase of popularity of an application, and the current consistency model proves to be too unavailable for the increased demand in requests, making the desire to change the applications consistency model to a more available one.

The VCML framework is the result of an effort to create a generic modular architecture that allows the importation of well known consistency models with ease, and has proven to achieve this in [4]. The framework is split into seven modules that we will detail below, in order to understand how to take advantage of them to make the change of consistency protocol possible dynamically in runtime.

2.3.1 Framework API

The Variable Consistency Messaging Layer framework provides two APIs, a public API for applications and a private API for the communication between nodes. They provide the following methods:

- `newMessage(content)`
- `newMessage(content, metadata)`
- `replicateMessage(content, metadata)`

They have the option to add additional information for consistency models that require it.
We can observe how VCML’s modules interact in Figure 4. In Figure 4, we can analyze the flow of a new message upon arrival to the framework. It arrives via a call to the `newMessage(content)` which is exposed in the API (Sec. 2.3.1). To decide what to do with the message, the node will invoke the `getRole()` method exposed in the Group Membership module (Sec. 2.3.2). If the node itself is not a timestamper, it should find one via the `getTimestamper()` method, also exposed in the Group Membership module, and forward the message to it.

After arriving at a timestamper node, the message will be ordered, making use of the `timeStamping()` method exposed in the Ordering module (Sec. 2.3.3). For example, if it is a causal consistency system, the method will check and return the dependencies of the message inside the metadata field. Subsequently, the message is marked with a timestamp, and it is ready to be replicated. So, the message is passed to the Replication module (Sec. 2.3.4) by the `replicate(content, metadata)` method to initiate the process of replicating a message.

The Replication module has to know where to replicate. Thus, it invokes the `getReplicationTargets()` method on the Group Membership module that returns a list of members to where it must replicate and then, it calls `replicateMessage(content, metadata)` on each member of the list. Some consistency models require the system to wait for the replication response before considering the message delivered. The `waitQuorum()` method of the Quorum module (Sec. 2.3.6) only returns when the replication conditions are satisfied. Finally, the message that was replicated needs to be applied to the local replica by the `apply(content, metadata)` method of the Replication module.
2.3.2 Group Membership

To establish a consistency model between a distributed system’s nodes, one must maintain a stable view of the membership of the system so as to know which copies of each data element exist and where they are located. This module keeps track of the system’s nodes, as well as their roles. There are two membership types: Timestamper and Forwarder, and the members have to specify which role they exercise. Timestampers mark new messages with timestamps, and Forwarders depend on Timestampers: if they receive a new message, they need to redirect this message to a Timestamper so it can be timestamped. In decentralized systems all members can behave as timestampers [7]. It is also possible to implement leader election on top of this module for systems that use it [25].

The most important method the group membership module exposes is:

- **getReplicationTargets()** - returns a list of nodes of the system to where a member has to replicate a message to. The result of the return of this method depends on the protocol being used: it can return all members of the system if messages have to be replicated to all the nodes, or just to some (gossip schema).

2.3.3 Ordering

This module is responsible for defining the timestamping mechanism that will be used, such as Lamport clock, vector clock or physical clock. It is also responsible for timestamping messages, assigning an order to the messages that arrived to the framework. Lastly, it provides conflict handling, to define an order for messages that have the same timestamp.

Messages are marked with metadata when they are processed by members, in order to distinguish them from messages forwarded by other members of the system. This metadata is a key-value map that contains information about the message, such as the message origin source or even the progress status of the message in the framework, like the number of replication successes. Dependency lists can also be generated and saved in the message’s metadata in the case of causal consistency systems.

This module provides the following methods:

- **timeStamping(content)** - this method timestamps messages, assigning an order to the messages that arrive to the framework.

- **compareMessages(message1, message2)** - this method compares messages to define an order for messages with the same timestamp.
2.3.4 Replication

As the name implies, this module is responsible for coordinating message replication. It interacts with the Group Membership (Sec. 2.3.2) and Quorum (Sec. 2.3.6) modules to replicate messages. This module gives the programmer the option of applying messages to the system before or after it has been replicated. This is useful in cases where the system has to first apply a message, then answer to a client and only after starting the replication process.

This module has the following methods:

- \texttt{replicate(content, metadata)} - replicates a message to the replication targets defined in the Group Membership module, \textit{getReplicationTargets()}, and with the help of the Quorum module, \textit{waitQuorum()}, it also knows when a message can be marked as replicated.

- \texttt{apply(content, metadata)} - applies the message to the system member that received it. It is supported by the Delivery Condition (Sec. 2.3.5) module.

It is possible to call the \texttt{replicate(content, metadata)} method inside the \texttt{apply(content, metadata)} method. This is useful when using gossip type protocols, where after receiving and applying a replicated message, the member has to replicate the message to other members.

2.3.5 Delivery Condition

In this module the programmer defines the conditions required for a message to be applied in the system. It interacts with the Ordering module (Sec. 2.3.3) to compare messages in the case of causally related messages.

This module exposes the following methods:

- \texttt{tryToApply(content, metadata)} - checks if all conditions that were initially defined are satisfied, and if so, applies the message to the system.

- \texttt{addToRemoteWaitingDep(dependencyRequest)} - adds a remote request of dependency to a queue to be answered when satisfied by a local system.
2.3.6 Quorum

Some consistency models demand a quorum, and this module allows the implementation of quorum algorithms and their semantics. This module only exposes one method:

- \texttt{waitQuorum()} - responsible for guaranteeing that the other members have returned a delivery status message and quorum has been formed.

2.3.7 Communication

This module's primary function is to essentially deal with the necessity to convert data types between the system and the framework itself, as well as the framework and the real communication protocol chosen by the programmer. The module is split into two parts: internal communication, for communications that take place within the member such as a call to write or read on a local database, and external communication for communications that occur from the node to the outside, for example, replicating to other members or answering to a client.

2.4 Motivation for Dynamic Adaptation

The decision to dynamically adapt a system is made when the current implemented model stops achieving its purposes efficiently. By efficiently, we mean that it might be overusing or underusing resources, such as CPU and memory, overloading one single node, being too costly in terms of inconsistency, causing a loss of performance due to high amounts of conflict handling for example, or even low throughput or high latency [26]. Without a clear picture of workload and network performance [27], reconfiguration of the system can't be justified, as well as the effectiveness of quality of service implementations or other technologies [28].

Dynamic adaptation is a recurring topic in several areas. Complex software systems like Apache [29], Tomcat [30], MySQL [31], provide the developers with numerous configuration options. This high configurability allows adapting the behavior of these services during execution. Changes in system workload and available resources can result in a need to reconfigure the system [32].

One approach to adapt a system's or service's composition is to consider it as a black box and use control theory as well as learning techniques to derive adaptation policies [33], [34]. This approach is, however, expensive. If the system configuration changes, the learning process has to be repeated, and the same applies for changes in the workload, where a small change can be very impactful on the adaptations that need to be selected. The system architect can also specify low-level adaptation policies for the system's composition. But this approach becomes less effective when the complexity of the system composition increases. Cholla [35] addresses a problem similar to the one previously mentioned, proposing a solution based on fuzzy control rules. Rules can be developed independently, but additional coordination rules are often required. These coordination rules are specific to the
chosen set of rules that are developed. A key work that shows the benefits of changing consistency dynamically is Kraska. In Kraska et al. (2009) [36] consistency rationing in cloud systems is explored. They experimented with different priority groups each representing a possible system state inspired by real world situations. In the group we will be analyzing, data is accessed concurrently by many clients, so the consistency level required can change anytime depending on the current situation. In Kraska et al. (2009) the price of inconsistency is calculated as the percentage of incorrect operations causing overhead due to conflict handling. Even though Kraska is used for transaction operations, which are not applicable for this work, the experiments we will study below prove to be relevant to the problem in question. They experimented with different policies each describing when to switch consistency levels, having their dynamic policy emerge as the best solution: “Probabilistic guarantee depending on the frequency of updates like general policy, but also using the data values. In simple terms it uses the probability that an update will cause the value to exceed its constraint.”. This policy provides performance and low costs as it’s dynamic and reactive.

Two different tests were run, one with a 80-20 rule distribution as explained in Gray et al. (1994) [37] and an uniform one. We analyze the response time results in Figure 5. The figure shows the response times for constant strong consistency (“A data”), constant available consistency (“C data”) and for dynamic adaptation of consistency (“Dynamic”). We can see that available consistency is much faster than strong consistency, but only ~5ms faster than the dynamic adaptation approach. Figure 6 shows the results when it comes to cost per transaction. We can see that constant available consistency is more efficient than constant strong consistency when it comes to expensiveness of the operations, but in the 80-20 distribution it is much more expensive due to the high penalty costs. The dynamic adaptation approach, however, shows to be the cheapest and the more consistent approach.

From these results we can conclude that dynamic consistency is the most efficient approach when it comes to low response time and monetary cost of transactions.

Dynamic adaptation shows to have good results. Being able to have this dynamicness in system’s consistency models could prove to have good results as well. Strong consistency provides good consistency while available consistency provides good performance, but the inverse is not true. There is a tradeoff that has to be taken into consideration when making the decision of whether to use strong or available consistency.
2.4.1 Handling transient state in dynamic adaptation - Consistent cuts

When a distributed system becomes faulty in terms of, for example, too many events between processes becoming conflictual, or even when a system requires some sort of reconfiguration of the way it deals with such events during runtime, a need to snapshot a state of the system emerges[38]. The system is then able to return to the state that was snapshot. Consistent cuts lets us define a clear cut frontier in the transition of state in a distributed system to solve this problem [39]. Because physical time cannot be perfectly synchronized in a distributed system it is not possible to gather the global state of the system at a particular time. Cuts provide the ability to assemble a meaningful global state from local states recorded at different times. But for a cut to be consistent, and provide a consistent global state that can be used to solve the issue at hand, no message may be sent before the global cut and received after it [40].

![Fig. 7. Consistent vs inconsistent cut [41]](image)

Let's consider a distributed system using causal consistency, and that a sudden need to change its consistency emerges. Assuming the new consistency model does not ensure that dependencies between messages, the following situation can occur: a message N arrives at a node, but before being applied, it depends on a message M to be applied posteriorly. But because of a transition, M is now set for the new consistency model, and upon arriving at the node does not trigger the dependency check of message N, making it so that it will never be applied. Using a consistent cut to deal with the transition would solve issues of this nature, and more.

To solve these issues, in Friedemann Mattern's work [39], he tackles the problem by equipping each process with a clock, consisting of a vector of length n, where n is the total number of processes. This enables all the processes to have some knowledge about other processes’ global time approximation. Everytime a process Pi 'ticks' it fixes its 'next' time s to the common snapshot time, and broadcasts it to all other processes. It then does not execute any event until it knows that every process knows s. It then ticks again, getting to the state of the common snapshot and takes a local snapshot. Pi broadcasts a dummy message to all processes to force them to advance their clocks to a value higher or equal to s. Each process then takes a local snapshot and sends it to Pi when their local clocks become equal or higher than s. In [39] solutions to the ‘frozen’ time that Pi suffers are also described, making use of virtual processes.
3 Architecture

In this section we will be presenting the design of our proposal: the adaptation of the Variable Consistency Messaging Layer in order to support the runtime reconfiguration of the framework’s consistency model. To ease the reading of the document, let's call this new version of the framework **dynamicVCML**. This goal is summarized in the following requirements:

1. The system should be continuously capturing and evaluating the tracked metrics, and deciding if there is a need for a reconfiguration of the consistency model.
2. All messages that arrive at the system before a reconfiguration happens, should be processed with the configuration setting that was taking place at the time of their arrival, while messages that arrive after the reconfiguration should be processed accordingly to the new setting.
3. The programmer should be able to effectively define module configurations in the configurations module.
4. The programmer should be able to configure when reconfigurations should occur, and how often the capture and evaluation of metrics occur.

It is also in our interest to not lower the performance of the framework. Therefore we set upon ourselves the following restrictions: a transition to a new configuration should take no longer than 2 seconds to conclude and the overhead caused by tracking metrics shouldn't be higher than 10% of the base latency.
We will first showcase the new Tracker API added to the framework and its methods. The methods this API exposes make it possible for the system’s consistency model to become dynamic. We will then demonstrate in depth how the reconfiguration of the system works, as well as the process of monitoring the system’s state via the tracking of metrics. Lastly, we will describe the changes and additions required to track latency, throughput and stale read rate, that will be handled by the Tracker node using a set of rules defined by the programmer, identified by a configuration id. This configuration id refers to one of the configurations built by the programmer, and indicates that specific configuration should take place in the system.
3.1 Tracker role

Each partition of a system using the framework must have a centralized node, let’s call it Tracker node, that has the sole purpose of receiving values captured by other nodes in the system and store them, to then produce relevant metrics, and of analyzing these metrics to make consistency reconfiguration decisions as well as reconfiguring the system during runtime. There should only be one tracker node per partition of a system. Therefore an API was added to the framework, making available methods used by the tracker node, as well as methods that will be used by the rest of the system’s nodes for them to contribute to the tracker’s work.

3.2 Tracking metrics

To help the programmer decide when a reconfiguration of the framework’s consistency model should happen, we added the monitorization of metrics to the framework as well as a new module where the programmer should set rules for when a new consistency model should take place with the help with the metrics provided by the framework. This new module is called Evaluation module and is later described in Section 3.3. Our new version of the framework is able to track latency, throughput and stale read ratio during runtime. We go more in depth in how this monitoring works in Sections 3.2.2 and 3.2.3.

The relevant methods described in the tracker API for this section are:

I. Tracker node:

• gatherMetrics() - this method asks all other nodes to send their captured metrics, making use of the sendMetrics() method. The frequency at which this method is called is configurable by the programmer, i.e., every x seconds, at a specific time of the day, or even after x operations have been done.

• evaluateMetrics() - after the tracker node has received the metrics from all the nodes, they are analyzed in this method. An average for each metric is calculated to then be sent to the new Evaluation module (Sec. 3.2.4), where the programmer can define multiple conditions for multiple reconfigurations of the system, making use of the gathered data. If the result of this evaluation is a configuration that is not currently active, the system proceeds to enter the reconfiguration phase, by calling the sendSwitchConfigWarning() method.
II. System node:

- `sendMetrics()` - sends the accumulated metrics to the tracker node. The stale reads calculation is called in this metric, with the method `checkStaleReads()`.

- `checkStaleReads()` - performs the calculation of how many stale reads have been done since the last metric gathering phase. How this method works is explained in section (Sec. 3.2.3).

- `resetTracking()` - clears the values that have been stored since the last metric gathering phase, after sending them to the tracker node via the `sendMetrics()` method.

![Fig. 9. Metrics tracking and evaluation](image)

The process of metric monitoring (Fig. 9) starts with each node of the system capturing the relevant values later described in Sections 3.2.2 and 3.2.3. The metrics are sent to a tracker node each time the `gatherMetrics()` method is invoked (1), (2). This method is invoked periodically, according to the set interval defined by the programmer.

The tracker node then, using the values gathered from all its nodes, and using the evaluation module (3), where the metrics will be evaluated and compared to the thresholds set by the programmer (Evaluation module, Sec. 3.2.4), will decide if a reconfiguration is to be made (4). If the metrics trigger the need for a reconfiguration, the evaluation module returns the appropriate configuration id (5), signaling that a reconfiguration will occur and which configuration should take place.
3.2.2 Latency and throughput monitoring

In order to calculate latency, we took advantage of the metadata field that is already appended to existing messages. The metadata attribute is a key-value map where developers are able to append metadata information on a per-request basis. A timestamp is added to the metadata of messages, and compared to the current time upon their arrival. The time difference between the current time and the metadata timestamp gives us a clear representation of the time it takes for the nodes of the system to communicate between them.

For throughput, we keep track of how many reads and writes are performed. Each node has a counter incrementing each time a read or write operation is completed, and after each time interval stated by the programmer in the gatherMetrics() method, it sends the counter value to the tracker member, resetting the counter.

3.2.3 Stale read ratio monitoring

To measure the stale read ratio (Sec. 2.4.2. II) of the system, additional metadata was added to new messages that enter the framework. Metadata now has a new field: creationTime. When a new message enters the framework, its time of creation is stored in the creationTime variable stored in its metadata.

System nodes have a cache running that will keep track of read times. Assuming that the clocks have a minimal drift from each other, each time a read type message arrives at a node, the cache stores a timestamp of the read. When a write type message is applied to the system, the tracker node stores the creationTime of the write to later compare it to the timestamps of the reads stored in the nodes' caches. Each read that was made in a timeframe older than the creationTime of the write message is considered a stale read. Every time the tracker node invokes the gatherMetrics() method, the node calculates the amount of stale reads made in that timeframe, and is then sent to the tracker node. The algorithm is shown in Listing 3.2.1, where keysWrittenInTheNode are the writes that have been applied to the node executing the algorithm, keysWrittenInTheSystem is a list containing the keys that have suffered a write operation in any node of the system, and keyReadsInTheNode is a list containing all the keys that have suffered a read operation in the node, having as a value an array of all the timestamps of when that key has been read in that node.

Even though this approach only takes into consideration the last timestamps regarding each key’s write time, in the system and the node, before the gatherMetrics() method is invoked, it still accomplishes to give the programmer an idea of how many stale reads happen between the time of when a write is applied in the system and the time when of when it is applied in the node. Calculating the stale reads for all the writes done for each key between gatherMetrics()'s calls proved to be too expensive in terms of performance.
Evaluation module and Configurations

A new configurable module was added to the framework where the programmer is able to define the conditions for a reconfiguration to occur and to which configuration to switch, making use of the metrics that the framework provides. Each configuration setting is associated with an id - \textit{config\_id}. This module is used by the tracker node each time a metrics collection happens (\textit{gatherMetrics()}), to evaluate the metrics received by the system nodes and return the appropriate id. The programmer is able to define each configuration in a text file, specifying which new modules should be instantiated, what clock type is to be used as well as other parameters (Listing 3.3.2). This module should return a \textit{config\_id} to the tracker class (Listing 3.3.1).

Listing 3.3.1. Example of a set of rules in the Evaluation module

```java
Result: config_id
if throughput < 1500 or writeLatency > 2000 then
    return config_id = 1;
end
else if staleReadRatio > 50 and readLatency > 2000 then
    return config_id = 2;
end
```

Listing 3.3.2. Example of a set of rules in the Evaluation module

```java
Result: staleReads
staleReads=0;
foreach key in keysWrittenInTheNode do
    if key in keysWrittenInTheSystem then
        keySystemWriteTime = key.getLastWriteTimeInTheSystem();
        keyNodeWriteTime = key.getLastWriteInTheNode();
        if keySystemWriteTime > keyNodeWriteTime then
            foreach keyReadTime in key.getReadTimes() do
                if keyReadTime > keySystemWriteTime then
                    staleReads++;
                end
            end
        end
    end
end
```

Listing 3.2.1. Stale reads algorithm
3.3.1 Configurations

As previously stated, the programmer is able to define different framework configurations, defining the type of clock, the version of modules, etc. In order for the programmer to have 2 different Delivery Condition modules to use in different configurations, he/she should treat them as two different modules, implementing them separately. The version of the module is then stated in the configuration text file. Each system node must have all the different implementations of modules as well as the config files defined in its framework. In Listing 3.3.2 we have two examples of configuration files.

Listing 3.3.2. Example of two configuration text files

3.4 Reconfiguring the framework

In this section we describe the steps that the framework takes in order to change its consistency model. The relevant methods in the tracker API are:

I. Tracker node

   - sendSwitchConfigWarning() - starts the reconfiguration process of the consistency model of the entire system. Upon calling this method, the tracker will be listening for acknowledgement messages, to make sure the other nodes of the system are on par with himself.

   - switchToConfig(config_id) - after receiving all the ‘acks’, the tracker invokes this method, telling the nodes that the reconfiguration is to proceed whilst sending them which configuration they should reconfigure to.

   - cancelSwitchConfig() - if after x amount of time (defined by the programmer), the entirety of the ‘acks’ has not been collected, this method is invoked to tell the nodes
that the reconfiguration is to be aborted having them return to their normal work.

II. System node

- `reConfig(config_id)` - performs the reconfiguration of the node, by instantiating the modules that take part of the new config by reflection, as well as other values such as the clock type, etc.

![System reconfiguration diagram]

**Fig. 10. System reconfiguration**

When the decision to reconfigure the framework is made, the tracker node invokes the `sendSwitchConfigWarning()` method (Fig. 10), which starts the reconfiguration process. This execution will be split in two phases: firstly, it will signal to all the other nodes of the system that a reconfiguration will occur (1).

The system nodes will acknowledge the reconfiguration after they are finished with their current tasks, as well as solving pending dependencies or any similar tasks regarding their current consistency model (2). During this time new messages that arrive at the system are stalled before any metadata is applied to them. These messages are tagged with the new configuration setting id. Messages that were already in the system bearing the old configuration setting id, will be processed using the consistency model relative to the configuration’s id they bear.

When the tracker receives the acknowledgement of all the other nodes, it will send them the `config_id` regarding the new configuration (3). Each node then executes the reconfiguration, according to the configuration id that is given to it (4). Configuration values are updated according to the values stated in the config file, and the modules associated to that config are instantiated by reflection and take the place of the old configuration’s ones. However, the old configuration’s modules will stay instantiated, to process messages that were caught in the queue that are still intended for that configuration. When
the reconfiguration has been completed the messages are unqueued and work resumes as messages are processed by order of arrival to the framework.

If the tracker fails to receive all acks needed in a period of time set by the programmer, a cancelSwitchConfig() message is sent to all nodes, and work will resume as before.

Even though consistent cuts were discussed in Section 2.4.1, for ease of implementation we decided to opt for this queueing solution. With our approach some messages may never be applied, in cases where messages are waiting for a dependency but the new configuration does not check for dependencies, these messages will never be applied, and can be considered lost. The use of consistent cuts strategies would improve the correctness of our solution, and is still considered for future versions of the dynamicVCML framework.
4 Evaluation

4.1 Methodology

To evaluate our proposal, we will compare a real system implementation using the Variable Consistency Messaging Layer with a static consistency model against our equivalent implementation of the system using dynamicVCML, with the goal of analyzing the overhead caused by the new additions made to the framework, as well as possible gains in performance in some specific scenarios. We will be taking into consideration the following metrics: latency and throughput.

4.2 Experimental Setup

Since the first version of VCML was implemented and tested in a Distributed Key-Value Framework [42] system, we decided to implement our new version in the same system for ease. More specifically the COPS [9] implementation. DKVF, is a framework that allows programmers to easily create and evaluate distributed key-value stores. DKVF based systems offer the client and the server-side that extends the client and server-side DKVF, respectively. It relies on Google Protocol Buffers [43] for marshalling and unmarshalling of data for storage and transmission. DKVF comes with a Berkeley-DB driver which it uses as a storage engine, and can be configured to handle data replication. We do not use the latter functionality, as one of the requirements of VCML is to provide replication. DKVF includes a YCSB driver [44]. We used this already implemented feature of DKVF, making variations to the number of operations and percentages of reads and writes, to give us the throughput and latency. In order to evaluate our framework in a COPS system, we built a cluster consisting of 9 servers and 6 clients. The cluster had 3 partitions each with 3 servers and two of three replicas of the cluster were connected to a hub that was connected to three clients each. Each partition's replicas were connected to other replicas with the same partition. The remaining replica actuated in the system as another replication point, not being connected to any client. We assumed full replication between replicas. For the cluster with our implementation of VCML each partition also had a centralized tracker node (3.1).

Each node of the cluster ran in an independent machine with 1 vCPUs, 2.13 GHz, Intel Xeon E5506 and 2 GiB memory RAM.
4.3 Experimental Results

We performed measurements varying the percentage of write and read operations as well as the number of operations to be performed. Each client had 8 threads, increasing the amount of load applied to the system. A record count (YCSB property) of 1000, which means that it created 1000 records on load phase of the YCSB execution. We varied the read:write operations ratio between 95:05, corresponding to a read-mostly workload, and a 50:50 ratio corresponding to an update heavy workload.

4.3.1 Overhead caused by the monitoring of metrics

For the first experiment we wanted to measure the overhead caused by the monitoring of metrics. We compared the original version of VCML (staticVCML) with our version (dynamicVCML). Since we only wanted to measure the overhead caused by the tracking of metrics, no reconfiguration happens in these experiments. The results are presented in Tables 4.1 and 4.2. We note that the minimum overhead observed is 1.6%, and the maximum 8.4%. The higher values of overhead happen with a higher number of writes. For more detailed discussion refer to 4.4.

<table>
<thead>
<tr>
<th>Operations Count</th>
<th>Throughput Overhead</th>
<th>Write Latency Overhead</th>
<th>Read Latency Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>1.6%</td>
<td>3.9%</td>
<td>1.5%</td>
</tr>
<tr>
<td>100000</td>
<td>1.6%</td>
<td>5.6%</td>
<td>1.3%</td>
</tr>
<tr>
<td>200000</td>
<td>2.5%</td>
<td>5.1%</td>
<td>2.4%</td>
</tr>
<tr>
<td>400000</td>
<td>5.1%</td>
<td>8.8%</td>
<td>4.7%</td>
</tr>
</tbody>
</table>

Table 4.1. staticVCML vs dynamicVCML (with no configuration change) overhead in COPS - 95:05 operations ratio

<table>
<thead>
<tr>
<th>Operations Count</th>
<th>Throughput Overhead</th>
<th>Write Latency Overhead</th>
<th>Read Latency Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>4.6%</td>
<td>6.8%</td>
<td>4.4%</td>
</tr>
<tr>
<td>100000</td>
<td>4.8%</td>
<td>8.4%</td>
<td>4.6%</td>
</tr>
<tr>
<td>200000</td>
<td>5.8%</td>
<td>8.4%</td>
<td>5.7%</td>
</tr>
<tr>
<td>400000</td>
<td>8.4%</td>
<td>12.1%</td>
<td>8.0%</td>
</tr>
</tbody>
</table>

Table 4.2. staticVCML vs dynamicVCML (with no configuration change) overhead in COPS - 50:50 operations ratio
4.3.2 Potential gain by reconfiguring the framework

We redid the experiments, but this time with a reconfiguration of the consistency model happening in the *dynamicVCML* version. Midway through the experiments, the *dynamicVCML* version of the framework reconfigured to a more relaxed consistency model, that ignores all the dependency handling that COPS’s causal+ offers. The *staticVCML* version stayed unchanged through the experiments. In Tables 4.3 and 4.4 we can observe the gains in performance of using our version of the framework compared to the original version. Keep in mind that a negative value means that there was no gain, but a loss in performance in its specific setup. We note that the maximum gain in throughput observed is 7.2%, while in the worst case we still have a loss of 0.2% in throughput. The lower values of throughput gain happen with a higher number of writes. For more detailed discussion refer to 4.4.

<table>
<thead>
<tr>
<th>Operations Count</th>
<th>Throughput Gain</th>
<th>Write Latency Improvement</th>
<th>Read Latency Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>7.2%</td>
<td>24.0%</td>
<td>-1.2%</td>
</tr>
<tr>
<td>100000</td>
<td>7.2%</td>
<td>21.8%</td>
<td>-1.1%</td>
</tr>
<tr>
<td>200000</td>
<td>6.1%</td>
<td>22.3%</td>
<td>-2.1%</td>
</tr>
<tr>
<td>400000</td>
<td>3.3%</td>
<td>17.5%</td>
<td>-4.4%</td>
</tr>
</tbody>
</table>

**Table 4.3.** *staticVCML* vs *dynamicVCML* (with a reconfiguration mid experiment) gains in COPS - 95:05 operations ratio

<table>
<thead>
<tr>
<th>Operations Count</th>
<th>Throughput Gain</th>
<th>Write Latency Improvement</th>
<th>Read Latency Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>4.0%</td>
<td>14.9%</td>
<td>-3.8%</td>
</tr>
<tr>
<td>100000</td>
<td>3.7%</td>
<td>12.4%</td>
<td>-4.0%</td>
</tr>
<tr>
<td>200000</td>
<td>2.6%</td>
<td>13.0%</td>
<td>-5.1%</td>
</tr>
<tr>
<td>400000</td>
<td>-0.2%</td>
<td>8.4%</td>
<td>-7.4%</td>
</tr>
</tbody>
</table>

**Table 4.4.** *staticVCML* vs *dynamicVCML* (with a reconfiguration mid experiment) gains in COPS - 50:50 operations ratio
One last experiment was conducted, where we changed the read:write ratio during the experiment. Every 50000 operations the write ratio will rise by 9%, while the read ratio will decrease also by 9%. The experiment starts with a 95:05 ratio and ends at a 50:50 ratio. The measurements were taken every 50000 operations. The setup is similar to the latter experiment shown above, where the dynamicVCML version reconfigures to a more relaxed consistency model mid-experiment. This experiment’s results are shown in Figure 11.

Fig. 11. staticVCML vs dynamicVCML throughput with a gradual increase of the write ratio

4.4 Discussion

The results show that our new version of the framework adds an overhead to the original version. It is no surprise that adding tracking mechanisms that involve communication between nodes reflects in this overhead increase. However we also observed the potential gains in performance that our framework can bring to a system.

In the first experiment the overhead between the staticVCML and the dynamicVCML, in a scenario where no reconfiguration happens, varies between 1.6% and 5.1% for the 95:05 operations ratio (Table 4.1) and between 4.6% and 8.4% for the 50:50 operations ratio (Table 4.2). It makes sense for the overhead to be significantly higher on the 50:50 ratio experiment, since more write operations result in more stale read checking (Sec. 3.2.3).

The second experiment shows the potential gain in performance that our version of the framework brings. We can observe a gain of 3.3%-7.2% of throughput for the 95:05 operations ratio experiment (Table 4.3) and a gain of 0.0%-4.0% of throughput for the 50:50 operations ratio experiment. This is due to the very noticeable improvement of the write latency, where we can notice a decrease of 17.5%-25% and 8.4%-14.9% for the 95:05 and 50:50 operations ratio experiments accordingly. The read latency still shows an overhead, since the new configuration doesn’t contribute for faster reads, but for faster writes. The reconfiguration to a consistency model that has no dependency checking for new write messages allows for much faster write operation flows. These improvements are almost doubled for the 95:05 ratio experiment compared to the 50:50 ratio experiment, since there are more reads being done before writes, bigger dependency lists are created that need to be satisfied before write operations can be considered complete. Therefore, the removal of these satisfactions is more noticeable in the mostly-read experiment.
During earlier experiments we encountered a throughput overhead higher than 10% on all experiments, some of which showcased up to 20% overhead just by adding metrics tracking. These numbers were too high to make our solution reliable. We decided to investigate where most of this overhead came from, and realized that the stale read ratio calculation (Sec. 3.2.3) was causing most of this overhead. In this earlier version this calculation was being made in all nodes everytime a new write message was made in the system, which proved to be too expensive. We decided to make it so instead of this calculation being made on all nodes every time a new write message entered the system, to only make it when the tracker node asked for metrics, via the `gatherMetrics()` method. Even though this greatly decreased the overhead of our solution, it made the stale read tracking less accurate, as described at the end of (Sec. 3.2.3). We concluded that reducing this accuracy in order to decrease the overhead by ~10% was worth it. With this change the programmer still has a general idea of how many stale reads happen between the time of a write type message entering the system, and the time it is applied in a node.

The reconfiguration of the system during runtime did not show to be too costly, and is easily overtaken by the performance gain from it as seen in Table 4.5, where the new version shows a gain in performance of up to 13.6%.
5 Conclusion

Systems tend to be built with a consistency model implemented coupled with their implementation, which makes switching between consistency models a tricky task. Thus, usually when a consistency model of a system has to be changed, either the system code needs to be deeply rewritten or replaced by a different consistency system. Variable Consistency Messaging Layer solves this issue by allowing a swift change of consistency models without a complete rewrite of the system, but this reconfiguration can only be done during build time.

With our analysis we explored how depending on the current state and workload of the system, different consistency models prove to be more effective than others when it comes to performance and consistency. Some systems that have strong consistency models implemented into them can become very unavailable during high workload periods, resulting in a poor user experience, and the trade off between losing some of its consistency for better performance could be valuable at these times. The same is also true for systems that use available consistency models, where during low workload periods consistency is needlessly low since emphasis in performance is not as required at that given time.

We proposed a variation of Variable Consistency Messaging Layer that allows the programmer to set up different consistency configurations in build time, and has the framework switch between configurations depending on the state of the system, making use of captured metrics to evaluate the performance and consistency of the system at any given time.

To evaluate our proposal, we measured the throughput and associated overhead between the original implementation of VCML and modified implementation with our framework. Although our implementation passively downgrades the performance of a system, due to the tracking and calculation of metrics, the ability to change the consistency model during runtime makes this downgrade in performance worthwhile, as we discussed in the previous section, where I found a potential gain in throughput of 13.6%.

5.1 Future Work

From what we discussed regarding our solution, there are some optimizations that could be done. The main cause for the passive overhead our solution exposes is the tracking of stale reads. An optimization to the stale read monitoring strategy is something that could prove worthwhile.

The way we deal with messages during a reconfiguration of the framework is not the most correct. Using a consistent cut strategy would improve the correctness of our solution and is something that could be revisited in future versions of the framework.
References
