Runtime Consistency Adaptation

Vasco António de Aguiar e Morganho
vascomorganho@tecnico.ulisboa.pt
Instituto Superior Técnico
Universidade de Lisboa

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 consistency 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 consistency 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. Performance 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

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 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, 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 [18] theorem states that having these three properties simultaneously is impossible. Insuring a high level of consistency at all times in terms of synchronous replication means a loss in 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 data 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 [19], 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, causing a not as desirable user experience.

When implementing a new system the developers have to be very careful about which consistency model to implement, 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 [8] uses a set of well-defined modules, aiming to frame the most common consistency protocols within these modules, as well as, to ease the switching of consistency
protocol 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 concludes in a loss in 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
Consistency models are essentially a contract between processes, clients or users and the data store and the replicas that store the data [11]. 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 [17]. But to have strong consistency, the scalability and performance of the system must be compromised. Network delays, acquisition of locks, and 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 [10], [4], [16], [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 [10], [4], [16], 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 Silberstein et al..

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 reliable scalable distributed systems, so the real tradeoff that has to be defined is between consistency and availability. In [2], 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 Golab et al. [5] 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.

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 in performance due to high amounts of conflict handling for example, or even low throughput or high latency [7]. Without a clear picture of workload and network performance [21], reconfiguration of the system can’t be justified, as well as the effectiveness of quality of service implementations or other technologies [6]. Dynamic adaptation is a recurring topic in several areas. Complex software systems like Apache [20], Tomcat [1], MySQL [12], 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 [15].

Dynamic adaptation shows to have good results [37]. 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.

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 (henceforth called 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.

5. 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.

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.

![Fig. 1. Overview of dynamicVCML](image)

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 monitorization 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.

![Metrics tracking and evaluation](image)

**Fig. 2.** Metrics tracking and evaluation

The process of metric monitoring (Fig. 2) 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 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 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.
3.3 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 - config_id. This module is used by the tracker node each time a metrics collection happens (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 config_id to the tracker class (Listing 3.3.1).

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

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>throughput &lt; 1500 or writeLatency &gt; 2000</td>
<td>config_id = 1</td>
</tr>
<tr>
<td>staleReadRatio &gt; 50 and readLatency &gt; 2000</td>
<td>config_id = 2</td>
</tr>
</tbody>
</table>

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.

When the decision to reconfigure the framework is made, the tracker node invokes the `sendSwitchConfigWarning()` method (Fig. 3), 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.

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 [14] system, we decided to implement our new version in the same system for ease. More specifically the COPS Lloyd et al.,

'Don’t Settle for Eventual'. 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 ‘Protocol Buffers’. 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 Cooper et al., [3]. 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 acted 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.
### 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 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>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 4.2. staticVCML vs dynamicVCML (with no configuration change) overhead in COPS - 50:50 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 4.3. 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>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 4.4. staticVCML vs dynamicVCML (with a reconfiguration mid experiment) gains in COPS - 50:50 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>

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.
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 needless low since enfansis 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.

6. REFERENCES


