CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Evaluating in-memory caching strategies for distributed web services

Master's thesis in Computer science and engineering

Love Lyckaro

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Evaluating in-memory caching strategies for distributed web services

Love Lyckaro

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Evaluating in-memory caching strategies for distributed web services

Love Lyckaro

Supervisor: Luca Di Stefano, Department of Computer Science and Engineering
Advisor: Korp Thidrandir, Antura AB
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Evaluating in-memory caching strategies for distributed web services

Love Lyckaro
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Caching is an incredibly common component in modern computer engineering. Present everywhere, caches benefit greatly from domain-specific knowledge. This thesis, in collaboration with Antura AB, targets caching between distributed web services and their database. The state-of-the-art in this caching domain is to have a shared, networked, in-memory, key-value store, such as Redis. This thesis had two goals. Firstly, to implement an alternative to this shared cache system using caches distributed on each server of the web application. Secondly, to implement a test suite for comparing these cache systems. A common interface was created for these cache systems and both were implemented using F#. A test suite was then created using a sample distributed web application and simulated user requests. The user simulation covered different read-write ratios, domain sizes, and behavior with respect to a commonly requested "shared" domain. The results of the tests found cases where either cache system performed better on average. Although the shared cache system performed significantly better as the degree of distribution increased.

Keywords: caching, distributed caching, redis, distributed systems, web caching, database caching, user simulation.

Contents

1

Introduction

Large memories are slow and small memories are fast. Ideally, one would want a large *and* fast memory. For computer engineers, combining small and fast memories with large but slow memories in an effective way is a well-known challenge. The usual solution to this problem is to introduce caches that allow one to simulate large *and* fast memories. In computer systems, caches can be found everywhere: Multiple levels of cache separate the CPU from RAM, caches accelerate access to comparatively slow hard drives, and Web caches decrease the load on databases and servers.

Although well researched, caching is far from a solved problem. With every domain comes possibilities for specific cache strategies. This project is just one of many taking a look at a specific domain, and considering "is our caching system the best for this use case?". Our domain consists of caching between a distributed Web service and its database. Adding a cache layer between a database and the servers that access it can have many advantages. It can improve latency for the end user, decrease the load on the database, and reduce the overall network traffic. Approaches in caching between servers can also be used to mitigate similar bottlenecks in Web services, for example, between Web clients and servers.

A common strategy in this niche is to have a networked, in-memory, key-value store that is shared between the different instances of the distributed Web application. The major benefit of sharing the cache between servers is that cache retrievals from one server can benefit the others. The drawback is an added latency on every cache lookup and write due to the network.

This thesis explores an alternative to this shared caching system. Focusing on implementation and evaluation of a system where each node of the Web service manages its own in-memory cache. This system decreases read latency, as it does not need to communicate over the network.

For evaluating these caching systems, a benchmarking framework is developed that enables evaluation of these systems across many different domains, with differences in both user and application behavior. This permits us to draw conclusions about under which circumstances we should prefer a distributed cache system over a shared one.

The structure of this thesis is as follows: chapter 2 covers the necessary background, including the basics of caching, caching using in-memory key-value stores, messag-

ing, and our use case. This is followed by chapter 3, which contains a description of the main problems the thesis tries to solve. chapter 4 gives a more detailed description of the technical solutions made in the thesis, as well as the factors taken into account in performance testing. The results of testing are presented in chapter 5. Finally, in chapter 6 conclusions are drawn about the main goals, as well as some recommendations for future work.

2

Background

This section covers the state-of-the-art in database cache systems, as well as criteria for evaluating cache systems, and models to formally describe what guarantees a distributed cache system can offer. Finally, it covers Antura's Web service as a specific case study for evaluating cache systems.

2.1 General overview of caching

Caches are incredibly common in computer systems, and distributed Web services are no exception. In general, a cache consists of a small and fast memory, where one saves the most commonly used entries from a larger slower memory; later, one may attempt to retrieve a needed entry from the fast memory before accessing the slow one, thus being able to simulate a large *and* fast memory. Caches in many domains share a number of characteristics and this subsection aims to cover the ones relevant to this project. We will follow the taxonomy of caching systems on the Web proposed by Wang [1]. In this survey the author identifies many relevant features, among which these are the most important for this project:

1. Where should a cache be placed for optimal performance?
2. How do caches cooperate with each other and with the slow memory?
3. Should the cache prefetch data or not?
4. Can the cache maintain data consistency?

The first point is of particular importance for this project and highlights the fact that one can build many different cache architectures, with different pros and cons. For instance, caches closer to the requester of data can improve latency, whilst caches further from the requester can improve opportunities for data sharing.

Regarding point 2, cooperation between caches and the slow memory is not a trivial task. One must identify where writes are first made: to the cache, or to the slow memory? In the first case, the write would need to propagate to other duplicate entries. In the second case, any cached entries of the overwritten value would need to be invalidated. Moreover, when your cache is distributed, the caches would need to communicate these things to each affected cache instance, as well as the slow memory.

Wang also considers a system where writes go directly to the large, but slow memory (in Wangs use case, a Web server), which then messages the cache to invalidate the entries. One could also think of a use case where invalidation is combined with updates to the stale entries.

Prefetching of data (Point 3) can improve performance, as it decreases mandatory reads from the slow memory. Prefetching is the act of trying to retrieve data likely to be used in beforehand. This can happen both on startup, or through fetching additional data upon the request of a resource.

Data consistency (Point 4) is discussed in subsection 2.2.1.

What one should especially consider about these features, and indeed about caching in general, is the need to tailor one's cache system to the use case. Since Wang's paper, the internet has grown significantly. With this growth new exciting arenas and approaches to caching have come into play, although these four considerations (placement, cooperation, prefetching, and consistency) have all kept their relevance. An example is the recent research into ICN (Information Centric Networking), which is trying to tackle the enormous amount of data throughput needed for modern information distribution, especially for use cases like video streaming. In their survey paper of caching in ICN, Din, Hassan, Khan, *et al.* [2] write about tailoring caches for this use case. Another example of tailoring a cache system to its use case is [3] where YouTube have tailored a memcached-based [4] cache system to their needs using a cache-replacement algorithm based on their users' behavior.

2.2 Database caching using an in-memory key-value store

There are a number of approaches to database caching. Arguably, the most common is to use an in-memory key-value store such as Redis [5] or memcached [4]. Among others, Facebook has used this approach to great effect [6]. The approach relies on having one or more instances of a key-value store between the application and the database. The former is able to serve requests far faster than the latter would. This key-value store is, most often, shared between the different servers, which brings both pros and cons. On one hand, sharing the cache over the network allows for cache misses by one server to benefit the others. On the other hand, sharing the cache over the network also brings overhead. Requests have to go through the network stack, which can be an order of magnitude slower than local memory access. A simplified structure of this type can be seen in Figure 2.1.

In their paper *Design Considerations for Distributed Caching on the Internet* [7], Tewari, Dahlin, Vin, *et al.* explore the considerations one should make when designing a cache system for the Web. The authors focus on large distributed cache systems, where they identify a number of important best practices:

1. Cache systems should share data among many clients to reduce compulsory misses and scale to a large number of caches.

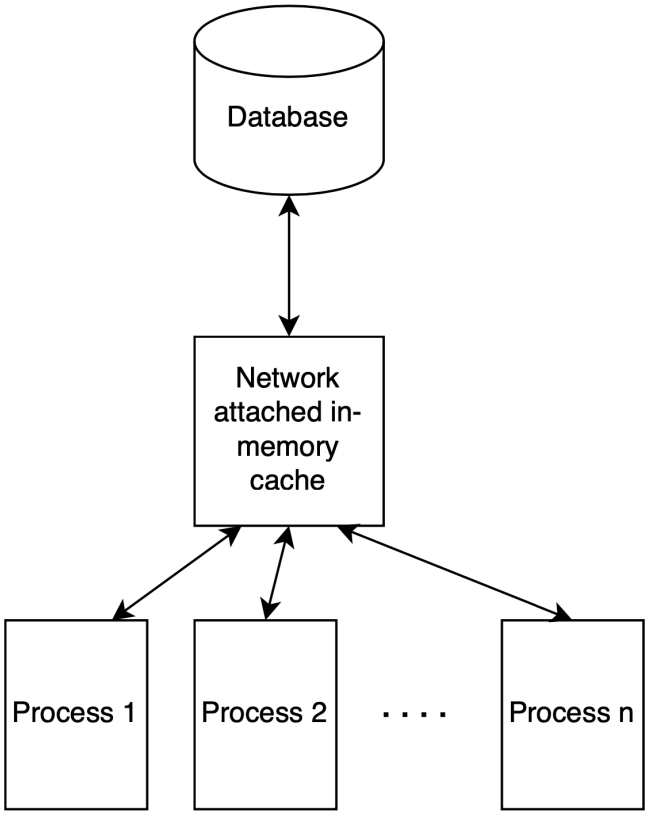


Figure 2.1: A database caching system with a shared in-memory cache.

2. Cache hit time constitutes a significant fraction of the total information access latency. Hence, cache architectures should minimize the cost to access a cache.

3. Even the ideal cache will have a significant number of compulsory and communication misses. Thus, cache systems should not slow down misses.

Interesting for this project is the first note: *Cache systems should share data among many clients to reduce compulsory misses.* In the paper, the authors note that testing on large cache systems reveals that following this rule improves performance. They then use this finding to justify developing a multi-tiered cache system for the Web. However, this system differs somewhat from the ones of interest in this project, both in terms of scale and location of the cache. With their caches lying in proxies between end users and Web applications, rather than between the application and database. An interesting question that then arises is whether the authors' assumption holds in cases like Antura's or if it would be more performant to e.g. lower cache request latency at the cost of decreasing data sharing.

2.2.1 Consistency models

A question relevant to distributed systems in general, and caches in particular, is the question of what guarantees these systems can uphold. This question has been researched before and a number of different *consistency models* have been developed

to describe these guarantees. In the paper *Eventually consistent* [8], W. Vogels reports on the different aspects that have been considered when choosing a consistency model for Amazon's many distributed storage services. Vogels explains the different models through the lens of three different, independent actors A, B and C. These are all processes that try to read from, and write to some distributed storage system. The concistency model is defined by when the actors are able to *observe* changes made to the system by the other actors. Some of the models relevant to this project can be informally described as:

1. *Strong consistency*: after A updates the storage system, any subsequent access by A, B or C returns the updated value. We may also say that the update is instantly *observable*.

2. *Weak consistency*: this model does not guarantee an updated return value after A has updated the storage system. It will usually only guarantee that the updated value becomes observable after some other requirement is met.

3. *Eventual consistency*: a specific form of weak consistency where an update to the storage system is guaranteed to *eventually* be returned on subsequent requests from the actors, given that no other updates are done in the meantime.

4. *Session consistency*: a special case of eventual consistency where requests are viewed through the lens of a *session*. A series of updates and requests made in the same session is guaranteed to be consistent: however, this guarantee is not upheld when switching sessions.

Of particular interest to this project are models 3 and 4, that is, *eventual consistency* and *session consistency*.

Strong consistency might seem the most enticing: after all, it does give the strictest guarantees. Vogels explains that these strict guarantees must be weighed against the consequences of distributing such a system. These are most succinctly summarized by the CAP Theorem[1] [9], which states that a distributed data store can at most provide two of the following three guarantees:

- *Strong consistency*: every subsequent read after a write is correct;

- *Availability*: every request receives a non-error result;

- *Partition tolerance*: system keeps running correctly despite an arbitrary amount of delayed or missing messages between nodes.

Thus, an available partition-tolerant system cannot also provide strong consistency, but may guarantee weaker forms of consistency

[1] Also known as Brewer's theorem, previously Brewer's conjecture.

2.3 Message brokers and the publish subscribe pattern

Message brokers are a component in computer systems facilitating communication between processes or computers. Originally developed for Prolog inter-process communication in [10], the messaging paradigm has gone on to widespread use.

In this project we use a message broker to facilitate a publish-subscribe pattern of messaging. Where messages are *published* to channels, which interested parties (in our case, the other caches) can *subscribe* to. This allows all parties to not have to worry about exactly which processes to communicate with. Originally called "News service", the messaging paradigm was first explained in [11].

2.4 Case: Antura

This thesis is done in collaboration with Antura AB. They are a software company focused on a project management Web service called Antura. This Web service is distributed over many different servers, each of which runs a number of different processes. Antura are currently using a distributed cache system akin to the one under review in this project. However they are unsure about how it performs compared to a shared cache system like Redis [5]. They are also uncertain about what factors they should take into account when choosing whether to migrate to a new system.

In particular Antura are curious about how the prevalence of "common" requests, and the shared state caused by these, affects the performance of both systems. They believe that their user behavior is quite disparate, with most users requesting data only relevant to the project they are working on. Through the use of sticky sessions, where users are consistently routed to the same server instance, they believe that this shared state is further reduced. These circumstances seem to imply that a distributed cache system might be preferable, however Antura currently lack the experimental data to support this conclusion. Another factor Antura are interested in is how the cache performance is effected by further distribution of the application.

3

Main idea

In this section we describe the main goals of the project, namely to:

1. Design a cache system with a per-server in-memory cache; and
2. Evaluate the performance of this type of system against one that uses a shared cache, in order to determine under which circumstances one should prefer the former over the latter.

3.1 Designing a per-server cache system

The first goal of the project was to design a proof-of-concept cache system that had one cache per server. The hypothesis was that, in use cases where the data shared between servers is small, this setup could improve performance by avoiding the network overhead on cache requests. Figure 3.1 shows a scheme of this kind of system:

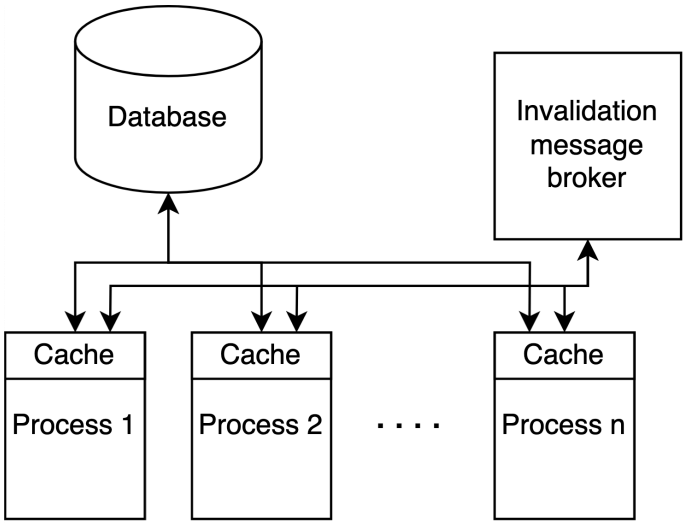


Figure 3.1: The idea for an alternative cache setup. Each server keeps its own cache. Allowing an order of magnitude shorter latency for cache lookups. With the trade-off being extra communication being necessary for cache invalidation due to data duplication.

This idea raised a number of challenges:

- How should duplicate cache entries be handled?
- How should cache invalidation be performed?
- What consistency model should the cache fulfill?

3.2 Comparing per-server and shared caching

The second goal of the project concerns the viability of a cache system such as this. The research question can be summarized as: *Under which circumstances does a per-server cache system outperform a shared equivalent?*

The project aims to create a test suite for testing these cache systems, allowing us to measure performance metrics like cache hit rate and latency. As with the implementation of the system, this question raises a number of challenges:

- How does one best test the performance of a cache? How can different request patterns be simulated in a good way?
- Which request patterns would favor one over the other?
- How does the system performance scale as more servers are added?

3.3 Delimitations

To keep the project viable within its allocated time frame and to focus attention on the matter at hand, a number of topics were not explored in this thesis. Additionally, as the project was necessarily focused on Antura's use case, some results might not be as general as they could be.

The project did not explore:

- How different shared and networked caches compare against each other.
- Cache replacement algorithms: though cache replacement algorithms are important in many cache systems where space is scarce, Antura's use case rarely runs out of memory for the cache.
- Distributed messaging. One could imagine a system where individual caches communicate directly with each other, avoiding a potential bottleneck in the message broker. This was not explored because we did not foresee the message broker being a bottleneck in this use case, and we feared that this would add a rather large amount of complexity to the project.
- Prefetching upon request. Prefetching can be done both on startup of the system and upon request of a resource. This thesis only covers prefetching on startup, as that was most feasable in Antura's use case.

4

Methods

This chapter outlines the methods used for solving the challenges set out in the previous chapter, both regarding the implementation of the cache system and testing. Throughout these parts, the language F# has been used for almost all programming tasks. F# is a functional programming language based on the dotnet platform. It has support for object orientation, and importantly in this project, for classes and interfaces.

4.1 Defining a cache model

An important factor in creating the caches in this project was specifying what we mean by cache and what guarantees we want this cache to provide. As outlined in subsection 2.2.1, there are a number of consistency models a distributed system can enforce. We have elected to have our caches fulfill *eventual consistency*: that is, any updates to our caches will eventually be propagated throughout the system, given enough time without conflicting updates.

We have also elected to have the caches save elements of one specific type. That is, for each type of item we want to cache, we will create a new cache object in our program. E.g. we could have one instance of our cache class for User data, and another for Profile pictures.

Our caches have to provide a few important operations: one has to be able to *get* cached items, *set* the value of a cached item, and *delete* cached items. Lastly, we demand that the caches support a function we call *getOrFetch*, which given a key, and a (possibly expensive) function for looking up the value, either returns the cached value if it exists, or invokes the function and returns its evaluation. These operations are specified in an interface for caches which can be seen in Listing 1.

4.2 The distributed cache system

The distributed cache system was implemented using dotnet's thread safe dictionaries. As the caches possibly contain duplicate data, each cache has to communicate upon deletion or modification of a cached value. To facilitate this, a message broker was added to the system. Originally we planned to use RabbitMQ, however we later

```
type ICache<'a> =
  member this.get (key: string) : option<'a>
  member this.set (key: string) (value: 'a) : unit
  member this.getOrFetch (key: string) (expensiveLookup: unit -> 'a) : 'a
  member this.delete (key: string) : bool
```

Listing 1: The interface used for caches in this project

```
// Invalidate (domain : string) (key : string) : InvalidationMessage

type Cache<'a> (domain: string) =
  let dict = ConcurrentDictionary()
  member this.delete (key: string) : unit =
    dict.TryDelete key
    messagebroker.publish (Invalidate domain key)
```

Listing 2: Deletion function for distributed cache. When deleting a cached item from a cache, a message is sent to all other caches in the same domain to also remove this item.

opted for Redis' built-in message broker, which allowed us to have one less component in the system. The message broker is used to facilitate a publish-subscribe pattern. In Listing 2 an example of this communication is given, where upon removal of a key from a cache, it also sends an invalidation message to any other caches in the same domain. By default, these messages are only used to flush stale data from caches, however another benefit with our messaging model is that it allows us to model dependencies between cached items.

In the distributed cache system, each cache is backed by its own thread-safe dictionary, which has to specify its domain in order to find the correct channel to subscribe to.

4.3 The shared cache system

The shared cache system was implemented on top of Redis [5]. As the Redis instance is shared among all nodes of the web service, we can be certain that the cache does not introduce data duplication. Each cache object on a Web node shares the same connection to the Redis instance, and differs only in a prefix added to the saved keys.

A problem introduced by the shared cache system that we did not have to consider in the distributed case is serialization. Since the shared cache uses the separate entity Redis, we could not simply store the cached items as regular F# values, but rather had to serialize and deserialize them upon write and read operations, respectively. For this purpose we used the JSON serialization library *JSON.Net* by Newtonsoft [12].

4.4 Testing and evaluation

In order to test the performance of the cache systems, we have developed a simple benchmark system. This system consists of an example distributed Web service exposing a simple CRUD (Create, Read, Update, Delete) API. This is accessed through a reverse-proxy which enables load balancing with sticky sessions. Finally performance data from each cache request is logged to a log aggregator. A schematic overview of the testing framework can be seen in Figure 4.1.

The Web service is based on an example domain with data types for people, pets, and families (which contain references to both people and pets). The idea is that through simulated user behavior in this example domain we can explore different scenarios in a systematic way.

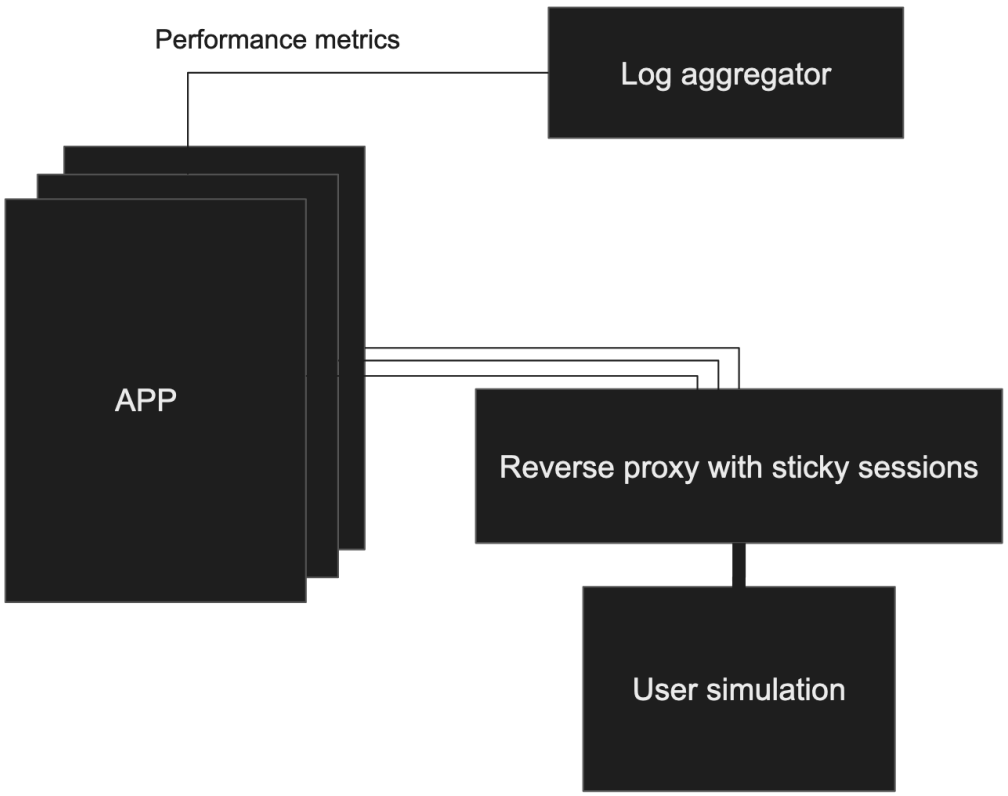


Figure 4.1: A schematic view of the testing framework

4.4.1 System parameters

There are four different cache systems we are comparing: a system may be distributed or shared, and it may or may not feature data prefetching. Prefetching concerns whether we load some, or even all, relevant items into a cache on startup. Due to lack of time, prefetching upon request was not explored. There is some interesting interaction between whether we prefetch or not, and whether the cache

is shared or distributed. As an example it raises the question of where to put our *source of truth*, that is, where we expect the correct version of each item to be. If we prefetch a domain, we can assume that each item in that domain is in the cache. It could then be beneficial for write-latency to write to the cache first, only to then update the database. Always having our most up-to-date version of each item in the cache, and only saving to the database to keep data persistence. However, this would not be without drawbacks. As mentioned earlier, both the distributed and shared caches provide only eventual consistency, and as such we would have to take care to ensure the data would not cause to large problems from possible data races.

Another very important factor in how the system will perform is how distributed the application is. That is, how many nodes are running the application. This impacts how many instances of the cache are created in the distributed case, and thus how much data-duplication is necessary.

4.4.2 User simulation

The project is concerned with what factors to consider when choosing a cache system. One of the most important factors is request patterns, and by extension, user behavior. As such a number of features of user behavior are considered. These can be seen below and were chosen as they were foreseen to have a significant impact upon cache performance.

- Read-write ratio. The percentage of requests which just read a value as opposed to editing or removing it.
- Size of the shared domain: Amount of data which every user is likely to request. A real life example could be data related to the front page of a website.
- Size of the non-shared domain: The non-shared domain is consists of all data which is not commonly requested. In a typical use case this would be far larger than the shared domain. A real life example would be all data related only to specific users or projects.
- Probability of a request going to the shared vs non-shared domain. The percentage of requests which should go to the shared vs non-shared domain. This is used to model whether most users are similar to one another, i.e. request mostly the shared domain, or if they are very disparate, i.e. request mostly from the much larger non-shared domain.

The user simulation is done through the help of a tool called Locust [13]. Which allows for the definition of example users. In Locust each user is defined through tasks, e.g. web requests, which they can perform. When running a benchmark with Locust, a number of users are created, each of which will pick random tasks to perform for an amount of time. These tasks can be weighted, such that some tasks are more likely than others. An example Locust user specification can be seen in Listing 3.

```
from locust import HttpUser, task
from random import randint

nonSharedDomainSize = 10000
sharedDomainSize = 1500
percentageShared = 50
weightDelete = 20
weightRead = 80

def getKey():
  isShared = randint(1,100) <= percentageShared
    if isShared:
      return randint(sharedDomainSize + 1, sharedDomainSize + nonSharedDomainSize)
    else:
      return randint (1, sharedDomainSize)

class User(HttpUser):
  @task(weightRead)
  def get(self):
    n = getKey()
    self.client.get(f"/cache/{n}")

  @task(weightDelete)
  def remove(self):
    n = getKey()
    self.client.delete(f"/cache/{n}")
```

Listing 3: An example Locust file with non shared domain size 10000, shared domain size 1500, 50% requests to shared domain, and 80% read ratio.

5

Results and Discussion

The following chapter covers the results of testing and discusses their implications. The results will be presented in the form of several graphs, all following the same pattern. For an example graph see Figure 5.3.

- *cache hit* denotes whether a requested element was present in the cache or not. Consequently *average on hit* means average within all hits, and *average on miss* means average within all non-hits.
- *cache response* denotes the time in which a cache determines whether it has a requested element or not. Thus *average cache response on miss* would be the average time before a cache determines that it does not have the requested element.
- *lookup* denotes the time used to fetch a missing element from the slow memory. In testing the slow memory was set to always return in 10 ms. Times higher than that are caused by the slow down induced by the cache.
- *total response* denotes the full time a cache request takes to handle. This is typically just the *cache response* + *lookup*.

5.1 Latency vs hit rate

Throughout all testing done the most clear result is the respective benefits of the two models. Namely hit rate vs latency. Consider Figure 5.4: Average cache response, on hit, miss and in general is an order of magnitude faster for the distributed cache system. This is to be expected: after all, memory access is far faster than network access. Going on to consider the average total response on hit we see the same thing. The average total response on miss is more interesting. Recall that our simulated slow memory is set to a latency of 10 ms, any additional time is incurred due to the cache. Again the distributed cache beats the shared system by a far margin. Which brings us to the most interesting bar, the average total response. Notice that though the distributed cache system had an on average lower latency in every case, the average total response is almost equal. This is due to the hit rate difference. In that test the shared cache had a hit rate of 59%, compared to the distributed caches 43%.

This general pattern stays true throughout all our testing, and most of our factors

had a larger impact on our hit rate rather than the latency times.

5.2 Impact of prefetching

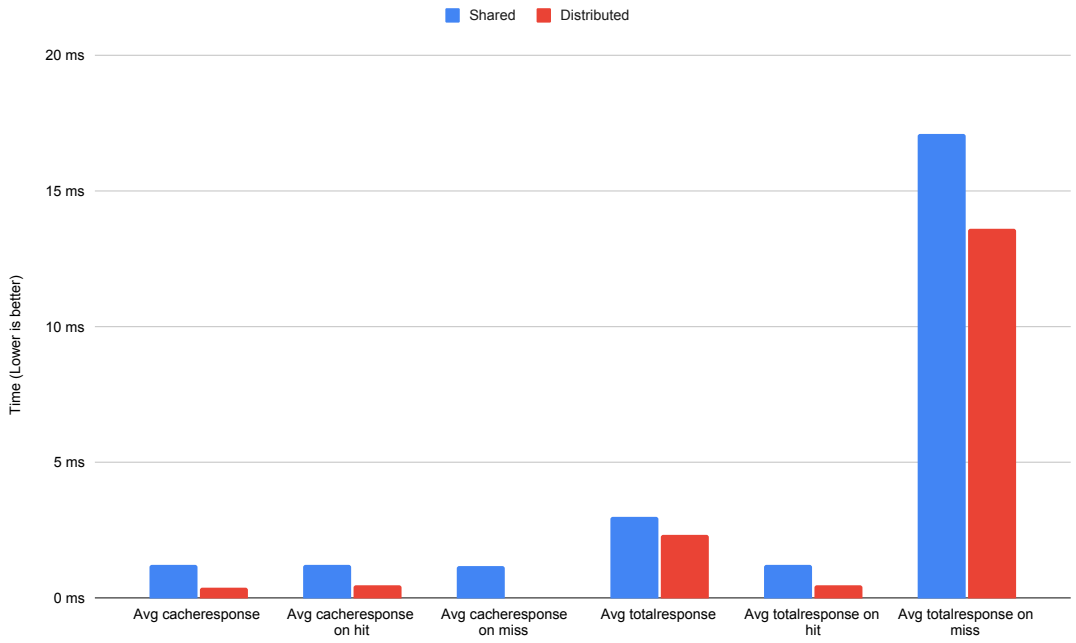
In our tests, prefetching had a consistent positive impact upon the hit rate, although the size of this impact differed slightly based on the read ratio of the test case. Consider the difference between Figure 5.4 and Figure 5.1: just turning on prefetching causes a significant increase in hit rate for both the shared and distributed caches (from 59% to 89% for shared, and from 43% to 86% for the distributed). Importantly, this almost completely removes the difference in hit rate between the two systems. However, as we can see if we consider the difference between Figure 5.1 and Figure 5.2, a lower read ratio causes this hit rate improvement to decrease.

5.3 Impact of number of nodes

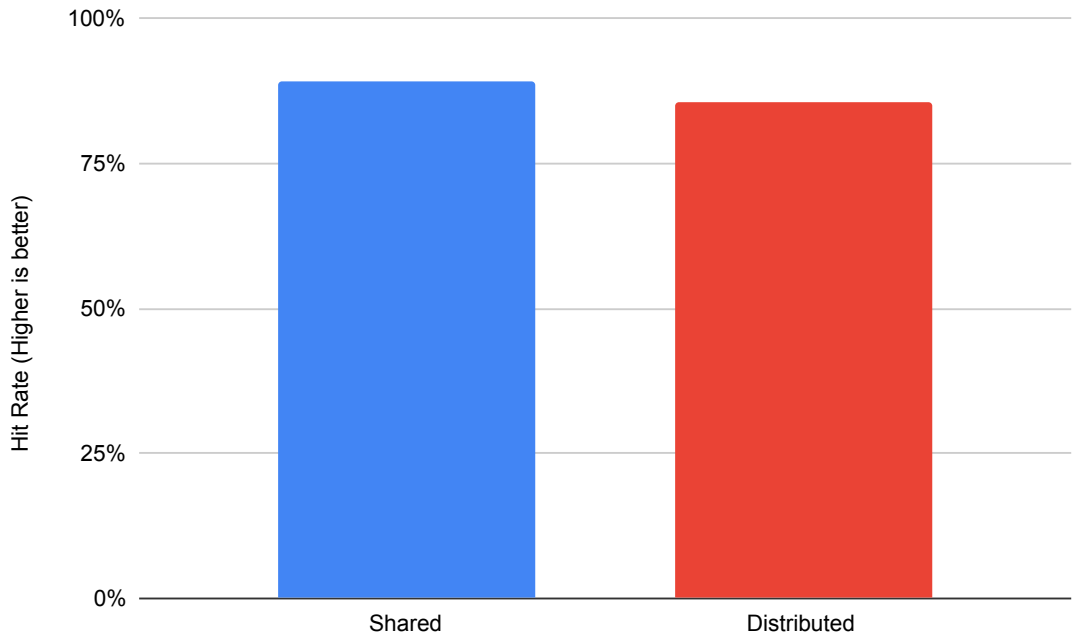
Perhaps the most significant factor in this comparison is the degree of distribution. The number of nodes running the Web service has a direct impact on the amount of data duplication needed for the distributed cache system. Another feature highlighted by the degree of distribution is the possibility for different nodes to benefit from each others' misses in the shared cache system. Consider the difference between Figure 5.4 and Figure 5.3. Increasing the amount of nodes from three to twelve more than halves the hit rate for the distributed cache, going from 43% to 20%. This also causes the difference in total response to go from being about equal to the shared cache performing more than twice as fast as the distributed system.

5.4 Impact of shared domain

The most subtle impact came from changes in domain sizes and ratio. Consider Figure 5.5 and Figure 5.6: Changing from 30% to 60% of requests going to the shared domain results predictably in increased hit rate. Though this hit rate increase is not equal in both models, with the shared cache system getting a greater increase (38% to 53%) compared to the distributed system (21% to 29%).

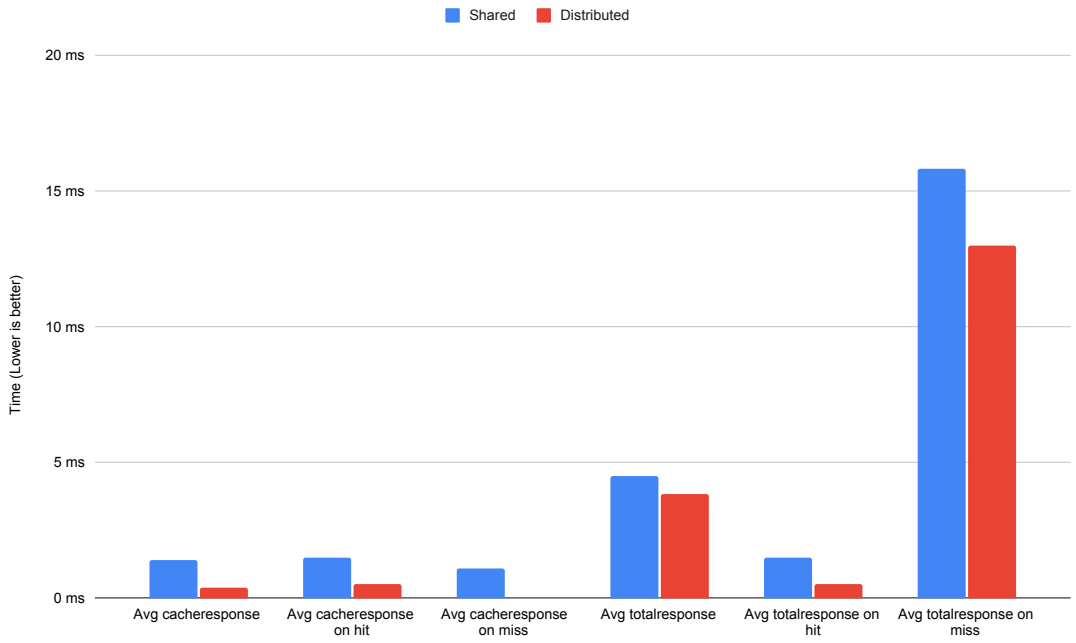


(a) Response times in ms, lower is better



(b) hit rate, higher is better

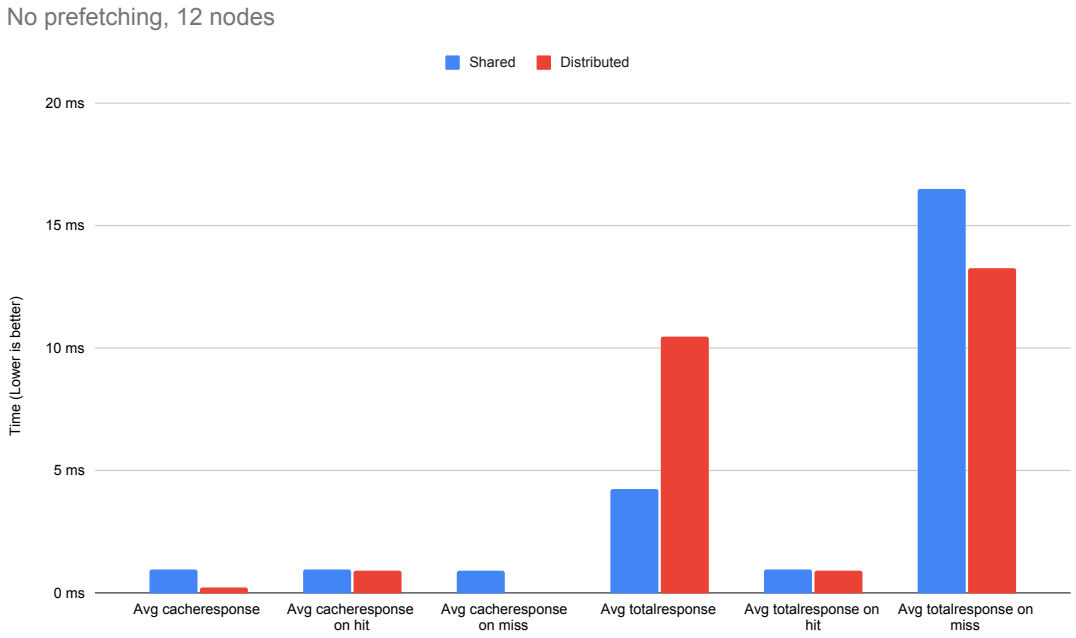
Figure 5.1: 3 nodes, with prefetching, 10000 elements non-shared domain, 1500 shared domain, 30% shared domain, 80% read ratio.



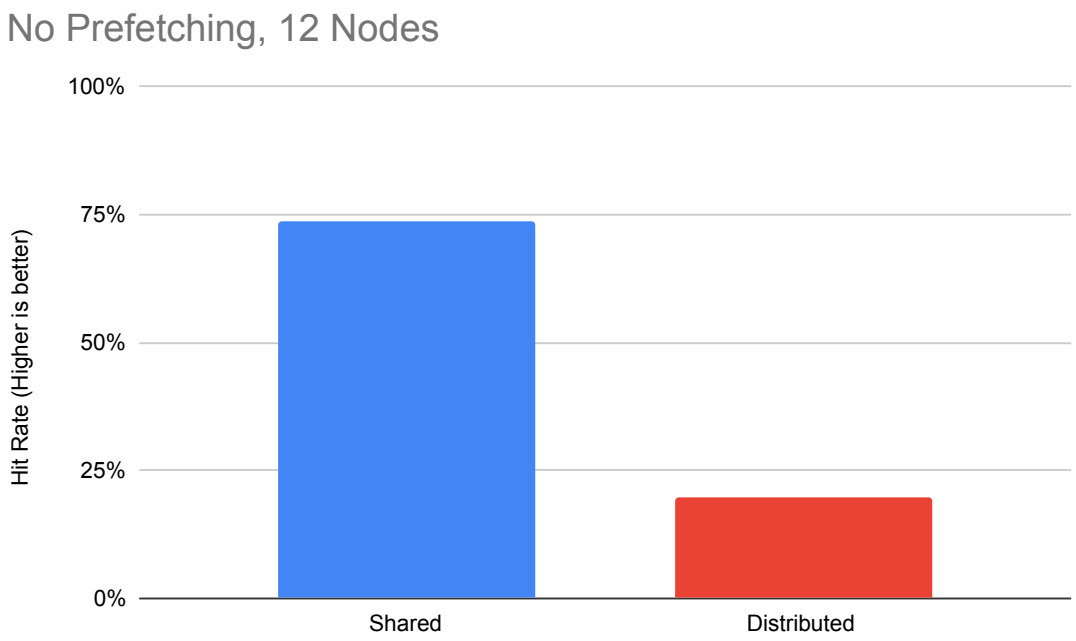
(a) Response times in ms, lower is better

(b) hit rate, higher is better

Figure 5.2: 3 nodes, with prefetching, 10000 elements non-shared domain, 1500 shared domain, 30% shared domain, 50% read ratio.

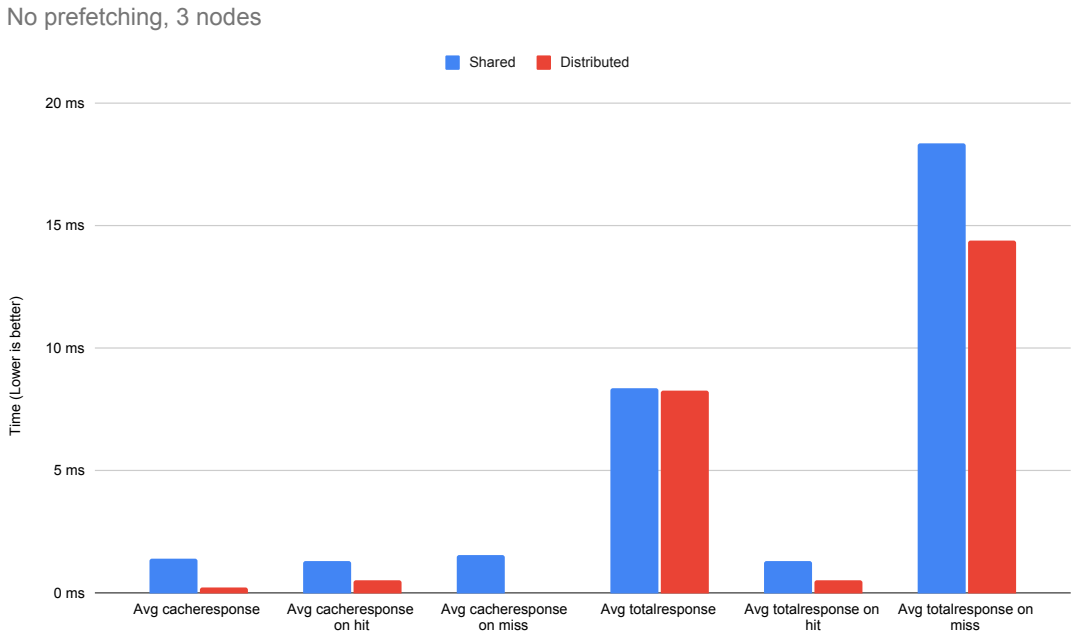


(a) Response times in ms, lower is better.

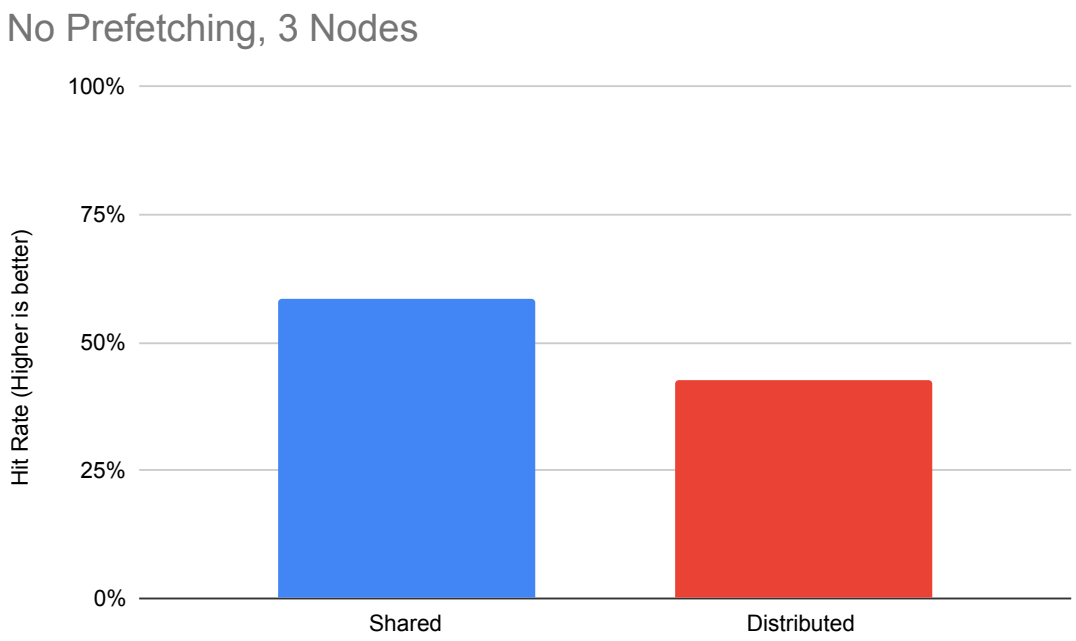


(b) hit rate, higher is better

Figure 5.3: 12 nodes, no prefetching, 10000 elements non-shared domain, 1500 shared domain, 30% shared domain, 80% read ratio.

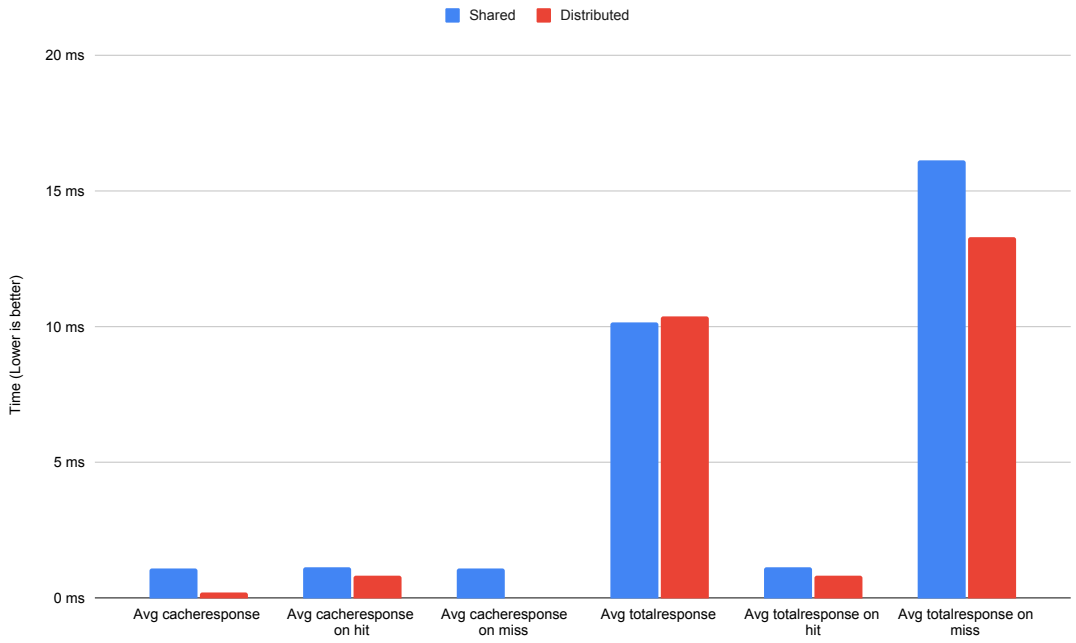


(a) Response times in ms, lower is better

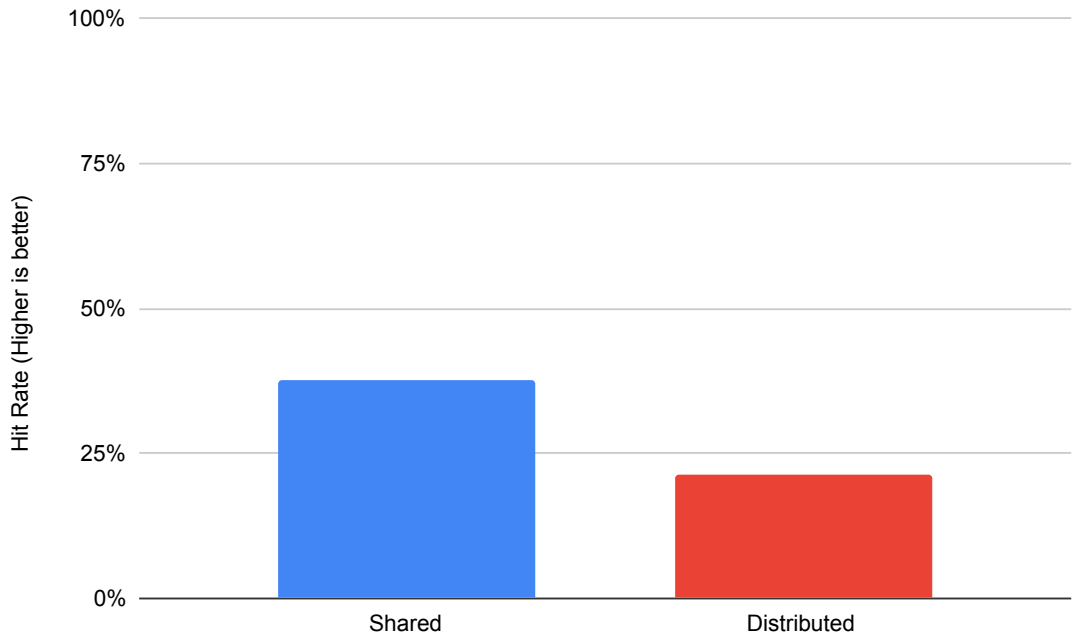


(b) hit rate, higher is better

Figure 5.4: 3 nodes, no prefetching, 10000 elements non-shared domain, 1500 shared domain, 30% shared domain, 80% read ratio.

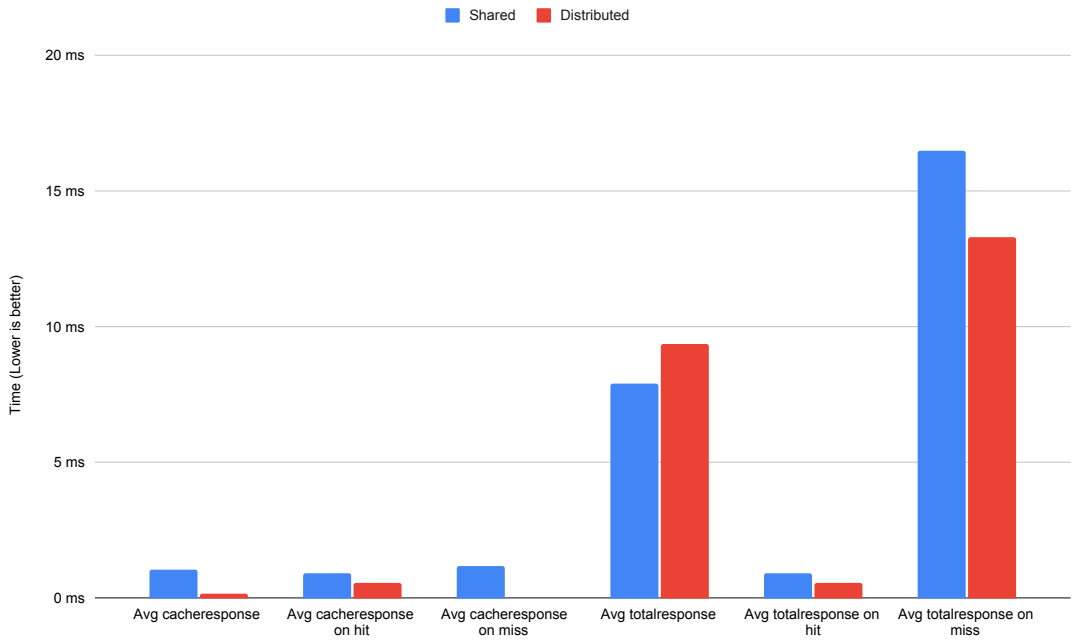


(a) Response times in ms, lower is better.

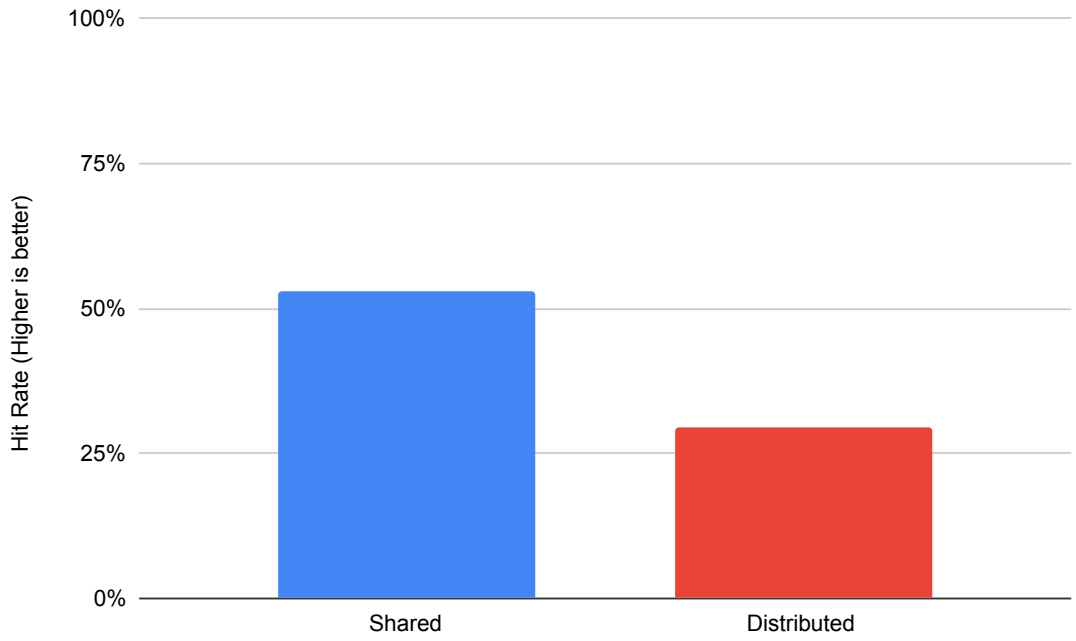


(b) hit rate, higher is better

Figure 5.5: 6 nodes, no prefetching, 50000 elements non-shared domain, 1500 shared domain, 30% shared domain, 80% read ratio.



(a) Response times in ms, lower is better



(b) hit rate, higher is better

Figure 5.6: 6 nodes, no prefetching, 50000 elements non-shared domain, 1500 shared domain, 60% shared domain, 80% read ratio.

6

Conclusion

The first goal of the project was to create a per-server cache system. The caching domain was modeled. Session consistency was chosen as the consistency model. A common interface was created for caches in this domain. Finally, two implementations of this interface were programmed. One using a per-server cache as described in the first goal and one as a reference using the state-of-the-art in memory key-value store Redis.

The second goal concerned performance testing. A test suite was created testing four variants of the caching system, under different user behaviors. A few conclusions can be drawn from these results:

- There is a clear trade-off between hit rate and latency to take into account when choosing the shared vs the distributed cache system.
- The shared cache system scales far better as you add nodes. The distributed cache system's hit rate is quickly hindered as the number of nodes was increased. Due to the fact that nodes cannot benefit from each other's cache misses.
- Prefetching at startup helps both cache systems, though not equally. In particular, it helps close the gap in hit rate between the two systems.
- A larger cache domain results in a lower hit rate for both cache systems. Though again, in the shared cache system different nodes can benefit from each other's misses, and thus the penalty is slightly lower for them.
- As users become more similar and request more frequently from some shared "common" domain, the hit rate for both cache systems goes up. However, again, the shared cache system benefits more than the distributed one.

In conclusion: there are cases where both cache systems are viable. As long as the distributed web app runs on 3 nodes or less, the systems often perform very similarly. Although the app is deployed on more nodes, the distributed cache system quickly starts to fall behind in hit rate. As this happens, the benefit of a lower latency diminishes quickly.

6.1 Future work

There are always more things to explore, and this project is no different. The project has necessarily been focused on Antura's use case. This has meant disregarding factors that could play a large part in other similar cases. The most important of these could arguably be cache-replacement and cache-eviction algorithms. This can benefit greatly from domain-specific knowledge.

Another factor that the project would have explored given more time is the impact of other prefetching methods. In testing, only prefetching of the complete domain on startup was explored. It would have been interesting to look at domain-specific prefetching and how a cache interface supporting this could have looked. In Anturas case, this could have included strategies like loading an entire project into the cache upon a cache request of a single object within it. Another thing that could be explored in more depth is different variants of the shared cache system. Redis has a depth of features which were not explored, including distribution options, which could be especially interesting in this comparison. One could also explore a hybrid system, with both shared and distributed features, which could theoretically be tailored for both common and disparate user request patterns.

Bibliography

[1] J. Wang, "A survey of web caching schemes for the internet," *Comput. Commun. Rev.*, vol. 29, no. 5, pp. 36–46, 1999. DOI: 10.1145/505696.505701. [Online]. Available: https://doi.org/10.1145/505696.505701.

[2] I. U. Din, S. Hassan, M. K. Khan, M. Guizani, O. Ghazali, and A. Habbal, "Caching in information-centric networking: Strategies, challenges, and future research directions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1443–1474, 2018. DOI: 10.1109/COMST.2017.2787609.

[3] M.-C. Lee, F.-Y. Leu, and Y.-P. Chen, "Pareto-based cache replacement for YouTube," *World Wide Web*, vol. 18, no. 6, pp. 1523–1540, 2015. DOI: 10.1007/s11280-014-0318-9.

[4] *Memcached - a distributed memory object caching system*, https://memcached.org/.

[5] *Redis*, https://redis.io/.

[6] R. Nishtala, H. Fugal, S. Grimm, *et al.*, "Scaling memcache at facebook," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[7] R. Tewari, M. Dahlin, H. Vin, and J. Kay, "Design considerations for distributed caching on the internet," in *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, 1999, pp. 273–284. DOI: 10.1109/ICDCS.1999.776529.

[8] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009. DOI: 10.1145/1435417.1435432.

[9] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, p. 51, Jun. 2002, ISSN: 01635700. DOI: 10.1145/564585.564601.

[10] M. J. Wise, "Message-brokers and communicating prolog processes," in *PARLE '92: Parallel Architectures and Languages Europe, 4th International PARLE Conference, Paris, France, June 15-18, 1992, Proceedings*, D. Etiemble and J. Syre, Eds., ser. Lecture Notes in Computer Science, vol. 605, Springer, 1992, pp. 535–549. DOI: 10.1007/3-540-55599-4_109. [Online]. Available: https://doi.org/10.1007/3-540-55599-4%5C_109.

[11] K. P. Birman and T. A. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, L. Belady, Ed., ACM, 1987, pp. 123–138. DOI: 10.1145/41457.37515. [Online]. Available: https://doi.org/10.1145/41457.37515.

[12] *Json.net - newtonsoft*, https://www.newtonsoft.com/json.
[13] *Locust - a modern load testing framework*, https://locust.io/.