Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Predictive Caching using Machine Learning in IoT Scenarios

Lars Wüstrich

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Predictive Caching using Machine Learning in IoT Scenarios

Vorausschauendes Cachen mittels Maschinellem Lernen in IoT Szenarien

Author:	Lars Wüstrich
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Stefan Liebald, M. Sc.
	Dr. Marc-Oliver Pahl
Date:	April 15, 2019

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, April 15, 2019

Location, Date

Signature

Abstract

Communication in IoT networks follows a certain pattern caused by the prevalence of machine-to-machine interaction. The structure in the observed communication can be used to optimize caching mechanisms that are employed to reduce the latency for information accesses. Machine learning offers promising solutions to detect these access patterns and make more informed decisions than traditional caching algorithms. By introducing two new cache replacement strategies it is demonstrated how machine learning can optimize caching in the context of IoT. The first strategy makes use of *K-Modes* clustering to process the semantic features introduced by information centric networks to perform content aware caching. By using these features, the strategy is capable of finding popular areas, data types or devices and predicts which item is least likely to be accessed in the future. The second approach proposes an adaption of reinforcement learning for caching in IoT networks. In this strategy the algorithm is trained by using a utility function that punishes having to refetch an item. By implementing the clustering strategy in the ICARUS framework it is shown that this approach leads to competitive results when compared to standard strategies.

Zusammenfassung

Datenflüsse in Internet of Things (IoT) Netzwerken werden hauptsächlich von Maschine-zu-Maschine Kommunikation verursacht. Dadurch ergibt sich eine Struktur im Datenverkehr, welche beobachtet werden kann. Diese Struktur kann genutzt werden, um Caching-Algorithmen zu optimieren, welche unter anderem dazu eingesetzt werden die Latenz solcher Zugriffe zu verringern. Maschinelles Lernen bietet verschiedene Möglichkeiten solche Strukturen zu erkennen, um die Caching-Entscheidungen traditioneller Strategien zu optimieren. Durch die Adaption zweier Machine-learning Algorithmen wird demonstriert wie Caching im Kontext des IoT durch Maschinelles Lernen verbessert werden kann. Die erste Strategie nutzt *K-Modes* Clustering, um semantische Attribute zu verarbeiten und *content aware* Caching zu betreiben. Das Ergebnis des Clusterings wird genutzt, um beliebte Inhalte im Cache zu bestimmen und vorherzusagen welche Daten vom Cache entfernt werden können. Die zweite Methode nutzt *Reinforcement Learning*, um eine optimale Cache-Ersetzungsstrategie zu erlernen. Dazu wird eine Utility-Funktion verwendet, welche Cache-miss bestraft und belohnt wenn eine Anfrage aus dem Cache beantwortet werden kann. Eine Implementierung der Clustering-Strategie im ICARUS-Framework zeigt, dass die Strategie vergleichbare Werte zu traditionellen Ansätzen liefert.

Acknowledgements

I would like to express my deepest appreciation to my advisors Stefan Liebald M.Sc., and Dr. Marc-Oliver Pahl for the continuous support. Their advice helped to steer this work into the right direction when needed.

I am also grateful to Prof. Dr.-Ing. Georg Carle for supervising my work and providing me with the opportunity to experience the scientific work at his research group.

Special gratitude goes to my family for their continuous support throughout this work and over the course of my studies.

I cannot begin to express my thanks to Anna K. Baumeister and Benedikt R. Graswald who helped me during the preparation of this thesis with their unparalleled support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years it has become more and more common to connect sensors and other devices to the Internet. This has enabled users to access information about the connected devices online and issue commands to these devices. This type of usage of the Internet is also referred to as the 'Internet of Things' (IoT) [39]. By connecting sensors to take measurements of the real world, users gain insight into processes and are capable of making better decisions based on the available data. In such a scenario, there are many sources that generate data and fewer sinks where the data is processed and the results published. Like all computer systems, the IoT is also affected by the *von Neumann bottleneck* which is a result of the different speeds of microprocessors and data storage. In contrast to single processor systems, the information is not stored in local storage, but has to be requested and received from different end points in the network. One solution to this issue is the concept of caching. With caching, the requested data is copied into high speed storage ahead of processing time to speed up the execution time. In distributed networks the equivalent to high speed storage is to keep the data at a node in the network that can be accessed quickly by the requesting device. This concept is also used in the context of the Internet, where content from the web is stored on proxies or a local cache in order to reduce latencies. Recently, some efforts have also been made to implement caching in the context of the IoT. In contrast to the traditional Internet, devices, applications and users are connected with minimal human interaction. This leads to patterns that can be used to implement efficient caching mechanisms.

This thesis explores possibilities of how caching mechanisms used in the context of IoT can be enhanced by machine learning. It is therefore necessary to evaluate what types of content are exchanged and which parts of it can be cached. Since many applications that evaluate data generated from devices in the IoT rely on a certain 'freshness' to

make decisions, it is also relevant to determine for how long the recorded data can be cached until it is considered stale. To assess how machine learning can actually improve caching it is necessary to analyze how caching strategies make decisions. This knowledge can then be used to determine kind of data is needed and how parameters can be used to train a machine learned model that improves the standard caching strategies. The final challenge that needs to be tackled is how the new model can stay accurate, even when the structure, setup or communication patterns of the IoT system change. All of these issues need to be considered to answer the question:

How can machine learning be used to improve caching in the context of the IoT?

In order to answer this question, background on the topics of the IoT, caching and machine learning are introduced in Chapter 2. Chapter 3 discusses related research with proposed solutions to similar approaches and problems. These include approaches that are specifically developed for the IoT or more general approaches that makes use of machine learning mechanisms. In Chapter 4 an answer to the stated question is given. It outlines how a machine learned approach to caching in IoT networks could be implemented. This implementation is then presented in Chapter 5. Chapter 6 then evaluates this approach by comparing it to already existing solutions. Therein, different network topologies and workloads are simulated and the performance of the proposed algorithms is measured with regards to different metrics. Finally a conclusion is drawn and future work is discussed in Chapter 7.

Chapter 2

Analysis

In order to answer how machine learning can improve caching mechanisms in the context of the IoT, it is necessary to introduce some background knowledge and challenges arising from these technologies. There are three fields that need to be discussed before a solution can be defined. These are

- the Internet of Things,
- Caching, and
- Machine Learning.

In the following sections these topics will be briefly introduced in order to give an overview of the existing technologies and approaches. The concepts of IoT are explained in Section 2.1, followed by an introduction to caching in Section 2.2. In the third part of the Chapter some basic machine learning concepts are explained. Finally, the requirements that a possible solution needs to fulfill are given.

2.1 The Internet of Things

A new trend that is shaping the Internet is the Internet of Things (IoT). In the IoT everyday objects are connected to the Internet with the intention of adding more value to existing services. These devices range from thermometers to refrigerators but can also be robots in industrial complexes. Since the IoT already has many applications and is still a growing market with a predicted turnover of about $7.1 trillion by 2020 [66], more research needs to be done in order to use this new technology efficiently. This is also illustrated in the August 2018 edition of Gartners IT Hype cycle, which

predicts that it will take another five to ten years until IoT systems are well-engineered enough to be used reliably in production environments [1]. Until then many technical and sociological challenges have to be considered. Some sociological challenges include privacy concerns when dealing with processing personal data. Technical challenges largely deal with setting up large scale interconnected infrastructures and integrating new types of devices into existing architectures [9]. This also includes considerations about the locations, where data is kept and stored. In the following, key concepts of the IoT are introduced and explained.

The reduction in cost and size of electronic devices made it possible to connect embedded systems to the Internet and to introduce an era of ubiquitous computing. In this era the environment is measured by sensors and influenced by actuators and other digitally influenced devices [28]. By using synergies of different sensors and other sources from the internet, the IoT makes it possible to improve the efficiency and effectiveness of the connected devices [39]. In order to integrate these new types of devices and connect them to each other, some adaptations to the currently used protocols that make up the Internet need to be done. This is necessary because instead of connecting hosts to other hosts that offer services, the concepts used in IoT take a more data centric approach. Instead of addressing specific hosts, data is accessed by explicitly addressing it. This is necessary since the connected devices mainly harvest information of their environment and make decisions based on the gathered data. In order to make the integration of those devices possible, IoT systems are logically structured into three layers [28][66][9] (also illustrated in Figure 2.1):

1. The **Device Layer** or things-oriented layer which consists of sensors, actuators and every device that should be connected to the IoT.
2. The **Connecting Layer** or Internet-oriented layer that connects the devices in the device layer to the Internet, IoT cloud layer and each other.
3. The **IoT cloud layer** or semantic-oriented layer that uses the gathered information to make decisions and act in order to achieve the best outcome for the system and its users.

The device layer is made up of devices with various purposes. This includes actuators, sensors and everyday devices like refrigerators and TVs, making it a heterogeneous layer.

Since the devices are not as homogeneous as the hosts on the traditional Internet, a new way of addressing the devices needs to be found to address this new diversity. One approach that tries to achieve this is explained in the following subsection.

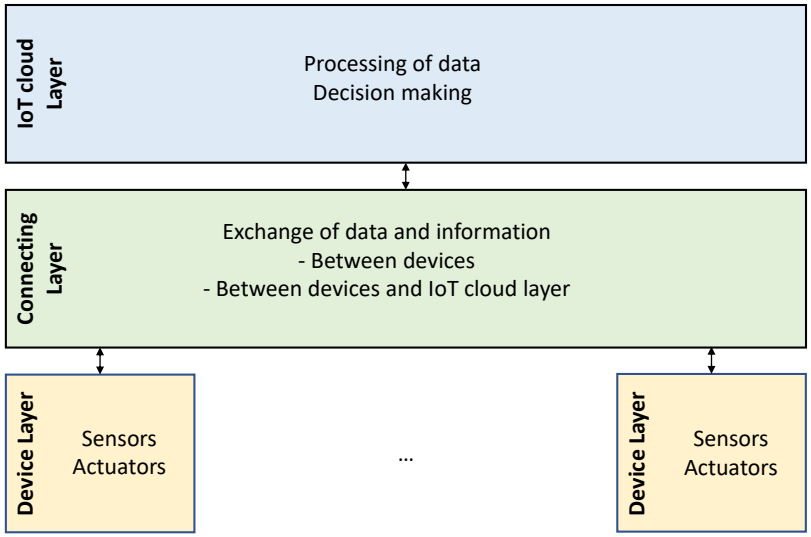


Figure 2.1: Logical structure of IoT systems

2.1.1 Information Centric Networks

In order to make it easier to address and connect these devices on the device layer, new concepts are developed to make them addressable on the connecting layer. This is done by generalizing the current addressing scheme used on layer 3 of the ISO/OSI model where hosts are addressed by IP addresses. Concepts like Information Centric Networks (ICN)[67] or the distributed smart space orchestration service (DS2OS) [50] propose solutions that make it possible to address the different devices by service and offered data without having to know the actual address of each device. The data centric approach of ICN is also referred to as Named Data Networking (NDN) [71]. By using this addressing scheme, applications on the IoT-cloud layer can use a more straightforward approach to gather data from specific devices. Furthermore, commands to single devices or groups of devices can be issued in a simple manner.

Even though systems that are set up in the context of the IoT follow the logical setup shown in Figure 2.1, the physical structure is often a lot more complex. In most cases there are many data sources (e.g. sensors) present while few data sinks collect and process the data. In this scenario when data from a specific sensor is needed, the request is typically sent to the data sink which then forwards the request to the sensor.

In NDNs, every node that forwards a request to a sensor or data sink has a cache that is capable of storing some of the information. This makes it also possible to cache data in order to answer similar requests immediately without having to reissue them. If the network is too big, multiple dedicated data sinks are often deployed to process the data. Each sink is then responsible to either handle requests issued to a certain part of the system or to a specific type of device in the network.

Nodes in NDNs do not maintain single dedicated caches at specific data sinks, but instead offer the possibility to cache data at every node that is part of the network. These caches are implemented by a special architecture which every node that is part of the ICN is required to maintain. The architecture consists of three data structures and a forwarding strategy [29][71]:

1. The **Content Store** (CS) where data is cached.
2. The **Pending Interest Table** (PIT), a table that stores all forwarded, but unanswered interests.
3. The **Forwarding Information Base** (FIB), a table that contains the name prefixes of data sources which the node can use to forward an incoming interest if it cannot be answered by the node itself.

Whenever a device receives a request, the node first checks if the request can be satisfied with the data that is stored in the CS. If so, the node replies with the corresponding information on the interface from which it has received the request. In case the requested data is not present in the CS, the node performs a lookup in the PIT. If there already is a pending interest for the requested data, the node appends the interface on which it has received the request to the entry. If there is no entry in the PIT for the requested data, a new entry with the name of the requested data is created, along with the interface from which the request was received. If a new entry in the PIT has to be created, the node forwards the request to the corresponding nodes listed in the FIB. In case the FIB has no corresponding entries, the request is dropped [71][7]. The process of forwarding data requests is illustrated in the upper half of Figure 2.2.

When handling requested data that passes through the node, it checks if an interest for the information is stored in the PIT. Otherwise, the data is discarded. In case there is an interest in the PIT, the data is stored in the CS and forwarded to all nodes that are marked as interested in the data. This process can also be seen in the lower half of Figure 2.2.

Since most devices are very limited in resources, the cache on each node is also limited and can only hold few items. In order to use this cache efficiently, there is a need for

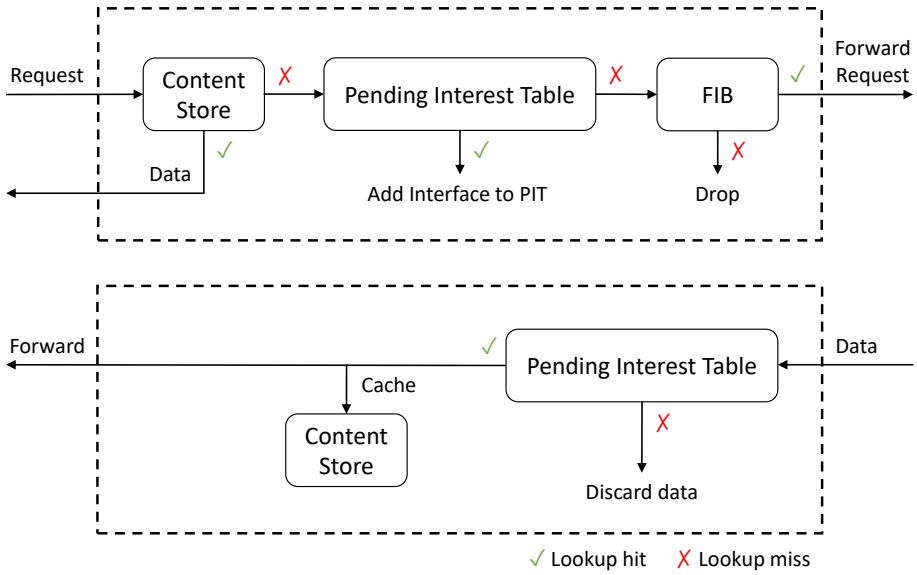


Figure 2.2: Request handling in NDNs illustrated in the top part, data handling in the lower part. Adapted from [71]

mechanisms that manage it effectively. Before these mechanisms are discussed in detail another key characteristic of the IoT is introduced.

2.1.2 Patterns in IoT

IoT networks often have certain characteristics that do not occur in the Internet. One main characteristic is that IoT networks consist of many data sources and few data sinks. Traffic is normally directed towards data sinks where the gathered information is processed. Information from data sources is often sent in small messages containing updates measured by the connected devices. The result is an influx of small packets that are sent towards the data sinks while few messages leave those sinks. Another characteristic of communication of IoT systems is that devices exchange information autonomously without human interaction. This traffic is also referred to as Machine-to-Machine (M2M) communication. Since this communication is based on algorithms deciding when to send data, a general pattern of message exchanges emerges in IoT networks which is disrupted by human interaction. This pattern can be useful when developing new mechanisms that optimize the usage of resources in the network as some of the traffic is predictable. The first pattern that can be identified is the periodicity of certain messages [60]. In IoT networks messages often appear periodically, e.g. a message from device A to device B is sent every 30 seconds. This holds for both requesting

services and services that push their information to the data sink. By anticipating these periodic messages and requests it is possible for a data sink to prefetch data in order to reduce the latency of the service. Another pattern that can be found are requests that are related to each other. This means whenever a certain message m_1 is observed, m_2 is always observed after some time. This makes it possible to predict future messages even when no periodicity is given. Finally, if IoT cloud layer introduces some semantics, these can also be used to predict patterns, e.g. requests to a certain type of devices. This makes it possible to predict the popularity of some devices and to make content aware decisions as more information of the message content is available.

2.1.3 Summary

Overall the IoT is an emerging technology that opens up a lot of possibilities for both home users and industrial applications when developed correctly. IoT networks connect everyday objects to the Internet, adding many data sources to the distributed network which makes up the Internet. Most of these devices have limited resources that need to be used efficiently. In many cases, devices are also connected via multiple hops to each other since connecting every device is inefficient in large networks. These links between data generator and data consumer can be slow or fail, leading to potential unavailability of services and high latencies. A solution to these problems can be the use of caches throughout the network as introduced in NDN. By storing data in places that can be accessed quickly and reliably, the uptime of services can improve and latencies can be reduced. Even when data cannot be fetched from the source, a copy of the latest data from the cache can still be served. Since IoT networks can benefit from the use of caches, the concept is introduced and explained in the next section.

2.2 Caching

As mentioned in Section 2.1, IoT networks can benefit from the use of caches as introduced in NDN. Caching is a mechanism that is needed to overcome the issue of having slow storage while having fast computation speeds. Since caching is a concept used to improve different parts of computer systems, the mechanism and challenges arising from it are introduced in the following.

2.2.1 The memory hierarchy

The concept of caching was introduced in order to mitigate the processor-memory performance gap which is also referred to as the *von Neumann bottleneck*. The main cause for this gap is that the semiconductor industry has concentrated on improving the speeds of microprocessors, while the memory focused industry has rapidly increased the capacity of storage instead [19]. One way to address the bottleneck is the introduction of multi-level memory hierarchies [44]. Each tier in the hierarchy reduces in size and increases in speed compared to the one before. At the bottom of the hierarchy is the slowest type of memory which usually consists of remote secondary storage and can hold the most data. Files that are located on this hierarchy level are usually stored on web servers, distributed file systems or other remote storage. Accessing information on this level is usually slow. Since it is on the lowest tier of the hierarchy it takes up more time than accessing information that is stored in higher tiers. Going up a level in the hierarchy, is the local secondary storage, which is normally the local hard drive or solid state disk. Even though data that is stored in memory of this level of the hierarchy is located on the same machine from which the data is requested, accessing it is still slower than if it was stored in the main memory of the system [35]. The main memory is the next tier in the hierarchy and often limited to a few gigabytes which limits the amount of data that applications can keep in it. This fosters the need for efficient caching mechanisms such that the benefit of high speed memory is maximized. Depending on the system, one or more layers of cache can exist, each offering less memory at higher speed. Similar to the main memory, caching is needed to efficiently use this level of hierarchy. Finally, on top of the hierarchy are the CPU registers that are used by the microprocessor to execute its task. Data that needs to be processed is loaded from lower tiers of the hierarchy into these registers [69]. An example of the memory hierarchy can also be seen in Figure 2.3.

Caching is performed by copying data that is relevant to the program execution from lower into higher layers of the hierarchy for faster access. The caching mechanism can be logically separated into two parts, a *caching decision strategy* and a *replacement strategy* [29]. The *caching decision strategy* makes decisions on what parts of data are relevant and are therefore loaded into the cache. This strategy often depends on information about the data that was accessed before. Based on previous data accesses, the strategy tries to predict which data may be needed in the future and loads the information into the cache in order to quicken the execution of the program. This process is also called *prefetching*. If the cache is already full and new data is supposed to be loaded into it, the *replacement strategy* comes into play. Its purpose is to decide which data in cache is supposed to be replaced by the new data. By removing data that is least likely

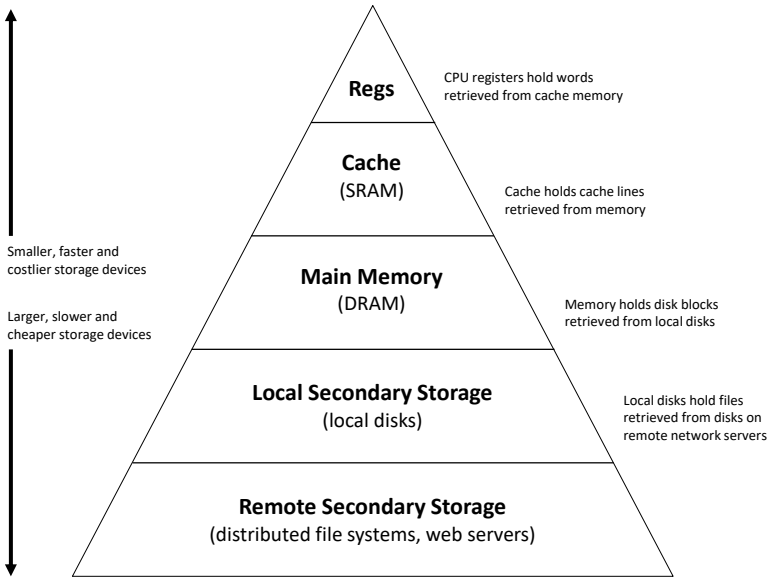


Figure 2.3: Example of a memory hierarchy, adapted from [69]

to be used again in the future, the cache replacement mechanism tries to minimize the number of replacement operations. Similar to the *caching decision strategy*, this prediction is based on the most recent accesses. There are multiple approaches on how the predictions are made, discussed in sections 2.5.1 and 2.5.3.

Caching is a way of dealing with the processor-memory gap by copying relevant data into fast storage to provide the microprocessor with quick access to needed data. The von Neumann bottleneck is not limited to the execution of programs on a local machine, but applies to all situations where data needs to be fetched from slow storage. Another example of a scenario where caching is needed is the Internet, where requests for data are routed around the world. Instead of having a single fast storage that keeps the data present, the information is kept at different places of the network where fast access can be provided. Some adaptions were made in order to maximize the benefit of caching in this new environment. These are explained in the next section.

2.2.2 Web caches

In the context of the Internet and web applications, caching is needed since all of the requested data is located in remote secondary storage. In order to provide faster

access to files, reduce latency and increase availability of content, caches are deployed at different locations of the network. Another effect of caching in this context is the reduction of bandwidth usage, since instead of downloading the same file again, the cached copy can be used instead. Caches increase the availability of a web service since in case the original service fails, the cache can be used as a backup that serves the content requests. Caches are deployed at a different place in the network depending on the goal that should be achieved [11]. A list of the properties a (web)cache should have is also given in [64]:

- *Fast access*: The cache should reduce the latency and provide faster access to users that are requesting the content.
- *Robustness*: By adding a second source for content, a web cache should increase the robustness of a service in case the original source has an outage. In addition to that, a failing cache should not affect the rest of the system.
- *Transparency*: Users should not realize that the content is served by a cache. The only effect that should be noticeable are lower latencies.
- *Scalability*: The scheme used to deploy the cache should be scalable so that it can also be used at large scale.
- *Efficiency*: The introduced caching scheme should be efficient with regard to two aspect:
  1. The cache should not introduce too much overhead
  2. The cache should not lead to under-utilization of critical resources
- *Adaptivity*: In order to operate optimally, the cache should adapt to changes in user demands as well as changes in the network.

In order to evaluate how much the network benefits from a cache, it is evaluated with regards to [64]

- speed,
- scalability and
- reliability.

In order to achieve better performance in one or more of these metrics, different architectures have been proposed and developed. Those are described extensively in [11], where also some use cases are described. Generally speaking, caches can be positioned at three different areas in the network depending on which metric should be maximized.

The first type of positioning is called *consumer-oriented* positioning. In this case, the positioning is done with the goal of optimizing the latency. When latency is of less importance but availability and scalability are of relevance, the *provider-oriented* placement is chosen. In this case, the cache is set up close to the content source. The third option is to place the cache at a strategic point between the user and content provider, offering a trade-off between the two earlier introduced options. One example for such a strategic point could be a common Internet Service Provider (ISP) that is shared by the user and provider for sharing files or other content. In this case the latency for the user would be reduced while offering the option for the host to store a copy of the files in a cache that is also accessed by other users.

2.3 Challenges for caching

Apart from all the benefits that a caching system can provide, there are also some disadvantages which are discussed in the following. One of the major issues that arise with the use of a cache system is the *stale data problem* [47] which affects the *consistency*. This problem occurs when the cached data differs from the data at the source. This difference could be introduced by some change that is made to the original file, e.g. when the file is updated at the source. In this case, the data in the cache needs to be updated such that the correct file is also delivered when requested from the cache. In order to overcome this problem a mechanism needs to be implemented that checks if the cached data matches the original.

Even though the purpose of a cache is the reduction of latency, this is only the case when a *cache hit* occurs which means that the requested data is already in cache. If the requested data is not in cache this is called a *cache miss*. In case of a cache miss, the cache first must fetch the data from the source before it can fulfill the request. This additional request causes delays that slow down the service. Therefore it is desirable that both the cache decision and cache replacement strategies ensure a high cache hit rate in order to maximize the benefit. Another problem that could occur is that the cache represents a bottleneck that slows down the operation of the whole system. In this case a failure of the cache can have an impact on the whole service and may even cause major disruptions of the operations [64]. Another challenge that arises is that caches often store items of different size which needs to be considered by the strategies that manage the cache [36]. This includes the decision on if a lot of small items or less, but larger items should be cached to improve the performance of the overall system.

In order to overcome some of these issues, different setups have been developed on how to deploy the caching system. To avoid introducing a bottleneck into the system, most strategies rely on multiple caches that collaborate in order to achieve the desired goal [65]. One example of such a setup is the *harvest cache* in which multiple caches are set up in a hierarchical order. Instead of requesting the data from the source, caches will request the information from higher tier caches and cache neighbors which will reply with the data in case there is a cache miss or will forward the request to the next tier or to the actual data source. Finally, the cache answers with the result that was received fastest [20]. The *cachemesh* system also uses a distributed approach to cache content. In this setup, single caches are assigned to certain content that they are responsible for. When a request is received, it is then forwarded to the corresponding cache which serves the request. In contrary to these methods there are also setups where caches adapt to the popularity of certain content. One of these methods is therefore also called *adaptive web caching* [70]. By forming overlapping but independent groups holding data that is spread to other groups via multicasting, popular content is stored on more caches while less popular content is stored in fewer groups.

In order to overcome the additional challenges that the IoT imposes on caching systems, current schemes need to be adapted and new ones developed. These are introduced in the following section.

2.4 Caching in IoT

Similar to the web, most of the data that needs to be processed in the context of IoT is stored in the lowest tier of the memory hierarchy. Instead of requesting data from remote servers, in the IoT the information is requested from devices on the device layer. Since in many cases sensor values are requested, the requested devices often need to do sensor readings before they can answer the request. Since these sensor values often change, the cached data needs to be checked regularly in order to ensure consistency, i.e. it needs to be 'fresh'. This freshness imposes that the stored data used for replying to requests needs to be measured within a reasonable time frame before the request is served. In addition, when technologies like NDNs are used, strategies need to be evolved to keep data consistent over multiple caches, since every node is capable of storing data temporarily. Since some connections to devices in the network may not be reliable, which leads to services becoming unavailable, caching is needed to reduce the service loss. The authors of [25] have shown that this is possible regardless of how the caching mechanism is implemented. Another benefit of caching in IoT is the reduction of the load on sensors in the system, making it more energy efficient. It is however

necessary to find a trade-off between the freshness of the data and reduction of the loads on the devices. An example strategy is described in [48] where the authors have shown that it is possible to maintain a high cache hit rate while maintaining a high accuracy of measured sensor values. Another approach is to introduce a *time to live* value that indicates when data has become stale with high probability in order to ensure the accuracy of the served requests. Furthermore, the implementation given in [10] was able to reduce the transmission in the network by 50%. These examples along with the implementation of the in-network capabilities in NDNs have shown that caching is also possible and beneficial to IoT networks. Finally, since most of the communication that happens in an IoT system is not triggered by human interaction but is based on Machine-to-Machine communication (M2M), some patterns as discussed in Section 2.1 will emerge. Since most of the caching strategies rely on predicting future requests, these patterns can be used to optimize the caching strategies in order to improve the cache hit ratio. These strategies are introduced in the following section.

2.5 Caching strategies

The performance of a cache depends on the prefetching and replacement strategies used. These strategies make its predictions based on the observed access patterns and information about the data stored in cache. Depending on how well these strategies work, the caching system can achieve a high cache hit ratio and perform better than systems that do not cache at all or systems using unsuitable prefetching and replacement strategies. A study conducted in [18] found that an optimal caching strategy must satisfy four properties.

The first property is about *optimal prefetching*, describing that a prefetch of data should load the next needed data block into the cache that is not already in the cache in order to make it available when it is needed. The second property is the *optimal replacement* strategy. It specifies that whenever a data block in the cache needs to be replaced, the strategy should choose the data block that is used in the most distant future should be discarded. It is important to note that this property should not violate the *do not harm* characteristic of the caching system. This implies that no block should be replaced by a new block if the new block is processed after the block which is already in the cache. Finally, a prefetch-and-replace operation should always happen at the *first opportunity*. This property states that the strategy should perform its task whenever the opportunity arises, which is either when a prefetch has been completed or a block that would have been discarded according to the optimal replacement strategy was used in the preceding time interval. Some existing and widely used approaches violate some

of these rules, leading to *thrashing*, which occurs when data blocks that are used more often are replaced by less used data blocks, leading to more data operations than needed throughout the execution of the program [18].

In the following sections it is explained how cache replacement strategies and caching decision strategies make decisions on which data should be kept in the cache to perform optimally.

2.5.1 Cache replacement strategies

One of the desirable properties identified in [18] is the *optimal replacement* strategy. A naive approach for a replacement strategy is the *first in first out* (FIFO) approach. In this strategy, data that is loaded into the cache first is also replaced first, regardless of the access pattern of the program [32]. This approach therefore violates the *do not harm* property required for an optimal strategy. Other caching strategies try to improve upon FIFO by considering additional data access patterns. The decision which block should be replaced next is then made with respect to some metric, including but not limited to [52] [5]:

- Recency
- Frequency
- Recency and Frequency
- Functional
- Randomized

In the following some commonly known strategies that make use of these metrics are introduced.

Recency Based Strategies

The first class of strategies bases its decision on the last access to an object. Objects that were used recently in are considered more important and are less likely to be replaced than objects that have not been accessed for a longer time. The simplest implementation of this strategy is *Least Recently Used* (LRU), which always replaces the object that has not been accessed for the longest time. Improvements on LRU rely on additional metrics that impose further restrictions on which objects in the cache are replaced. An example is the LRU-*threshold* where data is cached only if its size is bigger than a specified threshold [4]. This implementation makes efforts to keep larger

files in cache and to refetch small files, therefore reducing the access times to large data blocks. A similar approach is taken in the *partitioned caching* strategy which classifies objects into 'small', 'medium' and 'large' objects. Each class has dedicated space that is managed separately with the LRU strategy. This improves the LRU-threshold strategy since all types of objects are cached. Another adaption is the *SIZE* strategy that always replaces the biggest object in the cache and only uses LRU if the largest objects have the same size. There are further adaptations of LRU that calculate an *importance value* for each object based on a function that also considers the recency of the last access. Examples for these strategies are *HLRU*, *EXP1* and *value-aging* [52]. Even though there are many variants of LRU, the basic LRU strategy is one of the most commonly used replacement strategies in ICN [29].

Frenquency Based Strategies

Another commonly used class of replacement strategies makes its decision based on how frequent an object is accessed. The standard strategy in this class is called *least frequently used* (LFU). That basic LFU approach replaces the object in cache that is accessed least frequently. There are two versions of LFU – *perfect LFU* and *in-cache LFU* [17]. In-cache LFU only keeps track of how frequently the items in cache are accessed and discards the counter of an item when it is removed. Perfect-LFU on the other hand keeps track of all accesses to an object, even when it is no longer in the cache. This comes at the additional cost of maintaining the counter for all items that have ever been stored in the cache. When using LFU it can happen that objects that were popular before remain in cache for a long period of time, even when they are not accessed anymore. In order to overcome this problem *LFU-aging* builds an average access counter of all objects in the cache. In case this average exceeds a threshold, all access counters are divided by 2. The same process is also triggered when an object hits a maximum access count [8]. By using this strategy, it is ensured that objects that have been popular in the past are not kept forever in cache and are removed eventually. Another approach that tries to deal with this problem is the α-*aging* strategy, where each object counter is multiplied with $\alpha \in (0, 1]$ after some specified period of time [55]. Finally it can be said that LFU strategies are, similar to LRU strategies, widely used and applied in different fields, especially in the context of ICNs [29].

Frequency and Recency Based Strategies

In addition to LRU and LFU strategies on their own, there are also combinations of the strategies that try to overcome the issues of each class. An example for such an

approach is the *segmented LRU* (SLRU) strategy, which splits the cache into two segments. Frequently accessed objects are kept in the first segment from which data cannot be removed. This segment is therefore also called *protected segment*. The other – *unprotected* – segment of the cache holds less frequently accessed objects. Whenever an object in the unprotected segment becomes more popular, it is moved to the protected segment, while objects that lose popularity are moved to the unprotected segment. The replacement strategy used to manage both segments is an adaption of an LRU strategy.

Function Based Strategies

Instead of using recency and frequency as metrics to make decisions it is also possible to use a *function based* strategy. These strategies make use of a predefined function that takes other indicators into aspect to determine a value for each object in the cache. An often used category of replacement strategies in this class are the *Greedy Dual (GD)* strategy and its adaptions [36]. In GD, a value H_i is maintained for each object i that is cached. Depending on the adaption of the GD, the value H_i is calculated differently. In *GD-size*, H_i is calculated as $H_i = \frac{c_i}{s_i} + L$, where c_i denotes the cost of fetching and s_i the size of the object i. The factor L is an aging factor that changes over time and depends on how long the item is stored in cache. L is initially set to 0. After removing the object with the lowest H_i value, L is set to this H_i. Other adaptions of the GD strategy like $GDSF$ or $GD*$ use slight adaptions to determine H_i. Another function based caching algorithm is *RIPQ*[63] which implements a priority queue as cache replacement method.

Randomized Strategies

The final class introduced in this thesis is the *randomized* class in which algorithms choose the object which is replaced at random. In addition to this straightforward approach, some adaptions of strategies introduced earlier were also made using the random strategy, e.g. *HARMONIC*, in which an item is removed with the inverted probability of its cost. Even though there are no proven benefits to the randomized strategy, it is also commonly used in ICNs [29].

2.5.2 Storing strategies

The strategies introduced above are concerned with replacing items when the cache is full. In addition to these strategies, there are also methods that determine which items should be cached at different places throughout the network. This is especially important for concepts like ICNs and NDNs, where no dedicated data caches are present,

but all nodes have the capacity to temporarily store information. One strategy that can be used is the *always strategy* that caches every incoming packet at every node [29]. Since each node stores a replica of every data it has encountered, a lot of redundancies are introduced. This can be beneficial in case some services fail, but also imposes a lot of work to ensure the consistency of the data. Different approaches have been proposed in order to use the existing resources more efficiently. One approach is *probabilistic caching* [53], in which each cache stores the observed content with some probability. This leads to a distribution of the content throughout the network without storing it in every cache. Even though this approach can also be used in the context of the IoT, it was initially designed to be used for traditional Internet content [29]. Other strategies use different approaches on where to store content when there are multiple caches present in the network [3]. In ICNs so called *on-path* and *off-path* strategies for storing data at the nodes have also been explored [21]. When on-path caching is used, items are stored at nodes that are on the path between the data source and receiver. The off-path strategy avoids adding this redundancy by storing popular items on central caches and less often requested items at other caches.

Finally, caches can increase their efficiency by prefetching data into the cache before it is actually requested. Some of the approaches used to do this task are explained in the following section.

2.5.3 Cache prefetching strategies

Some caches are capable of predicting the data which will be requested in the near future. If done correctly, proactively caching data can greatly decrease latencies and increase the effectiveness of caching systems. In order to make predictions on which content may be needed next, the prefetching strategy needs to evaluate which objects have been referenced in the past. In addition to predicting the correct data to load into the cache, it is also necessary that the prefetching operation happens at the right time. When the prefetching is done at an unsuitable time, unnecessary replacement operations may take place. This can happen, for example if an aggressive prefetching strategy is chosen where items in cache are replaced too early and are replaced before they are accessed in order to free up space for other objects [18]. Prefetching can be realized in two different ways. If processes and applications indicate which resources are required next, the mechanism is called *informed prefetching*. The second method requires making a prediction based on the access pattern of the applications and is therefore called *predictive prefetching* [46]. Since the predictions heavily rely on the context, different strategies have evolved for prefetching in web applications that differ from mechanisms

that are deployed for the use in file systems. The schemes introduced in the following are limited to predictive strategies that are used in web and ICN applications. As mentioned above, many approaches rely on specific characteristics of the environment in which they are implemented. Despite of this, the authors of [49] introduced a mechanism that is an adaption of a strategy initially developed for prefetching in file systems. In this approach, a dependency graph is built which represents how documents and information are accessed in relation to each other. This dependency graph is then used to predict files that are most likely accessed next. This algorithm is an exception to most prefetching mechanisms that are deployed in the context of the Internet since most of the mechanisms used rely on modeling user patterns. In order to find user patterns that can be used later, server logs are processed in different manners to gain insight on how different content is related [46]. One example of an application that makes use of such predictions is Netflix, which prefetches video and audio onto servers close to users in order to reduce the latency when starting a video in the next session [61]. Another strategy that mines association rules from server logs is introduced in [40]. The mechanism implemented processes server logs in order to find association rules that describe the probability of a certain file being accessed after another file. In case the probability of a file access is above some threshold, it is pushed to the client and therefore prefetched.

In the next section some metrics for measuring the performance of caches are introduced.

2.5.4 Evaluating Caches

After setting up caches it is reasonable to evaluate how much the applications benefit from caching compared to the additional overhead that is required to manage a cache. In order to find out more about the impact of the cache on the system, measurements are needed. One metric to evaluate the performance of caching algorithms is the *cache hit ratio* (CHR) which measures how many incoming requests could be served from data in the cache. A high cache hit ratio indicates that the cached items are often requested and few refetches need to be done. In the context of the IoT and web the CHR also indicates how much bandwidth could be saved by [73]. Another metric that can be evaluated is the *data retrieval delay* which is defined as the time it took from receiving the request until serving the answer. A high delay is often correlated with a low cache hit ratio since this often means that the requested data needs to be loaded from the source. A low CHR can also be measured in the *link load* of a network which indicates how much data is exchanged between neighboring nodes in the network. A metric that is additionally important in the context of IoT is how fresh the served data is. This can

be measured by evaluating when the data was received and for how long it remained in the cache. By using time to live value for a mechanism that discards stored data after some time a minimum freshness of the served requests can be guaranteed [75].

2.6 Machine Learning

Since this thesis tries to enhance caching in the context of the IoT by using machine learning, it is necessary to introduce some background about it. Machine learning (ML) can be defined as a '*set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty*' [[45], p. 1]. Since caching strategies make their decisions based on predictions of the future, it is likely that ML can be used to improve these predictions.

2.6.1 Supervised and Unsupervised Learning

ML algorithms can be classified into two categories:

1. *Supervised*, or predictive learning
2. *Unsupervised*, or descriptive learning

In supervised learning, the training data consists of labeled examples for each class. Based on these examples, the algorithms are supposed to learn a model that can assign the correct label for a new data point that was not part of the training set. The labels can either be categorical or continuous. In case they are categorical, the problem is referred to as *classification*. If the model should predict a numerical value for each data point based on continuous labels, the process is called *regression*. Thus, supervised learning is concerned with approximating the mapping function from data points to labels by minimizing some objective or loss function. This loss function ensures that the model explains the data in the training set well enough but is still able to generalize to new data. Some prominent algorithms for supervised learning are Support Vector Machines (SVM) [15] for classification and linear or logistic regression. Unsupervised learning on the other hand is concerned with finding patterns in datasets that are unlabeled or where the number of classes is unknown. A popular algorithm for unsupervised learning is *k-means*, which divides the input data into k clusters based on their similarity with respect to some metric. Unsupervised learning is often used in dimensionality reduction method like the principal component analysis (PCA), where the dimensions with the lowest variance are selected and discarded [45][14].

2.6.2 Basic Machine Learning Concepts

In order to apply ML it is necessary to specify the underlying problem. Depending on how the problem was formalized different classes of ML algorithms can be applied to solve it. In this section some key concepts for formalizing and specifying a model are discussed.

Parametric, non-parametric and semi-parametric models

When a model is specified it can be either *parametric*, *non-parametric* or *semi-parametric*. A model is called parametric if it has a fixed number of parameters regardless of the used training data. Since the number of parameters is fixed, models of this class are less flexible. They are often faster to use than non-parametric models which can become more complex over time. This comes at the cost of flexibility as the model cannot be adapted when new information is gained. In addition to that, parametric models require some assumptions about the underlying distributions of the data in order to fit the scenario in the model. If a model's number of parameters grows with the size of the training set, it is considered to be non-parametric. These model assume that the underlying distribution cannot be described by a fixed number of parameters. Therefore the model can be extended when more training data becomes available, making it more flexible than a parametric model. This comes at the cost of increased complexity that makes the model computationally intractable when the training set is large [45]. In order to reduce some of the complexity of these models some parts of them are often replaced with parametric models. A model that consists of both parametric and non-parametric parts it then called *semi-parametric* [30].

Curse of dimensionality

A problem that often occurs when dealing with high dimensional data is the *curse of dimensionality*, especially when non-parametric models are used. It describes the phenomenon that an exponential increase of training data is needed to cover the full space when the dimension of underlying model increases. Obviously, this also has effects on how well a model performs. One way to overcome this curse and to reduce the dimensionality of the model is to make assumption about the underlying data which is then represented in parametric and semi-parametric models [14][30].

Online and offline learning

When a model is trained with data, two different approaches can be taken

- *offline learning*
- *online learning*

In offline learning, all training data is processed at once during the training phase. After the training of the model has concluded the model is then used to process new data and does not change anymore. In case the training data is not fully available or cannot be processed at once, online learning is used. Here, the model is trained by using only parts of the data. The model's parameters are updated after each part of the training data was processed. This incremental style of training a model is usually slower than offline learning but makes it possible to change the model when new training data becomes available without having to relearn the whole model [45].

2.6.3 Choosing the right model complexity

In order to apply machine learning in the most efficient way it is necessary to choose the right model.

A model can perform differently depending on its complexity. In order to choose the right model complexity it is necessary to consider the error the model makes when it is presented with new data. Therefore the trained model is evaluated on a test set to determine the *generalization error*. The generalization error represents the expected misclassification rate of a model when processing new data. It is therefore desirable to choose the model with the lowest generalization error [45].

Overfitting and underfitting

In order to minimize the error on the training data it is possible to learn a model that represents the underlying training data perfectly. A problem that occurs with this approach is called *overfitting*. To be able to generalize the model from the training data it is necessary to avoid modeling every minor detail of the given dataset. Therefore, overfitting is an indicator that the underlying model is too complex and needs to be simplified in order to reflect the underlying scenario.

A model not capable of representing the complete underlying mechanics is underfitted. This means the model is not complex enough to describe the data accurately. In order to find a good model for a scenario an optimum between overfitting and underfitting needs to be found [14].

In the following some common tasks from both supervised and unsupervised learning are introduced.

2.6.4 Machine Learning Tasks

There are many fields that use machine learning to gain insights about gathered data and make decisions based on those insights. Even though these fields often differ, the tasks fulfilled by ML are often similar. In the following some standard ML tasks are introduced.

Classification

The first task that machine learning is used for is *classification*. In order to classify new data, the model is trained with a labelled dataset first. After the training of the classification algorithm has concluded, the labels of new data points are predicted according to the model learned from the training data. An example algorithm for classification are decision trees in which data points are separated by testing features for certain characteristics. Each decision partitions the input space into different regions [54]. Each region in the space then represents a leaf of the tree which is used to label the data. Even though *classification and regression trees* (CART) are usually introduced together, *regression trees* are used for a fundamentally different purpose and other machine learning objective – *regression*.

Regression

The task of regression is to model the relationship between variables based on observed data, i.e. finding a mapping from the training data to labels. An example for such a relationship is a dataset that includes the size and weight of persons. By using the correlation that is seen in the dataset, the regression algorithm can predict the weight of a person given their size. Common algorithms used for this task are linear and logistic regression.

Clustering

Another machine learning task is '*[...]the process of clustering similar objects together*' [45]. To achieve this, a similarity measure is required which enables the algorithm to compare the data points. This measure also affects how the process works. In *similarity-based* clustering a distance measure is needed that can correctly group data points based on their similarities, e.g. the euclidean distance. This is not necessary in *feature based* clustering where specific features of each data point are used to determine the distance to other data points by comparing them directly [13].

These are just some of the common ML tasks. In order to give some context to the actual algorithms that are used, the two most relevant approaches for this thesis are introduced in the following.

2.6.5 Relevant Machine Learning Algorithms

There are many algorithms to choose from in Machine Learning. In the following, the most relevant algorithms for this thesis are introduced.

Clustering algorithms

The first class of algorithms that are considered are designed for clustering. There exist both offline and online learning algorithms for this task. Clustering algorithms can have two different types of output:

1. disjoint sets of the data points
2. a nested tree of partitions

Disjoint sets are the result of *partitional clustering* while nested trees are the result of *hierarchical clustering*. Even though hierarchical clustering is more useful than partitional clustering in most cases, it is computationally more expensive. In addition to that, most hierarchical cluster algorithms are deterministic and do not require model-selection before the clustering is done. This needs to be done in partitional clustering algorithms like choosing an appropriate k for *K-Means* and K-NN. Furthermore, the outcome of partitional algorithms also depends on initial conditions and is therefore not deterministic.

As mentioned before, it is necessary to choose a distance measure that is capable of comparing data points. In most cases, similarity between data points is calculated by comparing each attribute directly. How these attributes are compared depends on the analyzed data. When dealing with continuous data, standard distance metrics like the euclidean distance can be used. In case the data is categorical (e.g. sunny, overcast, rain) the distance is often calculated by the Hamming distance which adds up the attributes that differ from each other.

A well known partitional algorithm is the *K-Means* algorithm [31]. The K-Means algorithm is initialized by choosing K random centers μ_i. Then, for every data point the distance to each μ_i is calculated. It is then assigned to the cluster to whose μ_i it has the closest distance. For each class, the μ_i is recomputed as the mean of all data points that are member of class i. This is repeated until convergence. As it can be

seen, the outcome depends on both the chosen k and the randomly chosen centers of the clusters. The *K-Means* algorithm only works on continuous data as the distance of categorical data cannot be calculated by using standard distance metrics. Therefore an adaption of *K-Means* for categorical data was developed. It is called the *K-Modes* algorithm [34][33].

Instead of measuring the distance of the data points to all μ_i, *K-Modes* calculates dissimilarities between the data points. The dissimilarity is a score of how many features of two compared data points differ, i.e. when comparing two points X and Y with n features, the dissimilarity is calculated as

$$d(X, Y) = \sum_{k=0}^{n-1} \mathbb{I}(X_k \neq Y_k)$$

In addition to that, *K-Modes* replaces the μ_i with *modes*. A mode $Q = [q_1, ..., q_k]$ of a dataset D is a vector that minimizes the dissimilarity score between itself and each data point $X \in D$, i.e. a vector that minimizes

$$M(D, Q) = \sum_{X \in D} d(X, Q).$$

When $n_{c_{k,j}}$ denotes the number of objects that have kth category in feature F_j and $f_r(F_j = q_j|D) = \frac{n_{c_{k,j}}}{n}$ denotes the relative frequency, a mode can easily be found by using the fact that $M(D, Q)$ is minimized if and only if

$$f_r(F_j = q_j) \geq f_r(F_j = c_{k,j}), q_c \neq c_{k,j}, \text{for all features } F_j \text{ [33]}$$

The *K-Modes* algorithm then works similar to *K-Means*. In the first step, k modes k_i for the dataset D are generated. For each data point X in D, the dissimilarity $d(X, k_i)$ is calculated and X is assigned to the cluster with the most similar mode. After the assignment is done, the mode is updated by using the rule introduced above. After all $X \in D$ are assigned to clusters, the algorithm checks the dissimilarity to the k_i. In case some X are closer to some of the adjusted k_i they are reassigned to the cluster of the closer mode and the q_i of both clusters are recalculated. This step is repeated until convergence.

In contrary to calculating the distance or dissimilarity of data points to each other it is also possible to consider density functions to determine the clusters. An example for this is the *Density Based Spacial Clustering of Applications with Noise* (DBSCAN) algorithm [58] [24]. This algorithm additionally requires a neighborhood-value that indi-

cates the maximum distance two data points can have such that they are still considered neighbors. Finally, a value is needed that defines how many neighbors are required so that an area is considered dense. Based on these values DBSCAN returns a set of clusters in the dataset. Data points that are not part of a cluster based on the given values are considered as noise.

Most algorithms used for clustering are designed for data that does not change over time, e.g. when the data changes the whole process of clustering needs to be repeated. In order to deal with this there are also clustering algorithms for sequential data. Some algorithms are capable of performing clustering over changing data. The processed dataset is for example a sequence recent measurements. Since the underlying data is constantly changing, algorithms need to deal with changing, disappearing and new clusters. In addition to that, they need to be able to identify outliers quickly and to represent the results in a compact manner [58]. An example of an algorithm that is capable of processing streaming data is the *Balanced Iterative Reducing and Clustering using Hierarchies* (BIRCH) algorithm [72].

The second approach proposed in this thesis makes use of reinforcement learning which is explained in the following.

Reinforcement Learning

In Reinforcement Learning (RL), an agent learns a given model by using a trial and error approach [37]. The agent has a set of actions it can perform. These actions then have an impact on the environment. The agent chooses its next action based on the state of its environment. The actions the agents can perform are contained in the *action space*. Depending on the state the agent is in, different actions from the action space can be performed. The environment changes based on the action performed by the agent. In most cases the agent is not the only influence on the environment and its action only partially affect how it evolves. Each action taken by the agent results in two types of feedback. The first type of feedback it receives is a reward. This reward can be negative or positive depending on how the environment evolved and how the goals of the agent are set. The second type of feedback the agent gets back is an observation of how the environment has changed, i.e. a new state. Every possible state is part of the *state space*. This scenario is illustrated in Figure 2.4.

In order to perform efficiently, the goal of the agent is to maximize the reward it receives for its actions [14]. In other words, reinforcement learning is concerned with finding the best action in a given situation.

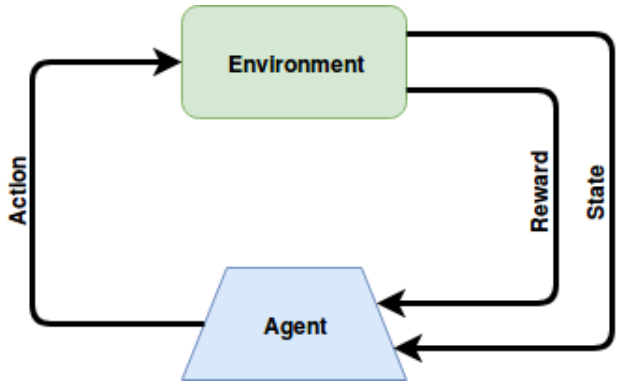


Figure 2.4: Standard Reinforcement learning scenario after [62]

Reinforcement Learning is often described by a Markov Decision Processes (MDP). A Markov Decision Process describes a sequential decision problem in which the state is fully observable at all times but state transitions are stochastic, i.e. the next state the agent will be in after taking an action is uncertain. Formally, a MDP can be described as a tuple $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{X}$ denotes the state space, $\mathcal{A}$ the action space, $\mathcal{P} = P(x'|x, a)$ the state transition probability and $\mathcal{R}$ the reward function [56].

A solution to the MDP is called a *policy* π. For each state, $\pi(s)$ is the action recommended by the policy in state s. Intuitively, a good policy is one that chooses the action that maximizes the expected utility. But, since the environment is stochastic, the outcome of an action is uncertain. The quality of a policy is therefore related to the *expected utility* in that an optimal policy π^* maximizes the expected utility.

A common way to solve these MDPs is by using dynamic programming. In dynamic programming large problems are separated into smaller overlapping sub-problems for which solutions need to be found. These solutions are then combined to provide a solution to the overall problem [12]. The dynamic programming approach shifts the focus from finding a well performing agent to finding an appropriate value function. Since it is often not feasible due to the large amounts of states that can be observed in large-scale problems, RL algorithms make use of two concepts. The first concept is the reduction of the whole problem into a set of samples. This is done in order to compactly represent a simplification of the underlying dynamic. Furthermore, algorithms in RL make use of function approximations for the compact representation of value functions. Algorithms that are used in RL therefore connect dynamic programming along with sampling and function approximation to efficiently reach the goal of learning an optimal policy.

There are multiple approaches that can be used to find the optimal policy π^*. The naive approach is to simply list all possible policies and to determine the best one. Since this strategy is infeasible for large problems, different solutions had to be explored. The first approach tries to compute an optimal value function V^* to get the highest expected reward for a state, assuming it is started in this state. π^* is then chosen such that the highest expected reward is achieved in every state. Similar to this approach, it is also possible to calculate an *action-value* function Q^π, which calculates the value of an action given a state and a policy π. Analogous to calculating V^* it is also possible to calculate an optimal action-value $Q^*(s, a)$ for state s and action a. The class of value and policy iteration algorithms makes use of these ideas in order to find the optimal policy π^*. Even though these algorithms can be used to find optimal behavior, they require the knowledge of the transition and reward function.

A large application area of RL are *control* problems. In this scenario the agent tries to learn a policy that performs optimal in a given environment. The class of control problems is divided into *interactive* and *non-interactive learning* problems, depending on if the agent can influence the observation it makes or not. Since in the scenario considered in this thesis the agent does not have any influence on its environment, only algorithms for non-interactive learning are introduced in the following. Since the agent does not know the transition and reward function it needs a way to learn these such that its behavior can be optimized. Therefore the agent constantly evaluates and optimizes its V and Q functions in order to incorporate new knowledge and observations. Some of the algorithms in this class even update the given policy before it has been fully evaluated. This is called *generalized policy iteration* (GPI) [62]. The algorithms work by using two separate processes working closely together. The first process is called *actor*. Its purpose is to improve the current policy. This policy is then also evaluated by the *critic*. There are a lot of different ways to implement such an actor-critic architecture [38][27], especially since the actor policy and evaluated critic policy often differ. Since actor and critic work separately, there are different algorithms that are used for those tasks. One layout of such an architecture can also be seen in Figure 2.5.

As mentioned before, critics estimate the value of the used policy of the actor Therefore adaptions of algorithms for value prediction problems are commonly used. These can vary from efficient $Q - learning$ algorithms to adaptions of the *TD(λ)* algorithm like SARSA(λ) or LSTD-Q(λ).

To implement the actor, two approaches can be taken. Either, by changing the policy in a greedy way according to the values predicted by the critic, or by performing gradient ascent on the surface of the parametric policy class [62]. An example algorithm for this task is LSPI(λ).

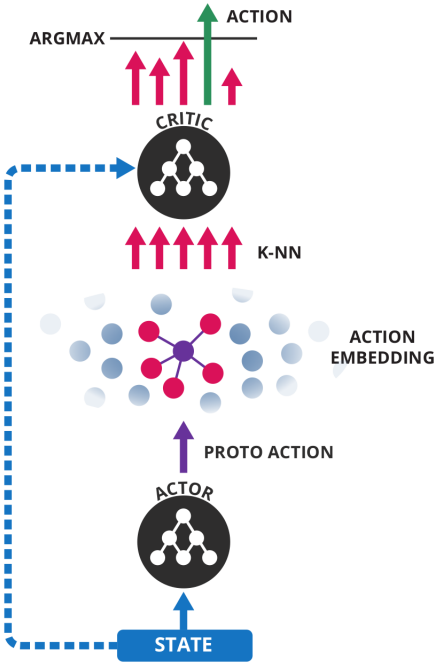


Figure 2.5: Layout of the Wolpertinger architecture as in [22]

Since exploring all possible actions for a state is computationally expensive and directly affects the calculation of the Q- and V- functions, different approaches have been taken in order to make actor-critic implementations more efficient. One of these approaches is the Wolpertinger architecture [22]. Similar to the standard approach, this architecture is divided into an actor and a critic part. In contrary to the standard model, an additional layer of a K-NN is added between the actor and critic. For training the actor the *Deep Deterministic Policy Gradient* [41] algorithm is used. In the Wolpertinger architecture, the actor determines a *proto-action* based on a function that maps states to an optimal action. While doing so, this function does not regard the actual action space $\mathcal{A}$. Therefore it is possible that the proposed proto-action is not in the action space for the state. This has the benefit that not the whole action space needs to be explored to choose an action. Instead, the proposed action has to be mapped to a valid action in the action space. This is done by using the K-NN in the next step to select the k most similar valid actions from the action space. Choosing the k closest actions is necessary since, even though most actions that are similar to the proto-action are likely to result in a high Q-value, the closest action might be an outlier resulting in a sub-optimal action. Therefore, the final step is to choose the action with the highest Q-value from the determined neighbors, while regarding the feedback given from the critic.

The final step is then to choose the action from this set of actions that has the highest Q-value. This approach of realizing an actor-critic architecture is also depicted in Figure 2.5.

2.6.6 Difficulties with applications of machine learning

When using machine learning, there is a set of issues that needs to be addressed for a successful deployment. Depending on how large the dataset that is used to train the model is, computational limitations pose an issue. On the other hand, when the dataset available to train the model is limited, the approximation-estimation error also needs be considered. These errors occur when the used algorithm generalizes from a small sample set [16]. Furthermore, it is important that the dataset used to train the model is coming from the same distribution as the data from the application. Even slight deviations from the actual distribution of the underlying problem can lead to a bad performance of the trained model [42]. Another issue that needs to be addressed is that even with a trained model there is a trade-off between speed and accuracy of the predicting algorithm [43]. Also, when relying on a pre-trained model without online training, e.g. classification without stream clustering, it is also necessary to implement mechanisms that measure the accuracy of the algorithm and indicate when retraining is needed. This could for example be realized by tracking the error rate and trigger a new training phase if the error rate exceeds a certain threshold.

2.7 Requirements

From the preceding analysis it can be seen that many things need to be considered when machine learning should be used to improve caching. Since the overall goal is to improve caching, the main requirement is to achieve an increase in the cache hit ratio (R1). This can be done by optimizing either the cache decision strategy (R2), the cache replacement strategy (R3), or both. These improvements should be achieved by using the additional semantic features that are introduced with ICNs and NDNs (R4). As an effect, this should also reduce the amount of network traffic that is present in the IoT system (R5) as well as the latencies (R6). In addition to that, the caching strategies should make use of the patterns that occur due to the M2M communication in IoT networks (R7). Further, as mentioned in Section 2.1, data sinks that process the gathered data rely on a certain freshness of the information. Therefore, the algorithm should have a mechanism that ensures freshness and consistency of the stored data (R8). When using machine learning to implement the functionality for those requirements it

is necessary to identify the available and relevant features for learning a model that helps to enhance the caching mechanism (R9). Even after the standard pattern of a network has changed it is necessary to continuously improve the mechanism along with changes of the communication pattern (R10). Since in IoT networks the computational resources are limited, the learning process and execution of the algorithm should also be lightweight (R11). Finally, since in ICNs and NDNs caches are usually distributed throughout the network it should be possible to employ the caches on any node on the network without them interfering each other (R12). In addition to that, they should find the optimal strategy for the locally observed traffic (R13). All of these requirements are also listed in table 2.1.

Number	Requirement
R1	Increase cache hit ratio
R2	Improve cache decision strategy
R3	Improve cache replacement strategy
R4	Make use of semantic features introduced in ICN and NDN
R5	Reduce network traffic
R6	Reduce latencies
R7	Make use of M2M communication of IoT systems predict future traffic
R8	Ensure freshness of cached data
R9	Identify available and relevant features for ML
R10	Strategies are capable to adapt to changes in the observed patterns
R11	Lightweight execution for resource limited nodes
R12	Capability of employment on multiple nodes
R13	Caches act optimal based on the locally observed communication

Table 2.1: List of requirements

Chapter 3

Related Work

The idea of enhancing caching with machine learning is not a new one. There already exist numerous approaches that apply machine learning to caching, and some that adapt existing caching strategies to ICNs. This chapter briefly introduces some of these methods and addresses their shortcomings with respect to caching in IoT scenarios.

One approach that uses caching to answer similar requests when the cached data is fresh enough is proposed in [75]. The authors make use of the periodic fashion in which many queries to the system are made. Based on the last k requests they determine local areas of the network which are requested more frequently to prefetch data from those areas for future requests. The idea behind this approach is that many requests are related to each other and hence need to be processed differently to the standard approaches where each of them is handled separately. The requested data is cached in a single data sink that queries sensors in the network. The sink also calculates a popularity index of certain areas of the network based on the history of the last requests. This index is then used to prefetch data from nodes in these areas if data from it is not already in the cache of the sink node. To prevent data from being stored too long and becoming stale, the authors introduce a TTL that is used to indicate how long data can be cached until it needs to be replaced. The evaluation of this strategy showed a high CHR for the determined popular areas which lead to lower latencies and less network traffic. In contrary to the approaches considered in this thesis, the strategy only uses some history to determine popular locations of the network.

An implementation that uses machine learning to improve caching is presented in [42]. In this approach, a caching algorithm is enhanced with a machine learned oracle that helps to determine which data needs to be removed from the cache. By adding the oracle

to the standard algorithm to enhance the decision making, the oracle increases the cache hit ratio if it performs well. After running separate experiments to evaluate the concept, the authors show that the new implementation outperforms standard implementations. The proposed algorithm is not implemented to enhance caching in ICN and IoT but only relates to the standard implementation of cache replacement strategies used on computers.

In [6], two enhanced GDSF implementations are introduced. One implementation makes use of Support Vector Machines while the other one uses a decision tree in order to optimize the standard GDSF strategy. In addition to the GDSF implementations, the authors also introduce an enhanced implementation of LRU that makes use of Support Vector Machines. After training the algorithms with the server logs from training sets of different web proxies, the authors evaluated the performance of the optimized strategies and show that it is possible to enhance existing cache replacement strategies with machine learning in the context of the Internet.

Similar to the approach discussed above, the authors of [68] use data mining of log files to determine patterns in accesses to files on web servers. The strategy learns association rules between different web pages to send hints to the proxies which content may be accessed next. These proxies then incorporate these indications into their cache decision and replacement strategies. This approach has been proven to increase the cache hit ratio in comparison to standard strategies used on web proxies.

Another strategy supported by machine learning is introduced in [26]. The approach introduces a policy that changes the replacement strategy based on the observed access patterns. By monitoring the access patterns, the master policy determines which strategy should be in control of managing the cache. In order to do this some space of the cache needs to be reserved to keep track which items the other strategies would keep in the cache. Whenever a strategy takes over, the respective files need to be refetched into the cache. The evaluation of this approach shows that this mechanism outperforms standard techniques since the strategy can adapt to changes in access patterns and choose the best policy for the observed pattern.

Another approach that enhances caching is proposed in [74]. The authors use reinforcement learning in order to learn a long-term replacement strategy. The presented approach uses the Wolpertinger architecture [22], which helps in the reduction of the action space by discarding certain actions and still enabling online learning. The algorithm is split into an offline and online learning phase. During the offline phase, the model is pretrained with historic data. After the pretraining, the model is used with online learning which is updated after some time. After each decision, the agent

observes how its decision is rewarded and adapts accordingly to learn the long term access pattern. This approach has been tested and evaluated in a simulation and has shown to outperform standard replacement strategies like LRU and LFU with regards to cache hit ratio. Further, the strategy was capable to adapt to changing patterns and to maintain a high cache hit ratio. In contrast to this thesis, the proposed framework is not deployed with an IoT setting in mind. Therefore, in order to be suitable for this scenario, the approach has to be adapted.

Similarly, [59] uses reinforcement learning to improve caching. In contrast to the approaches introduced before, the authors propose a strategy that tries to prefetch relevant content. Further, this model considers that data is created over time and has a different lifetime, i.e. information is only considered relevant for a limited time. In order to achieve this, the cache management is defined as a Markov Decision Process with side information that is used to find the best long-term policy for prefetching. One metric that is used in this scenario is the remaining lifetime of the content that is used to determine the importance of an item that should be loaded into the cache. The scenario in which this strategy is used is the prefetching of content from a social network onto the mobile device of a user. Similar to IoT, content is created over time and remains relevant for a limited time. In this case, the access patterns of the user are incorporated in order to predict relevant content and to optimize when this content is downloaded onto the device.

In conclusion, some approaches have been proposed in order to optimize caching. Furthermore, most of the work focuses on cache replacement strategies. In this context the new frameworks improve existing strategies by extending them in order to improve the predictions that are based on the current features, but do not regard semantic features. Further, few approaches have been taken in order to improve caching in the field of IoT, but improve caching mechanisms in different fields. An overview of the topics that are covered by the related work can also be seen in Table 3.1. In this thesis, the semantic features and other patterns available in IoT play an important role for the usage of machine learning for improving the performance of caching in IoT.

Paper	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
[75]	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓
[42]	✓		✓						✓	✓			
[6]	✓		✓		✓	✓			✓	✓			
[68]	✓	✓	✓			✓			✓				✓
[26]	✓		✓						✓			✓	✓
[74]	✓		✓						✓	✓			✓
[59]	✓	✓							✓	✓			✓

Table 3.1: Related work

Chapter 4

Design

In this chapter some possible answers to the question '*How can machine learning be used to improve caching in the context of the IoT?*' are discussed. In the rest of this chapter, approaches are outlined that try to answer this question while considering the specified requirements in table 2.1.

4.1 Design possibilities

The requirements specified in Section 2.7 cover aspects from the IoT, ML and caching that need to be considered when developing a new approach. To improve the caching and thereby the cache hit ratio (CHR) either the prefetching or replacement strategy can be enhanced. To improve the predictions of the caching strategies, patterns occurring in IoT networks can be analyzed with machine learning. This can be done by either improving the decision making of existing strategies like LRU and LFU, or developing a new strategy that makes its decisions purely based the suggestions of the machine learned model. Furthermore it is also possible to switch between standard strategies based on observed patterns as implemented in [26]. To maximize the benefit of machine learning, it is necessary that the chosen model also considers the additional semantic features from ICNs to make predictions. The solutions suggested in this thesis focus on improving the CHR by designing new cache replacement strategies. To achieve this enhancement, the chosen machine learning models analyze both access patterns as well as the semantic features to make their decisions. These decisions should also be affected in case the observed pattern changes, i.e. it is not sufficient to rely on a purely offline trained model. Finally, it is necessary to design a lightweight process that can also

run nodes with limited resources. In the following section the scenario for which these algorithms are designed is described.

4.2 Scenario

The scenario considered for designing the new approaches is a simplified version of IoT networks. It is composed of a cache with limited in size C. This caching node connects to multiple other nodes. These nodes are divided into data sources and data sinks that request information from the sources. Since the cache observes all traffic that is sent between sources and receivers, the strategy can observe all information that is to find the optimal strategy for the whole network. As discussed in Section 2.1.2, the traffic will follow some generated pattern that is sometimes disrupted by random requests that imitates human interaction. Whenever a request from is issued, the requested item can be in two states. Either the requested information is stored in the central cache or the request has to be forwarded to the source. If the item is cached, the cache can answer the request immediately without having to forward it to the source of the information. The amount of requests during a time epoch T,can be denoted by N_T. It can differ depending on T but will also follow a pattern. When data passes through the cache, the new strategy needs to determine if the incoming information should be cached or not. Furthermore, if the cache is full, the strategy needs to decide which item or items should be removed from the cache in order to fit new information.

The new strategy that manages the cache needs to learn the observed pattern in order to maximize the cache hit ratio (CHR_T) in a given time epoch T. The cache hit ratio is defined as

$$CHR_T = \frac{\text{requests answered from cache in epoch } T}{N_T}$$

For simplicity it is assumed that the consistency of items in the cache is ensured and a mechanism ensures that items in the cache are kept up to date.

4.3 Learning from observed patterns

As discussed in Section 4.2 the cache replacement strategy that manages the cache needs to analyze the access patterns to maximize the CHR by predicting future requests. These patterns can be derived from standard metrics like recency and frequency of access. In

addition to these features, the strategy also has access to semantic information about the stored content that can also be used to learn the data patterns.

Features are extracted from different sources that are collected for each data point. Depending on the chosen features, the learned model can differ, even when using the same data set. In the following sections relevant features are identified and different machine learning approaches that can be used to maximize the cache hit ratio are introduced.

4.3.1 Features

Similar to the standard cache replacement strategies the new approaches have a variety of features available that can be used to detect underlying access patterns. The features are recency, frequency of access and size of the stored information. Since the scenario considered revolves around IoT networks, the proposed strategies also have access to additional semantic features. These include the source and data type of the stored information. Depending on the underlying implementation, the semantic features also include information about device type and location of the source node. Since the receiver of the is often not known for the node serving the request, this feature is also not considered in the proposed approaches. The available features are

- Frequency of access
- Recency of last access
- Data source
- Size of data
- Time received
- Data type
- Device type of source
- Location of source

The next section explains how these features can be used to learn a model that maximizes the CHR of the strategy.

4.3.2 Learning the model

The previously introduced features should now be utilized by to detect underlying patterns in information accesses. To this end, two approaches are discussed that make

use of the available information in order to make predictions on future requests. These predictions are then used to determine an efficient strategy that can compete with the performance of standard approaches like LRU and LFU.

Clustering

The first approach that is proposed makes use of clustering. This strategy makes its prediction based on the content of the items in the cache. Since the algorithm can make use of the additional semantic features, a similar approach to [23] can be taken. By using features like source type, source location and information type it is possible to perform content aware caching. The clustering in this case is used to determine popular information based on the information that has been recently requested and is therefore also stored in the cache. By clustering the cached information, the strategy can determine a popularity index for each stored item. By using the different features, the approach can therefore detect popular areas of the network, device types or data types that are accessed more frequently than others. This also exceeds the approach implemented in [75] which could only determine popular areas of the network based on the history of the last k accesses. Since the algorithm also makes use of the currently stored items, it can adapt to changing access patterns.

In order to determine the least important item, the following steps need to be executed: First, the available features for each item need to be extracted. These features can then be used for clustering, which groups items based on different categories. The membership of an item in a cluster then determines its importance. If an item is part of a cluster that gets accessed often it is considered more important than if it is part of a cluster that gets accessed less often. The clustering strategy therefore discards the item with the lowest access frequency from the cluster that gets the lowest number of hits.

A function that implements this behavior consists of three parts. The first part is responsible for extracting all available features for each item in the cache and storing them in a table that can be used for clustering. The second part then performs clustering on the created table. The third and final step of the process is to determine the least important item from the results of the clustering. Therefore, the relative hit frequency for each cluster needs to be calculated and the item that gets accessed least in the least popular cluster gets discarded.

The computationally most expensive step of this strategy is the clustering. Therefore it is necessary to choose the appropriate algorithm for this step such that the strategy can run at a reasonable speed on resource limited nodes. Since the features are categorical

it is necessary to choose the algorithm accordingly. In order to computations and to keep the required computations to a minimum, a partitional clustering algorithm should be used. Therefore, this design makes use of *k-modes* which is discussed in Section 2.6. *k-modes* requires the k for the amount of clusters it needs to find to be specified. The clustering strategy determines the amount of clusters k it should form by considering two values: The first value considered is the amount of items in the cache. The second value that is used to determine k is the amount of different categories for each feature. In order to ensure that not too many clusters are built, the strategy uses a threshold between some fraction $\alpha \in (0, 1]$of the cache size C and the maximum amount of categories c for a feature f. Therefore

$$k = \min(C\alpha, \max_{f}(\mid c \mid))$$

The clustering should be done with different sets of features, depending on which features are available. The strategy only makes use of semantic features that are introduced in ICNs. The reason for this is that information like recency and frequency of access change for the requested item when a cache hit occurs. Therefore it would be necessary to recalculate the clusters in order to represent that change. Since cache hits do not have an impact on the semantic information of the stored item, this decision saves computational resources as reclustering is not necessary. In addition to that the clustering may find that many items in the cache are not accessed frequently, putting them together into a large cluster.

In the second step *k-modes* determines the clusters for each items. In order to find the clusters that are accessed the most and therefore contain the most important items, the relative frequency for each cluster is calculated. The strategy then chooses the least important cluster and discards the item accessed the least

The resulting process can also be found in Algorithm 1.

Therefore, when the same strategy is employed on different nodes of a network it will cache different items that are considered optimal on the specific node.

There are also some issues that need to be addressed when using the strategy in the final design. The first issue that needs to be considered it that it is possible that some large persistent clusters are built. This can lead to a similar problem that is addressed in LFU where items that were once popular remain in cache for a long period of time even though they are not accessed any more. In order to solve this problem it is necessary to remove items that were not accessed after a while even when they are still part of larger clusters. Another issue is the determination of k. In the worst case, the amount of

Data: Current cache
Result: Least important item in cache
feature_table ={};
clusters={};
for *item in cache* **do**
| add features of item to feature_table
end
max_features = maximum amount of categories for feature in feature table;
k = min(cache_size/2, max_features);
clusters = k-modes(k, feature_table);
for *cluster in clusters* **do**
| calculate relative frequency of cluster
end
c_min = cluster with lowest relative frequency;
evict = item with lowest access frequency in c_min;
return evict

Algorithm 1: Cache replacement strategy realized with clustering

clusters is always half of the cache size. This can slow the strategy down depending on the available processing power of the node. Therefore, a maximum of number clusters that can be used need to be set in accordance with the cache size as there may be a more optimal way to determine the best k. Finally it needs to be said that this strategy only makes implicit use of the observed patterns as it only considers items that are already cached. In addition to that, the strategy does not use an optimal long term strategy but only adapts to patterns that are observed in the recent past.

Reinforcement Learning

An approach that actively considers every request that is observed can be implemented with the use of reinforcement learning. When using RL, the agent that is being trained is responsible for managing the cache on the node. Based on the requests that are received on that node, the agent is tasked to learn an optimal long term strategy. Since all traffic in the considered scenario is observed at the cache, the agent can optimize its policy towards the general pattern that occurs in the network. Since the overall goal of the cache replacement strategy is to maximize the CHR, the goal of the agent is also to serve as many requests as possible from the cache. Therefore the reward function for the agent is chosen accordingly. It rewards the agent whenever it can serve a request and penalizes when the agent has to (re)fetch items from the source.

Since the underlying MDP is not known and differs depending on the network setup and and observed pattern it is necessary to choose a model free approach. This is also necessary to lean a model that adapts to the patterns at the local node.

When the agent observes a request and the cache is full it needs to decide which item should be replaced. Therefore, the action space $\mathcal{A}$ consists of n actions $a_0, ...a_{n-1}$ that represent the removal of a single or combination of items from the cache. Since the items stored in the cache often have different sizes it may be necessary to free up a combination of smaller items that are cached. Since the removal of many combinations of items is possible to free up the required space, the size of the action space is large. Depending on the state the agent is in, a large portion of the action space needs to be considered for determining the optimal action. Due to the amount of actions that need to be considered this step is computationally expensive, regardless of the algorithms used. The value of each action depends on the observed pattern and determines which item is removed. Since the agent gets penalized for every refetching of files it will chose an action that minimizes the amount of refetching operations in the future.

In order to reduce the computational power that is needed to explore the action space a similar approach as in [74] is chosen. Therefore, the agent is implemented by using an similar structure to the one discussed in Section 2.6.5.

The agent needs to perform three steps in order to determine the item that should be replaced. The first step is to determine the *proto-action*. As mentioned before, this step is the computationally most expensive step due to the amount of actions that need to be considered. In order to make reduce the amount of calculations necessary, the Deep Deterministic Policy Gradient (DDPG) algorithm is used to suggest an optimal action. Since the suggested action will most likely not be part of $\mathcal{A}$ the second step is to choose the k most similar actions from the action space. In the third step then to calculate the value of each of the k closest action to choose the most valuable one to perform. After the selection of the potential action is done, the final step is to choose the most suitable action that is performed in the end.

The resulting policy of this process then represents a near optimal long term strategy that can be used to manage the cache.

The result is an agent that tries to learn an optimal policy based on the requests it observes and therefore a strategy that adapts to the observed patterns. Since the agent learns based on a model free strategy, it is possible to deploy it at different places in the network such that the local agent finds an optimal policy for the specific node. Since every node in the network observes different patterns, the agents on different nodes learn different strategies that become optimal in the local environment. In addition to that

the agent is forced to adapt to changing patterns. Even though it is forced to adapt to the changes, this change does not happen as quickly as with the clustering strategy as the convergence of the policy takes longer.

4.4 Summary

In this Chapter, two cache replacement strategies that make use of machine learning are described. One strategy makes use of the semantic features introduced in ICNs in order to determine popular items based on these features. The popularity of an is determined by using a combination of *k-modes* and the relative frequency of the items in a cluster to determine the least popular cluster. From this cluster, the strategy then discards the least frequently accessed item. By using this approach, the clustering strategy tries to find an optimal short term strategy. The second approach makes use of reinforcement learning. By using a reward function that penalises the agent that manages the cache in case of a cache miss it learns an optimal long term strategy that maximizes the CHR based on the observed patterns.

In the following section it is outlined how the clustering strategy can be implemented.

Chapter 5

Implementation

In this section it is discussed how the clustering approach proposed in Chapter 4 can be implemented. In order to implement a new cache replacement strategy, different things need to be considered.

One thing, especially when evaluating the strategies is the environment in which the cache should be used. Since the mechanisms considered in this thesis have the goal of improving the performance of caching in IoT networks it is necessary to implement them in a corresponding environment. The setup therefore can be a IoT network or a simulation that imitates the desired environment.

Since IoT networks differ in size and structure depending on their tasks it is reasonable to simulate the environment instead of using a single real IoT network. Therefore the implementation consists of two parts:

1. Implementation of the environment.
2. Implementation of the cache replacement strategies.

5.1 Programming Language and Machine Learning Frameworks

In order to implement both the environment and the clustering strategy different programming languages can be considered. In this case, both the *C* and *Python* programming language are considered to realize both tasks.

Feature	C	Python
Access to main memory	✓	
Networking features	✓	✓
ML packets		✓
Efficient cache operations	✓	

Table 5.1: Comparison of language features between C and Python

C is considered a low level programming language as it offers the possibility to implement the operations that are needed to efficiently manage a cache. Since the cache is simulated, the advantage of *C* can by neglected. In addition this feature comes at the cost of complexity of the program since the according functionality needs to be implemented. Even though *Python* does not offer the functionality to directly access the main memory, it has modules that can be used to imitate a cache. This has the advantage that caching can also be used in a framework used for simulating the network. Both *C* and *Python* have extensive support to implement networking functionalities. These are needed in order to simulate the IoT environment for the caching mechanisms.

The final important feature that needs to be considered is the availability of ML frameworks. Since ML is a key functionality of the clustering strategy, it is desirable to make use of existing frameworks that efficiently implement ML algorithms. In *Python* several frameworks that implement ML algorithms are available. Two of the most frequently used frameworks are *TensorFlow*(TF)[2] and *PyTorch*[51] which are widely used for ML applications in different fields. Even though *C* also has libraries that implement ML algorithms, these libraries are not as well maintained as TF and PyTorch which are actively being developed.

Although the strategies can be implemented more efficiently in *C*, the advantages of *Python* with regards to ML outweigh the advantages of *C*. Therefore *Python* is used for both simulating the network and implementing the clustering strategy.

5.2 IoT network simulation

In order to simulate the environment of IoT networks for caching purposes it is possible to either implement the networking functionality or to make use of frameworks that have been developed for this purpose.

A framework needs to fulfill several requirements regardless of implementation. It needs to be able to simulate different nodes that can be connected to each other. Each node needs to have a cache that can be managed. Furthermore, the framework needs to able to

simulate different effects that can occur within networks. These include the simulation of packet loss, different connection speeds, delay or the failing of links between nodes.

Instead of implementing all this functionality, it is also possible to use existing frameworks. There are several available for this purpose that are written in different programming languages. One of these frameworks is ICARUS [57] which is described in the following section.

5.3 ICARUS

To simulate the IoT environment, the ICARUS framework [57] is used. ICARUS is designed for the purpose of simulating, testing and evaluating caching strategies in the context of ICNs. In order to use the framework in this thesis some adaptions to ICARUS are needed. Since the ICARUS is also implemented in Python, and has been implemented with future extensions in mind, the structure of the code makes it possible to be adapted in a straightforward way.

The functionality of the framework has been separated into different modules that serve a certain purpose.

In particular there are modules for:

- Specifying network topologies
- Implementing cache replacement strategies
- Implementing the communication pattern that occurs during a simulation

Each simulation that is executed in ICARUS consists of four steps:

1. Scenario generation
2. Experiment orchestration
3. Experiment execution
4. Result collection and analysis

In particular, to implement a new cache replacement strategy, a new extension of the *Cache* class needs to be implemented. This class needs to override the default operations as well as implement new functions to achieve the desired behaviour. After the class has been extended to implement the clustering strategy it can be used like the other implemented strategies in future simulations.

The scenario used for the simulation is made up of a network topology, workload and a content placement strategy. In addition to that it is possible to specify the properties of each cache during an experiment. The properties specified for each cache are the size, the decision strategy and replacement strategy.

5.4 Extension of ICARUS

In order to implement the clustering strategy within ICARUS, some adaptations need to be made.

5.4.1 Workloads and Content placement

The first adaption that is implemented is the addition of semantic features to both the workload and content placement as the standard implementation in ICARUS addresses content by unique integers. This is necessary to enable the clustering strategy to perform well on the given features.

The semantic features implemented replaces the integers by using a unique identifier for the data that is placed on each node. The information can be accessed by using a request of the structure *nodeID/content type*. The nodeID is a unique identifier for every source node of the network. The content type is the actual information that is supposed to be addressed. In this implementation the content types are identified by *value0, ..., valueN-1* for N different types of content. To ensure that the content can be requested it needs to be placed on the specific nodes. The naive approach is to place every content type on every node. This ensures that whenever a request for some content is issued the receiver of the request will get a result. When evaluating the cache strategies it is necessary to consider that it is not realistic that every node stores every type of data. It is sufficient to implement this constraint in the workload since the placement of content does not affect the measurements in any way.

In order to emulate the periodicity and other patterns described in Section 2.1.2, the corresponding workloads need to be implemented. The emulation of the periodicity is needed to evaluate how the clustering strategy performs when dealing with typical IoT traffic. Events during a simulation are generated by *workloads*. An event of a workload consists of an *event time*, a *receiver* which triggered the request and a *query for the content* that is requested.

By using this adaptions it is possible to implement the clustering strategy proposed in Chapter 4.

5.4.2 Topologies

In addition to the different workloads it is also necessary to add the considered network topologies to ICARUS.

The scenario described in section 4.2 requires the implementation of four types of setups:

(**A**) a single receiving node and a single data source

(**B**) a single receiving node and multiple data sources

(**C**) multiple receiving nodes and a single data source

(**D**) multiple receiving nodes and multiple data sources

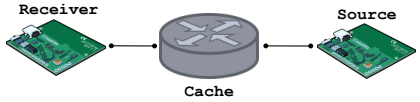
Setup A is a topology which has the cache located in between the source and receiver (*A*). It can be described as a simple path topology. This scenario is not realistic since in most cases there are either multiple data sources (**B**) or multiple data receivers (**C**). Finally, the most common occurrence is a combination of n receiving nodes and m source nodes (**D**). These four scenarios can also be seen in Figure 5.1.

5.5 A new cache replacement strategy

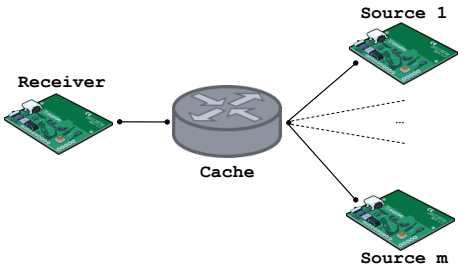
In this section the implementation of the clustering strategy is described. To add a new cache replacement strategy to ICARUS it is necessary to extend the *policies* module in the source code. The clustering strategy needs to extend the basic cache class in order to be available in simulations. The basic functionality that is required to be implemented deals with basic operations to add items to the cache or to get or remove items from the cache. This can also be seen in the class's UML diagram in Figure 5.2.

In addition to this basic functionality it is necessary to implement the cluster based cache replacement strategy that is specified in Section 4.3.2.

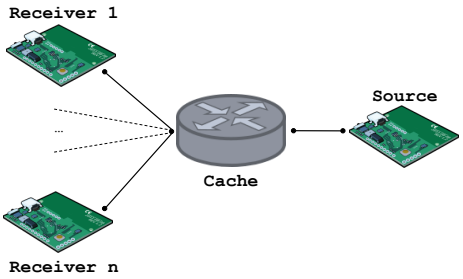
In order to perform clustering on the semantic features, the strategy needs a way to extract those features from the stored items. The workloads used for the simulations issue requests for content by using the *nodeID/content type* scheme. From this scheme, the clustering strategy can extract the features for further processing by parsing them from the request. In case a mapping from nodeID to either a location or device type exists, these features can be derived from this scheme as well. Since items of the same content type also often have the same size the same holds for a mapping from content type to size if that feature should be considered in the strategy. By extracting every feature from the item this implementation does not need to store the additional



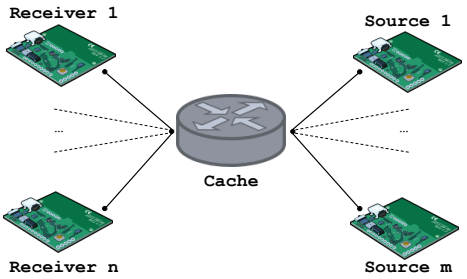
(a) Scenario (**A**)



(b) Scenario (**B**)



(c) Scenario (**C**)



(d) Scenario (**D**)

Figure 5.1: The four considered setup scenarios

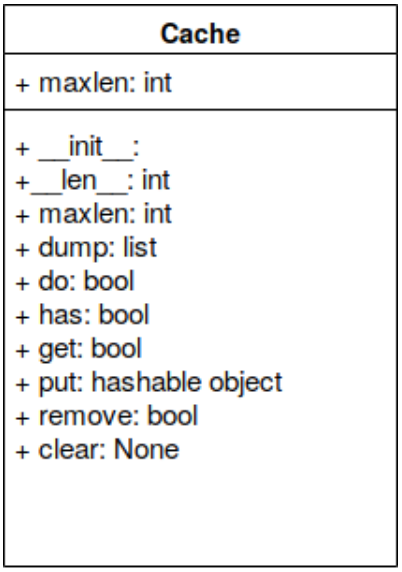


Figure 5.2: UML diagram of the base cache class

information on the node. This saves memory on the often resource limited nodes of the IoT at the cost of a slowdown of the strategy as it needs to extract the features every time when an item is supposed to be removed. In addition to that, since the strategy uses the frequency of access to an item, the cache needs to maintain a data structure that keeps track of how often an item is accessed.

Here this is realized by using a dictionary that stores the frequency of access along with the key.

When a new item is added to the cache and another item needs to be removed the proposed clustering approach is implemented. In the first step, the keys are parsed in order to extract the features. They are stored along with the frequency in a pandas dataframe that is then used for the *k-modes* clustering. *K-modes* uses the features *nodeId*, *content type* and the *full request* for clustering. As mentioned in Section 4.3.2 the amount of clusters that *k-modes* should form for needs to be determined. The k has an upper limit of half of the cache size and a lower bound of the maximum amount of unique categories for each feature. After the k is determined it can be used by the *k-modes* algorithm to process the feature table. The results from this step are appended to the feature table for further processing. The next step is to calculate the relative frequency for each cluster. To obtain these values the function groups the clusters and gets the mean of frequency values in each cluster. It then uses the result to get all items from the feature table that are part of the cluster with the lowest mean. From this set of items, the function then chooses the item with the lowest frequency of access to be evicted.

In conclusion it can be said when an item is added to the cache when it is full, the following procedure is triggered:

1. Items in cache are retrieved
2. For each item the features are extracted and stored along with their frequency of access in a pandas dataframe
3. The value k is determined based on the features and the cache size
4. The dataframe is used by *k-modes* to determine the k clusters
5. The result of step 4 is appended to the dataframe
6. The dataframe is then processed to calculate the relative frequency for each cluster
7. The function extracts the items of the cluster with the lowest relative frequency
8. From these items, it chooses the item with the lowest access frequency

9. The chosen item is removed from the cache

5.6 Summary

The new cache replacement strategy is implemented in the ICARUS framework. In order to make use of semantic features in the strategy, the framework is extended by some rudimentary features. To be able to test the clustering strategy in the environment of typical IoT patterns, the according workloads are also added to ICARUS. Finally, the framework is also extended by the clustering strategy. This strategy makes use of pandas dataframes to process the extracted features and determine the item that should be evicted efficiently.

Chapter 6

Evaluation

After the new strategies have been implemented it is reasonable to see how they compare to the already existing strategies. Therefore, the new strategy is evaluated with regards to different metrics to measure the performance.

6.1 Evaluation Metrics

As in Chapter 2, there are different metrics that can be applied to measure the performance of cache replacement strategies. The most important metric is the Cache Hit Ratio (CHR), which should be optimized in order to maximize the benefit of the system. In addition to that, the latency is also measured. Since in every setup there is pre-set delay for each request that cannot be answered from the cache, the latency is another indicator that shows how well the strategies work. As mentioned in Chapter 2, caches can also reduce the amount of messages in the network. Therefore, the link load for each connection between the data sources and receivers is also measured to show how much traffic is generated when using the different strategies.

6.2 Measurement Setup

In order to test the success of the new methods in comparison to LRU, the random and perfect/in-cache LFU strategies are evaluated in ICARUS. This is done by simulating different setups and workloads and measuring with respect to the previously introduced metrics.

In the following the different topologies and workloads used for the setups are explained.

6.2.1 Topologies

Since strategies can perform differently depending on how the network is set up, different network topologies are simulated as these setups influence which traffic is observed at the cache. Depending on the observed traffic, different features on the semantic layer are more useful for clustering than others. An example for this are topologies of setup type (**A**) and (**C**) where the source of the content is always the same. In order to find out how the setups affect the performance of caches in the simulations the experiments are repeated for each topology type.

6.2.2 Workloads

Similar to the effects of different topologies, different types of workloads also have an impact on the performance of cache replacement strategies. Therefore, the performance of the new algorithms is also evaluated using different workloads.

The first workload considered for the simulation is an adaption of the existing 'Uniform' workload. It assumes that all content types are available on any source node. Events for requested content are created according to a *Zipf*-Distribution. The Zipf-Distribution describes how content on the Internet is accessed by a fixed userbase [17]. In particular, it describes the phenomenon that some content becomes more popular than other for some period of time. Even though this scenario is unlikely to happen in an IoT setup, it shows if the clustering strategy can detect and adapt to changes in the popularity of content. In addition, this simulation then also gives an indicator of how well the clustering strategy performs in a traditional web environment.

To simulate the discussed type of behavior new workloads that also incorporate periodicity of messages as well as dependencies from certain traffic are implemented. This also includes content which is not available on all sources.

6.3 Simulations

The topologies and workloads discussed in the previous section can be used to simulate different scenarios. Each scenario is made up of a combination of a workload, a topology, a cache replacement strategy and different cache sizes. In this prototype, the size relates to the amount of items that can be stored in the cache while disregarding the size of the

Scenario	Setup type	Cache Sizes	Workload	Strategies
1	(**A**)	60, 120, 300, 1500	Semantics	LRU, RAND, CLUSTERING, LFU
2	(**B**)	60, 120, 300, 1500	Semantics	LRU, RAND, CLUSTERING, LFU
3	(**C**)	60, 120, 300, 1500	Semantics	LRU, RAND, CLUSTERING, LFU
4	(**D**)	60, 120, 300, 1500	Semantics	LRU, RAND, CLUSTERING, LFU
5	(**A**)	60, 120, 300, 1500	Periodic	LRU, RAND, CLUSTERING, LFU
6	(**B**)	60, 120, 300, 1500	Periodic	LRU, RAND, CLUSTERING, LFU
7	(**C**)	60, 120, 300, 1500	Periodic	LRU, RAND, CLUSTERING, LFU
8	(**D**)	60, 120, 300, 1500	Periodic	LRU, RAND, CLUSTERING, LFU

Table 6.1: Scenarios used for simulations

items. In order to find out how the cache size affects the performance of the algorithm, each setup is run in different cache sizes. A simulation is done in two phases. The first phase is the warmup phase in which a certain amount of requests is made to populate the cache with items. In the second phase - the measurement phase - more events are processed. Contrary to the warumup phase the amount of cache hits and misses are recorded and used for the evaluation of the performance. After the second phase is done, the results are stored in an external file.

In Table 6.1 the different scenarios are listed. In the first column the number of the experiment is listed followed by the setup type classification according to the previously introduced classes of setups. The third column then describes the cache sizes used for the experiment followed by the workload in the fourth column. In the last column the cache replacement strategies that are tested and compared are written down. The configuration for each experiment specifies 30000 requests in the warumup phase and 60000 requests for the measurement phase. Finally, the configuration also specifies that there are 30000 different content types that can be addressed.

As it can be seen in the table, each setup type is combined with different cache sizes, workload and different cache replacement strategies. Since every experiment is executed twice, this results in 2*4*4*2*5 = 320 experiments that are simulated. These can be distinguished into eight classes of scenarios that can be seen in Table 6.1.

6.4 Results

Before the actual results from the simulations are shown in Section 6.4.2, the expected performance of the new strategies is laid out in Section 6.4.1.

6.4.1 Expected results

In scenario 1 the clustering approach is expected to perform similar to a random strategy, caused by the fact that the algorithm cannot establish clusters based on the given features. Having only a single node available as data source, this feature cannot be used to classify data. In addition to that, all semantic data types differ, which results in k different clusters, where k is the amount of items in the cache. Therefore no clusters can be established and thus making the approach ineffective.

A similar behavior is expected in setups of type (**C**) in which also only a single data source is present. In this case, the algorithm could become more accurate if the receiving node would also be used as feature in the clustering process which is not implemented in this prototype. One could also argue that in a real setup, the actual receiver of the data is not known to the node storing the data as requests are only forwarded. This rules out the receiver of the data as feature for clustering. Finally it is important to note that this behavior is independent from the workload since it is caused by the way the clustering is configured and the setup type.

Since the receiving nodes can not be distinguished by the cache it is expected that the strategies perform similar in (**A**) and (**C**) as well as in type (**B**) and (**D**).

Another result that is expected to show is that the cache hit ratio increases when the cache size increases. Since the *k-modes* has more data that can be used to find clusters which then determine the popularity of items.

6.4.2 Simulation results

The system which was used to run the simulations has the following properties

- a 4 cores CPU (Intel Xeon E3-1245 v6 @3.70Ghz)
- 16GB DDR4 main memory
- 256GB SSD

The ICARUS framework was installed on a Debian Stretch operating system. In addition to that, the Python3.5 environment was used to run the framework.

In the following the results from the experiments listed in Table 6.1 are presented.

The first set of simulations was done with an implementation of the clustering strategy that also considered the frequency of access as a feature. Since many items had a low access frequency, these items were grouped together in large clusters. In addition, the

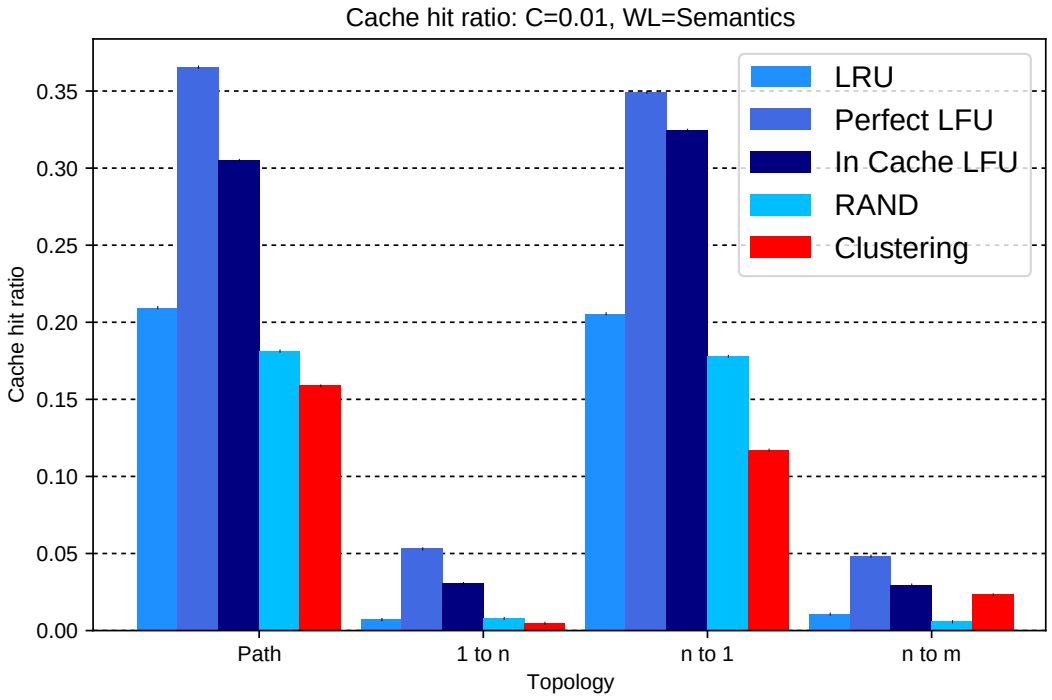


Figure 6.1: Cache hit ratio of initial cluster strategy compared to LRU,LFU and random strategy with Zipf distribution

few items with a higher frequency were grouped in smaller clusters. Since the initial implementation of the clustering strategy determined the importance of the items by the size of the clusters which they were a member of, the algorithm discarded items that were frequently accessed. Hence, the measured CHR for this implementation was lower than the CHR of any other used strategy, regardless of the setup it was tested in. This can also be seen in Figure 6.1. The figure shows the cache hit ratio for a fixed cache size in the different scenarios. On the left, the CHR of a path or 1 to 1 connection is shown, followed by the CHR for a 1 to n setup. The third part of the figure shows the CHR for a scenario of type (**C**) in which multiple receiving nodes are connected to a single source. Finally, the fourth part on the right illustrates the performance of the compared cache replacement strategies in a setup of type (**D**) where multiple receivers are connected to multiple source nodes. It can also be seen that perfect LFU outperforms in cache LFU which in turn has a greater CHR than the LRU strategy. Another aspect of this graph shows that the strategy performs similar depending on the network topology. As expected in Section 6.4.1 the strategies have a similar performance in topologies of type (**A**) and (**C**) as well as in (**B**) and (**D**).

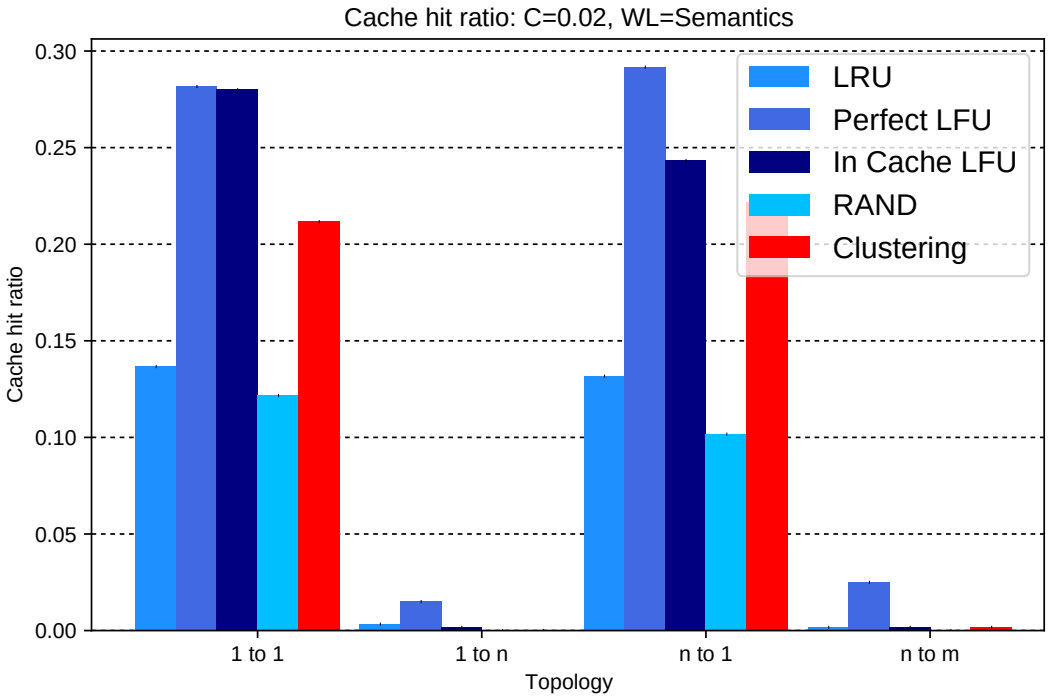


Figure 6.2: CHR of LRU, LFU, random and clustering of the improved clustering strategy

In order to improve the CHR of the clustering approach, the strategy was revised to reflect the algorithm that is described in Chapter 4. In contrast to the first implementation of the clustering strategy, the updated algorithm does not consider the size of a cluster but how often it is accessed to determine the importance of cached items. These changes to the clustering strategy lead to an increase in performance as it is illustrated in Figure 6.2. Similar to Figure 6.1 it shows the CHR of the five compared cache replacement strategies. In this comparison it can be seen that perfect LFU and in cache LFU perform better than all other strategies. Contrary to the old implementation the new clustering approach outperforms both the random and the LRU strategy in setups of type (**A**) and (**C**).

Another effect shows namely that the CHR of the algorithms increases with the size of the cache. This is illustrated in Figure 6.3, where the relation of the CHR to the cache size in a setup of type (**D**) is displayed.

In particular, the cache hit ratio of the clustering algorithm also increases along with the cache size. This indicates that the clustering strategy can successfully identify popular clusters based on the extracted features and then make decisions based on the gained information. It can be seen that the clustering strategy outperforms both the random strategy and LRU strategy in all the considered cluster sizes.

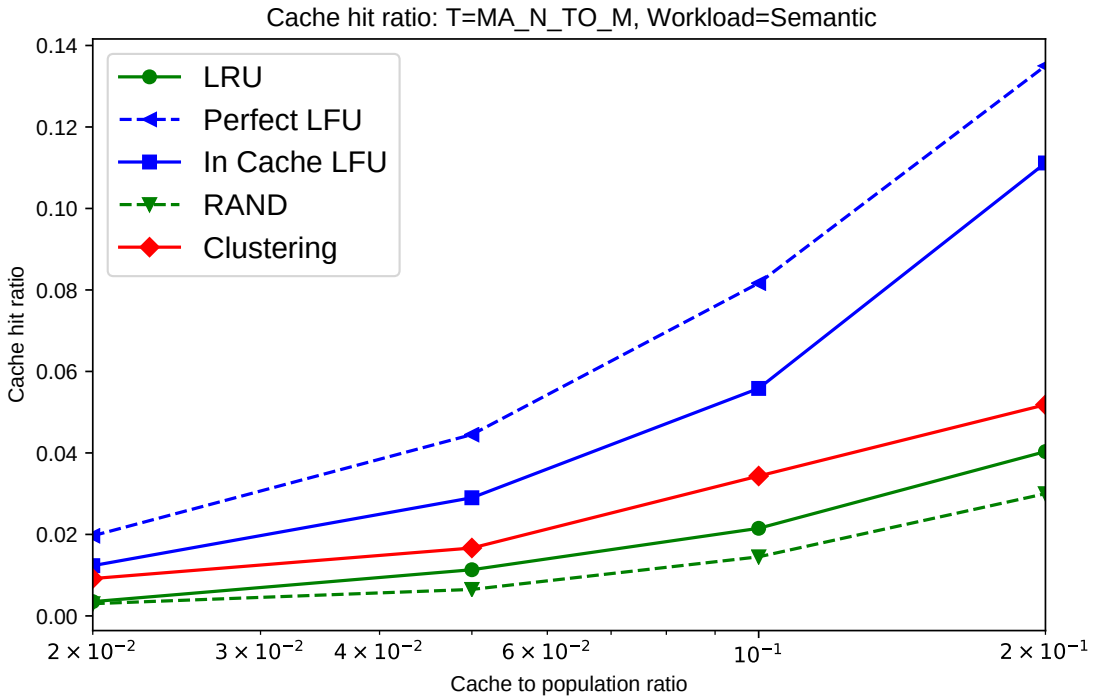


Figure 6.3: CHR in relation to the cache size

6.5 Summary

In order to evaluate the implemented clustering strategy it has been tested extensively in different simulations. After some adaptions the clustering algorithm outperformed both the random and the LRU strategy while falling back behind the LFU strategy with regards to the CHR. As expected, the performance of the clustering strategy increased with larger cache sizes, confirming that it could make decisions based on the semantic features.

Chapter 7

Conclusion

Many efforts have been made to optimize caching in different scenarios. These efforts aim at adapting widely used standard strategies to a specific use case in order to improve the cache hit ratio (CHR). This thesis it is outlined how machine learning can be used to detect access patterns in IoT networks and dynamically adapt a cache strategy to the locally observed traffic. In the following section, the findings to the question

'*How can machine learning be used to improve caching in the context of the IoT?*'

are summarized. Finally, it is discussed how these findings can be extended and improved in the future work.

7.1 Findings

Strategies used for managing caches rely on different metrics to predict which cached items are least likely to be used in the near future. Since most communication in IoT networks consists of machine-to-machine communication that is disrupted from time to time by human interaction a general access pattern occurs. In addition to that, information centric networks (ICNs) which are designed for the IoT add semantic features that can be used by caching strategies to improve their predictions. Finally, machine learning offers solutions to detect patterns in data which can then be used to make predictions. In this thesis it was shown that machine learning can be used for caching in IoT by using it to learn the access patterns from the observed communication at the cache. By using this synergy, two strategies for cache replacement strategies for IoT are adapted. The first approach uses the semantic features for *K-modes* clustering

to perform content aware caching. Therefore, the strategy is capable of determining popular content types and devices to predict which items are least likely to be accessed in the future. The implementation and evaluation of this strategy in the ICARUS framework has shown that it performs similar to standard approaches that are used in ICNs. The second proposed strategy uses reinforcement learning to improve the cache hit ratio. By introducing a reward function that penalizes a cache miss and rewards in case of a cache hit the strategy is supposed to adopt an optimal long term cache replacement strategy over time. Since the strategy learns directly from the locally observed patterns it adapts to the specific scenario that the cache is employed in. Overall it is shown that it is possible to use ML to implement new cache strategies that exploit the patterns of IoT networks.

7.2 Future Work

Since both the clustering and reinforcement learning strategies are cache replacement mechanisms, it is reasonable to research how, based on the M2M communication in IoT networks, cache prefetching strategies can be improved. Another question that arises is how the performance of the caching strategies changes when multiple machine learned caches are present in the network. In addition, when considering the clustering strategy it might be of interest which other semantic features can be used to further improve the content aware caching and if there are clustering algorithms more suitable than *K-Modes* for this task. Finally, it is necessary to implement and evaluate how the reinforcement learning supported strategy performs compared to the other commonly used approaches.

Bibliography

[1] *5 Trends Emerge in the Gartner Hype Cycle for Emerging Technologies, 2018*. https://www.gartner.com/smarterwithgartner/5-trends-emerge-in-gartner-hype-cycle-for-emerging-technologies-2018/. Accessed: 2018-11-30.

[2] Martín Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[3] Ibrahim Abdullahi, Suki Arif, and Suhaidi Hassan. "Survey on caching approaches in information centric networking". In: *Journal of Network and Computer Applications* 56 (2015), pp. 48–59.

[4] Marc Abrams et al. "Caching proxies: Limitations and potentials". In: (1995).

[5] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. "A survey of web caching and prefetching". In: *Int. J. Advance. Soft Comput. Appl* 3.1 (2011), pp. 18–44.

[6] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. "Intelligent Web proxy caching approaches based on machine learning techniques". In: *Decision Support Systems* 53.3 (2012), pp. 565–579.

[7] Marica Amadeo et al. "Named data networking for IoT: An architectural perspective". In: *Networks and Communications (EuCNC), 2014 European Conference on*. IEEE. 2014, pp. 1–5.

[8] Martin Arlitt et al. "Evaluating content management techniques for web proxy caches". In: *ACM SIGMETRICS Performance Evaluation Review* 27.4 (2000), pp. 3–11.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey". In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[10] Emmanuel Baccelli et al. "Information centric networking in the IoT: experiments with NDN in the wild". In: *Proceedings of the 1st ACM Conference on Information-Centric Networking*. ACM. 2014, pp. 77–86.

[11] Greg Barish and Katia Obraczke. "World wide web caching: Trends and techniques". In: *IEEE Communications magazine* 38.5 (2000), pp. 178–184.

[12] Richard Bellman. "Dynamic programming". In: *Science* 153.3731 (1966), pp. 34–37.

[13] Manuele Bicego, Vittorio Murino, and Mário AT Figueiredo. "Similarity-based clustering of sequences using hidden Markov models". In: *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer. 2003, pp. 86–95.

[14] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[15] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. "A training algorithm for optimal margin classifiers". In: *Proceedings of the fifth annual workshop on Computational learning theory*. ACM. 1992, pp. 144–152.

[16] Léon Bottou and Olivier Bousquet. "The tradeoffs of large scale learning". In: *Advances in neural information processing systems*. 2008, pp. 161–168.

[17] Lee Breslau et al. "Web caching and Zipf-like distributions: Evidence and implications". In: *Ieee Infocom*. Vol. 1. 1. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE). 1999, pp. 126–134.

[18] Pei Cao et al. "A study of integrated prefetching and caching strategies". In: *ACM SIGMETRICS Performance Evaluation Review* 23.1 (1995), pp. 188–197.

[19] Carlos Carvalho. "The gap between processor and memory speeds". In: *Proc. of IEEE International Conference on Control and Automation*. 2002.

[20] Anawat Chankhunthod et al. "A Hierarchical Internet Object Cache." In: *USENIX Annual Technical Conference*. 1996, pp. 153–164.

[21] Martin Dräxler and Holger Karl. "Efficiency of on-path and off-path caching strategies in information centric networks". In: *2012 IEEE International Conference on Green Computing and Communications*. IEEE. 2012, pp. 581–587.

[22] Gabriel Dulac-Arnold et al. "Deep reinforcement learning in large discrete action spaces". In: *arXiv preprint arXiv:1512.07679* (2015).

[23] Mohammed S ElBamby et al. "Content-aware user clustering and caching in wireless small cell networks". In: *arXiv preprint arXiv:1409.3413* (2014).

[24] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Kdd*. Vol. 96. 34. 1996, pp. 226–231.

[25] Frieder Ganz et al. "A resource mobility scheme for service-continuity in the Internet of Things". In: *Green Computing and Communications (GreenCom), 2012 IEEE International Conference On*. IEEE. 2012, pp. 261–264.

[26] Robert B Gramacy et al. "Adaptive caching by refetching". In: *Advances in Neural Information Processing Systems*. 2003, pp. 1489–1496.

[27] Shixiang Gu et al. “Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic”. In: *International Conference on Learning Representations 2017*. OpenReviews. net. 2017.

[28] Jayavardhana Gubbi et al. “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future generation computer systems* 29.7 (2013), pp. 1645–1660.

[29] Mohamed Ahmed M Hail et al. “On the performance of caching and forwarding in information-centric networking for the IoT”. In: *International Conference on Wired/Wireless Internet Communication*. Springer. 2015, pp. 313–326.

[30] Wolfgang Karl Härdle et al. *Nonparametric and semiparametric models*. Springer Science & Business Media, 2012.

[31] John A Hartigan and Manchek A Wong. “Algorithm AS 136: A k-means clustering algorithm”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[32] Yih-Chun Hu and David B Johnson. “Caching strategies in on-demand routing protocols for wireless ad hoc networks”. In: *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM. 2000, pp. 231–242.

[33] Zhexue Huang. “Extensions to the k-means algorithm for clustering large data sets with categorical values”. In: *Data mining and knowledge discovery* 2.3 (1998), pp. 283–304.

[34] Zhexue Huang and Michael K Ng. “A fuzzy k-modes algorithm for clustering categorical data”. In: *IEEE Transactions on Fuzzy Systems* 7.4 (1999), pp. 446–452.

[35] Adam Jacobs. “The pathologies of big data”. In: *Queue* 7.6 (2009), p. 10.

[36] Shudong Jin and Azer Bestavros. “Popularity-aware greedy dual-size web proxy caching algorithms”. In: *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. IEEE. 2000, pp. 254–261.

[37] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[38] Vijay R Konda and John N Tsitsiklis. “Actor-critic algorithms”. In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[39] Hermann Kopetz. “Internet of things”. In: *Real-time systems*. Springer, 2011, pp. 307–323.

[40] Bin Lan et al. “Making web servers pushier”. In: *International Workshop on Web Usage Analysis and User Profiling*. Springer. 1999, pp. 112–125.

[41] Timothy P Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).

[42] Thodoris Lykouris and Sergei Vassilvitskii. “Better Caching with Machine Learned Advice”. In: (2018).

[43] Mohammad Mahdian, Hamid Nazerzadeh, and Amin Saberi. “Online optimization with uncertain information”. In: *ACM Transactions on Algorithms (TALG)* 8.1 (2012), p. 2.

[44] John Mellor-Crummey, David Whalley, and Ken Kennedy. “Improving memory hierarchy performance for irregular applications using data and computation reorderings”. In: *International Journal of Parallel Programming* 29.3 (2001), pp. 217–247.

[45] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[46] Alexandros Nanopoulos, Dimitris Katsaros, and Yannis Manolopoulos. “Effective prediction of web-user accesses: A data mining approach”. In: *Proceedings of the WEBKDD Workshop*. 2001.

[47] Kyle J Nesbit and James E Smith. “Data cache prefetching using a global history buffer”. In: *Software, IEE Proceedings-*. IEEE. 2004, pp. 96–96.

[48] Dusit Niyato et al. “A novel caching mechanism for Internet of Things (IoT) sensing service with energy harvesting”. In: *Communications (ICC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–6.

[49] Venkata N Padmanabhan and Jeffrey C Mogul. “Using predictive prefetching to improve world wide web latency”. In: *ACM SIGCOMM Computer Communication Review* 26.3 (1996), pp. 22–36.

[50] Marc-Oliver Pahl, Georg Carle, and Gudrun Klinker. “Distributed smart space orchestration”. In: *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE. 2016, pp. 979–984.

[51] Adam Paszke et al. “Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration”. In: *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration* 6 (2017).

[52] Stefan Podlipnig and Laszlo Böszörmenyi. “A survey of web cache replacement strategies”. In: *ACM Computing Surveys (CSUR)* 35.4 (2003), pp. 374–398.

[53] Ioannis Psaras, Wei Koong Chai, and George Pavlou. “Probabilistic in-network caching for information-centric networks”. In: *Proceedings of the second edition of the ICN workshop on Information-centric networking*. ACM. 2012, pp. 55–60.

[54] J. Ross Quinlan. “Induction of decision trees”. In: *Machine learning* 1.1 (1986), pp. 81–106.

[55] Sam Romano and Hala ElAarag. “A quantitative study of recency and frequency based web cache replacement strategies”. In: *Proceedings of the 11th communications and networking simulation symposium*. ACM. 2008, pp. 70–78.

[56] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[57] Lorenzo Saino, Ioannis Psaras, and George Pavlou. “Icarus: a Caching Simulator for Information Centric Networking (ICN)”. In: *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*. SIMUTOOLS ’14. Lisbon, Portugal: ICST, 2014.

[58] Matthias Schubert. *Lecture notes in Big Data Management and Analytics*. 2019.

[59] Samuel O Somuyiwa, András György, and Deniz Gündüz. “A Reinforcement-Learning Approach to Proactive Caching in Wireless Networks”. In: *IEEE Journal on Selected Areas in Communications* (2018).

[60] Vasilis Sourlas et al. “Autonomic cache management in information-centric networks”. In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE. 2012, pp. 121–129.

[61] Jim Summers et al. “Characterizing the workload of a Netflix streaming video server”. In: *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 1–12.

[62] Csaba Szepesvári. “Algorithms for reinforcement learning”. In: *Synthesis lectures on artificial intelligence and machine learning* 4.1 (2010), pp. 1–103.

[63] Linpeng Tang et al. “RIPQ: Advanced Photo Caching on Flash for Facebook.” In: *FAST*. 2015, pp. 373–386.

[64] Jia Wang. “A survey of web caching schemes for the internet”. In: *ACM SIGCOMM Computer Communication Review* 29.5 (1999), pp. 36–46.

[65] Alec Wolman et al. “On the scale and performance of cooperative web proxy caching”. In: *ACM SIGOPS Operating Systems Review* 33.5 (1999), pp. 16–31.

[66] Felix Wortmann and Kristina Flüchter. “Internet of things”. In: *Business & Information Systems Engineering* 57.3 (2015), pp. 221–224.

[67] George Xylomenos et al. “A survey of information-centric networking research.” In: *IEEE Communications Surveys and Tutorials* 16.2 (2014), pp. 1024–1049.

[68] Qiang Yang, Haining Henry Zhang, and Tianyi Li. “Mining web logs for prediction models in WWW caching and prefetching”. In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2001, pp. 473–478.

[69] Mohamed Zahran. *Lecture notes in Computer Systems Organization – Lecture 15-16*. 2015.

[70] Lixia Zhang, Sally Floyd, and Van Jacobson. “Adaptive web caching”. In: *Proceedings of the NLANR Web Cache Workshop*. Vol. 97. 1997.

[71] Lixia Zhang et al. “Named data networking”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 66–73.

[72] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. “BIRCH: an efficient data clustering method for very large databases”. In: *ACM Sigmod Record*. Vol. 25. 2. ACM. 1996, pp. 103–114.

[73] Baihua Zheng, Jianliang Xu, and Dik Lun Lee. “Cache invalidation and replacement strategies for location-dependent data in mobile environments”. In: *IEEE Transactions on Computers* 51.10 (2002), pp. 1141–1153.

[74] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. “A deep reinforcement learning-based framework for content caching”. In: *Information Sciences and Systems (CISS), 2018 52nd Annual Conference on*. IEEE. 2018, pp. 1–6.

[75] ZhangBing Zhou et al. “Periodic query optimization leveraging popularity-based caching in wireless sensor networks for industrial iot applications”. In: *Mobile Networks and Applications* 20.2 (2015), pp. 124–136.