Auto-Scaling Techniques for Spark Streaming

Master-Thesis von Seyedmajid Azimi Gehraz
Tag der Einreichung:

1. Gutachten: Prof. Dr. rer. nat. Carsten Binnig
2. Gutachten: Dr. Thomas Heinze

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Data Management

Auto-Scaling Techniques for Spark Streaming

Vorgelegte Master-Thesis von Seyedmajid Azimi Gehraz

1. Gutachten: Prof. Dr. rer. nat. Carsten Binnig
2. Gutachten: Dr. Thomas Heinze

Tag der Einreichung:

Contents

List of Figures

List of Tables

1 Abstract

2 Problem Definition

2.1 Introduction

Cloud computing has been on rise over the last decade. Parts of this popularity is due to its inherent features. It lets application developers to run their applications on virtual infrastructure. Virtual infrastructure lays the foundation of on-demand infrastructure. Developers acquire and release resources as required by workload. Examples of cloud providers are Amazon AWS [1], Microsoft Azure [34] and Google Cloud [15]. Today's cloud infrastructure is widely used by many customers for different purposes such as batch processing, serving static content, storage servers and alike.

As cloud environment brings up *elasticity* [22], it also introduces a new set of challenges and problems. Modern applications face fluctuating workloads. Typically, if a workload is *predictable*, resources are allocated ahead of time before load-spike starts. However, in many other scenarios predicting even near future workload is a not so easy task. Even though running an application in cloud environments helps to overcome a long standing problem of *over-provisioning*, low utilization is still one of the major problems of cloud applications. This has been confirmed by multiple studies [10] [36].

The root of the problem is originated from the fact that, most developers do not have enough insight about bottom and peak workload of their application. Thus, they fail to define an effective scaling strategy. Therefore, they end up with conservative strategies which in turn leads to low utilization. Hence, we need a system that automates the process of resource allocation. Auto-Scaling has been well studied in the context of web application. [18] [11] [23] are just a few examples. Chapter 8 explores more techniques and strategies.

2.2 Objectives of Auto-Scaling Systems

The ultimate goal of an Auto-Scaling system is to automate the process of acquiring and releasing *resources* in order to minimize the *cost* with minimum violation of *service level objectives* (SLO). The definition of *resource* depends on the context. As an example, for a stateless web applications it means virtual machines or containers that run web server software. For an Auto-Scaling system to adjust required resources, it shall consider different aspects of application and environment. Additionally, the term *cost* is also defined in the context. As an example, it might mean monetary cost or just numerical value of resources. SLOs are predefined rules that shall not be violated during application runtime and are also defined in the context of application.

2.3 Auto-Scaling in Data Stream Processing Systems

Data Stream Processing Systems are data processing systems that process *unbounded* stream of data unlike their *batch-oriented* counterparts. With the ever increasing adoption of IoT applications, it is critical to design stream processing systems that handles the incoming messages with high throughput and low latency. With static workloads, these problems could be solved by dominating stream processing systems like Apache Spark [3], Apache Storm [38] and Apache Flink [2]. However, the problem of low utilization also applies for stream processing systems as well. This leads us to a new generation of stream processing systems called *Elastic Data Processing Systems* that adopts elasticity concepts to stream processing system.

Prior to this thesis a number of studies have been performed on elastic stream processing system. [8], [19] are just a few samples. One of the dominating stream processing systems is Apache Spark which support both batch and stream processing. In order to support both workloads, Apache Spark has a unique architecture that partitions the input workload into predefined window of batches – an architecture known as micro batching. With this common architecture as a foundation, number of interesting challenges arise that need to be considered for elastic workloads.

This thesis will focus on dynamic resource allocation in the context of Apache Spark. An extensible framework will be developed based on a prior work by Kielbowicz [26]. This thesis will extend the existing prototype and implement multiple Auto-Scaling techniques for Apache Spark Streaming and will evaluate these techniques using real-world workloads. The ultimate goal is to identify how the architecture of Spark Streaming influences the performance of the different Auto-Scaling techniques.

2.4 Summary

As mentioned this thesis will focus on dynamic resource allocation in the context of Apache Spark. The thesis is organized as follows. Chapter 3 introduces and explains basics of Auto-Scaling techniques. Chapter 4 introduces the architecture of Apache Spark Streaming. Chapter 5 explains structure and design considerations of this thesis. Chapter 6 discusses implementation details and challenges faced during implementation. Chapter 7 evaluates the implementation under different workloads. Chapter 8 includes discussion of prior and related work. Finally, chapter 9 concludes.

3 Introduction to Auto-Scaling

3.1 Introduction

As mentioned in Chapter 2 the key characteristic of cloud environments is emphelasticity behavior. However, manually adjusting resources are is not effective approach to exploit this feature. Hence, we need to automate this procedure with minimal human intervention. This chapter introduces foundations of Auto-Scaling techniques. Different techniques and architectures will be discussed from a high level standing point. It shall be noted that, this chapter has been heavily inspired by work done by Lorido-Botran, Miguel-Alonso, and Lozano [32].

3.2 Basic Concepts

The ultimate goal of an Auto-Scaling system is to automate the process of acquiring and releasing *resources* in order to minimize the *cost* with minimum violation of *service level objectives* (SLO). However, *resource* is a broad and context-dependent term. It refers to any form processing engine that provides application developers some form of computation power. This general purpose definition is broad enough to capture different kinds. In most cases it, it means virtual machines allocated by cloud provider. In more modern distributed systems, a resource refers to *containers* like Google Kubernetes [16]. However, a resource might be as simple as a single process or thread.

The term *cost* refers to any form of expenditure that users pay in order to acquire a resource. It doesn't necessarily mean *monetary* cost. It can also refer to numerical values of resources, like number of virtual machines or number of running processes. Although minimizing cost is the ultimate goal of any Auto-Scaler system, not in all cases cost reduction is desirable. It should be achieved with respect to defined *service level objectives*.

Service Level Objectives are any predefined rules that shall be respected during application runtime. The following, defines a couple of SLO definitions for different applications:

- 99 percentile round-trip latency of requests in a web application should be less than 150 milliseconds.
- All committed records in master database must be replicated with a maximum delay of 5 milliseconds.
- All messages pushed by a producer, should be processed by respective consumers in less than 5 minutes.
- At least 95% of images published in the last 24 hours should be served by cache servers.

Service Level Objectives are typically determined and defined by business requirements. Defining effective and meaningful SLOs is a challenge on its own. However, it is out of the scope of this thesis.

From a high level point of view, Auto-Scaling is a *trade-off* amongst cost and violation of SLOs. To make an effective decision, an Auto-Scaler needs to consider application and its environment:

- **Infrastructure pricing model**. Some infrastructure providers charge customers hourly. That is, if customer acquires a resource at 10:30AM and releases it at 11:30AM, the customer is charged for two hours. Some other service providers might charge on minute basis. Pricing model has a huge impact on decisions made by Auto-Scaler system, since it makes some decisions pointless.
- **Service level objectives**. Each application has its own set of objectives that should be adhered by Auto-Scaling system. These SLOs might be defined and applied at *soft* and *hard* levels. Violating an SLO at soft level is not critical, albeit alarming. However, violating a resource at hard level is a critical issue and is a negative point for an Auto-Scaler.

- **Acquire/Release delay**. Depending on type of the resource, it might take some time for the resource to become responsive and ready to process user requests. For example, booting a virtual machine typically takes couple of minutes, whereas launching a container takes time in order of seconds. An Auto-Scaler shall consider whether acquiring and releasing a heavy weight resource in a *zig-zag* manner worth the overhead or not.

- **Unit of allocation**. In some cases, it might be beneficial to allocate multiple instances of a same resource at once. This might be due to the startup and initialization overhead or it might be the case that Auto-Scaler predicted a huge load spike in near future.

Auto-Scaling can be done with different techniques and strategies. The remainder of this chapter is organized as follows. Section 3.3 defines general architecture of an Auto-Scaler. Section 3.4 clarifies which sort of actions can be applied by an Auto-Scaler. Section 3.5 classifies different techniques and briefly explains each category. Finally, section 3.6 concludes.

3.3 Generic Auto-Scaler Architecture

Figure 3.1 illustrates generic architecture of an Auto-Scaler. This architecture is broad enough to capture different kinds of applications. First, responsibilities of different components shall be clarified. Table 3.1 summarizes all components of the system. An Auto-Scaling system typically consists of following sub-components.

Component	Description
Clients	Users of the application which might be applications on their own
Load Distributor	Distributes incoming requests to application instances
Application	Runs business logic defined by developer
Metrics Engine	Monitors and collects metric from application and provides it to other components
Infrastructure API	Provides API to adjust (acquire/release) resources
Auto-Scaler	Runs the auto-scaling algorithm based on metric collected by Metrics Engine

Table 3.1: Auto-Scaler components summary

Clients Most kinds of applications, typically have some form of client that sends requests to the system and either waits to get a reply or operates under *fire-and-forget* strategy. It shall be noted that, a client is not necessarily an end user. In today's modern distributed applications, an application might be client of another application.

Load Distributor In order to provide some degree of *transparency*, usually clients connect to a Load Distributor component. It is the responsibility of the Load Distributer that *proxies* client request to application. Load Distributor is also a generic component and might represent different technology in read-world applications. In the context of a web application, it might be an HTTP load balancer. Even a *Message Broker* can also be represented as a form of Load Distributer. It shall be noted that Load Distributor itself can be replicated or sharded for *high availability* or *scalability* reasons.

Application The application component runs the business logic. This architecture doesn't impose any limitation on application architecture. It might be a simple stateless web application. It might access to a back-end cache or database service. It might push some messages to a message broker, as a result of client request. It might be just a simple process consuming messaging provided by a message broker. It might forward client requests to other applications for further processing.

Infrastructure API Typically, when an Auto-Scaler system decides to take any action, it doesn't touch the application directly. In order to provide *separation of concerns*, this responsibility is handed over to infrastructure via an API provided by resource/service provider. An important issue that shall be noted here is that, service provider may schedule resource changes and execute them some time later. Thus, resource changes might not take effect immediately at the moment Auto-Scaler requests them. This fact shall be considered by Auto-Scaler system.

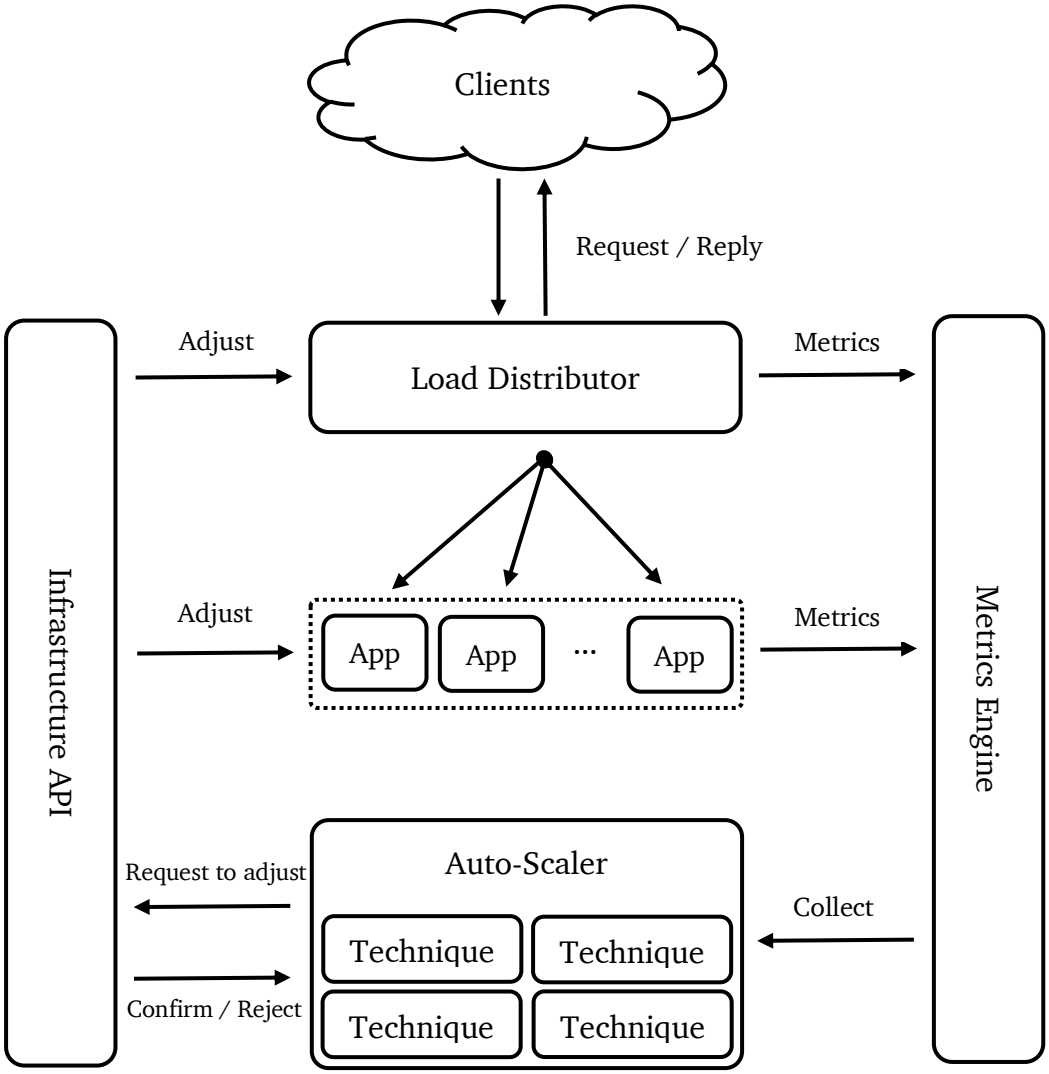

Figure 3.1: General Auto-Scaler Architecture

Metric Engine Auto-Scaler system needs to have a good insight on current status of the application and incoming requests. Metric engine – known as *monitoring engine* takes the responsibility of measuring and collecting different aspects of the application. The term *metric* refers to any form of measurable aspect of an application or its environment. Ghanbari et al. [14] has defined and proposed a list of different type of metrics that could be exploited for different purposes.

- **Hardware** dependent metrics such as CPU usage, disk access time, memory usage, network bandwidth usage, network latency.
- **Operating System** provided metrics such as CPU-time, page faults, real memory.
- **Load balancer** provided metrics such as size of request queue length, session rate, number of current sessions, transmitted bytes, number of denied, requests, number of errors.
- **Web server** provided metrics such as transmitted bytes and requests, number of connections in specific states (e.g. closing, sending, waiting, starting, . . .).
- **Application server** provided metrics such as total threads count, active threads count, used memory, session count, processed requests, pending requests, dropped requests, response time.

- **Database server** provided metrics such as number of active threads, number of transactions in a particular state (e.g. write, commit, roll-back, ...).

Since storing and reporting metrics has its own overhead, typically metrics engine aggregates collected values at different scale depending on how fresh it is. Rationally, fresh values are more important for Auto-Scaling system. As an example, it might provide near real time values for about 15 minutes. Then, for the last 5 hours, collected values are aggregated by a window of one minute. For last two days, it is aggregated by a window of 15 minutes and finally, for any record older than last two days, it is aggregated on hourly basis. Whether these aggregated values are sufficient for Auto-Scaler to make an accurate decision is out of the scope of this thesis. However, it may be a good idea to empirically adjust this system until it fits the requirements of Auto-Scaler system.

Auto-Scaler This component is the core of Auto-Scaling system. It consists of different techniques. Typically, an Auto-Scaler works in *consecutive rounds*. First, it collects metrics from Metrics Engine and offloads them to one or multiple technique implementations. It shall be noted that, an Auto-Scaler might utilize different implementation of techniques simultaneously – for different stages of the application as an example. Even it might utilize a single implementation under different configurations. This architecture does not impose any limit on the order of techniques. Then, based on some preferences or ordering mechanism, it chooses the *final decision*. Finally, it requests the Infrastructure API to change the number of resources. Naturally the assumption is that, infrastructure provider should be able to adjust the required or released resources. However, in some case it might not be able to fulfill the task. Thus, Auto-Scaler shall wait for a response to check whether the request has been successfully fulfilled or not.

For the sake of understandability, Algorithm 1 describes the above procedure in pseudo code.

3.4 Actions

In each round an Auto-Scaler decides to take an action and request that specific action to Infrastructure API. Table 3.2 summarizes feasible actions for an Auto-Scaler. This thesis, assumes three possible actions.

- **Scale-In**. This action implies that Auto-Scaler has decided to remove one or more resources. It doesn't necessarily means, Infrastructure API will be able to remove all the requested resources. Consequently, Infrastructure API might be able to remove any subset of resources among those that were requested to remove. In this thesis, we assume Infrastructure API offers such a behavior to notify Auto-Scaler about the actual removed resources either *synchronously* in response to Auto-Scaler's request or via an *asynchronous* API provided by Auto-Scaler.

- **Scale-Out**. This action implies that Auto-Scaler has decided to add one or more resources. Similar to Scale-In action, Infrastructure API might be able to fulfill the request or not. In case, it doesn't have enough resources to allocate, this incident should be reported to Auto-Scaler component. This thesis assumes this behavior. Response shall propagate back to Auto-Scaler similar to steps described in Scale-In action.

- **No-Action**. This action implies that Auto-Scaler has decided to do nothing but stay with the same number of resources as last round. This is a kind of *no-operation*.

It's noteworthy that in all cases, Auto-Scaler is allowed to store any history of actions taken so far. In fact, it is a special category of Auto-Scalers known as *stateful* Auto-Scalers. Refer to section 3.5 for further discussion and explanation on taxonomy of Auto-Scalers.

Another aspect of taking an action is that, Auto-Scaler is allowed to Scale-In/Out *horizontally* or *vertically* in each round independent of previous rounds. Horizontal Scale-In/Out refers to a category of actions that acquires or releases resources in parallel to each other. For example, in the context of a web application, adding or removing one or more virtual machines is considered as a horizontal scaling action. While on the other hand, Auto-Scaler might decide to just

Algorithm 1: General work-flow of an Auto-Scaler

```
// different implementations of techniques
implementations ← []
// decision of each implementation
decisions ← []
// final decision of Auto-Scaler
finalDecision ← null
// instantiate as many techniques as required
for i ← 0 to n do
    implementations[i] ← InstantiateTechnique(i)
end
repeat
    // load monitoring data from metrics engine
    currentMetrics ← GetCurrentMetricsFromMetricsEngine()
    // initialize decisions
    decisions ← []
    for i ← 0 to n do
        impl ← implementations[i]
        decisions[i] ← GetDecision(impl)
    end
    // calculate final decision based on some weight or ordering mechanism
    finalDecision ← GetFinalDecision(decisions)
    // request infrstructure API to adjust resources
    reply ← RequestInfrastructureAPI(finalDecision)
    // in case infrastructure reject request, warn developers
    if reply = ReplyStatus.REJECT then
        // issue a warning
        LogError("request can not be fulfilled")
    end
until Auto-Scaler is running
```

Action	Description
Scale-In	Remove/Release one or more resources
Scale-Out	Acquire/Add one or more resources
No-Action	Do nothing

Table 3.2: Summary of feasible actions

scale by adding hardware resources. For example, it might decide to add more RAM or remove couple of CPU cores in one specific virtual machine. This kind of scaling action is considered as vertical scaling action.

Actions are not necessarily applied at a constant rate. Auto-Scaler is in full charge of taking actions at *exponential* rates. For example, in consecutive rounds, an Auto-Scaler can decide to acquire 1, 2, 4, 8, 16 virtual machines per round. This also applies for Scale-Out actions. Nothing hampers an Auto-Scaler from changing rate of scaling actions in each round. It might even decide to apply different rates for different stages of the application like *startup* phase, or *near-ending* phase, etc.

Last but not least, an Auto-Scaler might decide to apply a *grace period* after taking an action independent of previous rounds. A grace period is a time frame, in which Auto-Scaler does not take any further action in order to let cluster of resources stabilize. Similar to action rates, grace period can also be applied at different rates. For example, Auto-Scaler might wait for 10, 20, 40 seconds after taking Scale-Out action in consecutive rounds.

3.5 Taxonomy of Auto-Scaling Techniques

Auto-Scalers can be modeled and classified in different categories.

3.6 Conclusion

4 Apache Spark and Spark Streaming

5 Design

6 Implementation Detail

7 Evaluation

8 Related Work

8.1 Introduction

Dynamic resource allocation in cloud environments has been studied extensively in literature. In this chapter prior work will be discussed and explored. It is organized as follows. Section 8.2 delves into threshold-based techniques. Section 8.3 investigates techniques based on time-series analysis. Section 8.4 analyses techniques based on queuing theory. Section 8.5 explores Reinforcement Learning techniques comprehensively. Finally, section 8.6 concludes.

8.2 Threshold-Based Techniques

Hasan et al. [18] proposed four thresholds and two time periods. `ThrUpper` defines upper bound. `ThrBelowUpper` is slightly below `ThrUpper`. Similarly, `ThrLower` defines lower bound and `ThrAboveLower` is slightly above the lower bound. In case, system utilization stays between `ThrUpper` and `ThrBelowUpper` for a specific duration, then cluster controller decides to take a scale-out action, by adding resources. On the other hand, if system utilization stays between `ThrLower` and `ThrAboveLower` for a specified duration, then the central controller decides to take scale-in action. Furthermore, in order to prevent making *oscillating* decisions, *grace period* is enforced. During this period, no scaling decision is made. Defining two levels of thresholds helps to detect workload *persistency* and avoids making immature scaling decision. However, defining thresholds is a tricky and manual process, and needs to be carefully done [11]. It shall be noted that, computation overhead of this approach is very low.

RightScale [37] applies voting algorithm among nodes to make scaling decisions. In order for a specific action to be decided, majority of nodes should vote in favor of that specific action. Otherwise, no-action is elected as a default action. Afterwards, nodes apply grace period to stabilize the cluster. The complexity of the voting process in trusted environments is in the order of $O(n^2)$, which leads to heavy network traffic among participants when cluster size grows. This approach also suffers from the same issue – accurately adjusting threshold values – as other threshold-based approaches.

Heinze et al. [20] proposed a novel threshold-based solution in the context of FUGU [17] – a data stream processing framework. This techniques uses an adaptive window [4] to monitor the recent changes in workload pattern. In case a change in workload is detected, optimization component is activated and fed with recent short-term utilization history. Thereafter, the optimization component determines monetary cost of current system configuration and then simulates the cost of different scaling decisions. The *latency-aware* cost function has the responsibility to calculate monetary cost of system configuration. The search function is an implementation of *Recursive Random Search* [43] algorithm which consists of two phases. First, in *exploration* phase, the complete parameter space is explored to find a solution with minimum cost. In second phase – *exploitation phase* – only specific parts of the parameter space which has been discovered in first phase, will be investigated. Kielbowicz [26] has implemented this techniques in the context of Spark Streaming. Thus, It is considered in evaluation scenarios.

8.3 Time-Series Analysis Techniques

Herbst et al. [23] surveys different auto-scaling techniques based on time-series analysis in order to forecast *trends* and *seasons*. *Moving Average Method* takes the average over a sliding window and smooths out minor noise level. Its computational overhead is proportional to size of the window. *Simple Exponential Smoothing*(SES) goes further than just taking average. It gives more weight to more recent values in sliding window by an exponential factor. Although it is more computationally intensive compared to moving average, it is still negligible. SES is capable of detecting short-term trends but fails at predicting seasons. These approaches are more specific instances of *ARIMA* (Auto-Regressive Integrated

Moving Average) which is a general purpose framework to calculate moving averages. However, time-series analysis is only suitable for stationary problems consist of recurring workload patterns such as web applications. Additionally, more advanced forms of time-series analysis which are capable of forecasting seasons (such as *tBATS Innovation State Space Modeling Framework* [28], *ARIMA Stochastic Process Modeling Framework* [25]) are computationally infeasible for streaming workloads.

Taft et al. [40] applied time-series analysis in the context of OLTP databases. The authors argue that reactive approaches don't fit to database world. By the time, auto-scaler component decides to scale-out, it is already too late for a database system. This premise comes from the fact that taking scaling actions in a database doesn't take place in timely manner. The database system has to replicate some of the records which is an additional burden on a heavily loaded system. Thus, database system must take proactive approach and take scaling decisions ahead of time. While this is convincing argument, the auto-scaler module depends on a couple of parameters that are hard to calculate in heterogeneous public cloud environments. First, target throughput of a single server. Second, shortest time to move all database records with single sender-receiver thread. While this might be feasible in some scenarios, on today's cloud environments with virtual machines hosted on heterogeneous physical nodes, getting a near-precise number is unconvincing. It worth noting that author assumed an approximately uniform workload distribution for all database nodes – each database shard serves a fairly equal portion of total workload which is a questionable assumption.

8.4 Queuing Theory Techniques

Lohrmann, Janacik, and Kao [29] proposed a solution based on queuing theory. The solution is designed for *Nephele* [30] streaming engine which has a master-worker style architecture. Similar to Spark Streaming, a job is modeled as a DAG. It utilizes *adaptive output batching* [42] – which is essentially a buffer with variable size – to buffer outgoing messages emitted from one stage to the other. Each task – an executer that runs user defined function (UDF) – is modeled as a G/G/1 queue. That is, the probability distributions of message inter-arrival and service time are unknown. In order to approximate these distributions, a formula proposed by Kingman [27] is used. From a bird's eye view, this solution seems promising. However, authors made two inconceivable assumptions that led us to abandon the proposal. First, worker nodes shall be homogeneous in terms of processing power and network bandwidth. Second, there should be an effective partitioning strategy in place in order to load balance outgoing messages between stages. In reality both assumptions rarely occur. Large scale stream processing clusters are built incrementally. Depending on workload, data skew does exist and imperfect hash functions are widely used by software developers.

Zhang, Cherkasova, and Smirni [44] proposed a solution for multi-tiered enterprise applications based on regression techniques. Regression based models can absorb some level of uncertainty and noise by compacting samples. Each tier is modeled as G/G/1 queue and scaled differently compared to other tiers. The system has fixed number of users – a principle known as *closed-loop queuing network*. In order to calculate system workload – incoming message rate – and service time which is required by queuing models, the authors proposed to use Mean Value Analysis [33]. In order to simplify the queuing network, the system is modeled as a *transaction-based* system with independent requests coming from clients. However, It is widely believed that multi-tiered enterprise applications are *session-based* systems [9]. Each request from the same client depends on her previous request during a specific session.

8.5 Reinforcement Learning Techniques

Herbst et al. [21] surveys on state of the art techniques to predict future workload. It includes workload forecasting based on *Bayesian Networks* (BN) and *Neural Networks*. There are several issues with each of them that makes them unsuitable for streaming workloads. As an example, there is no universally applicable method to construct a BN. Furthermore, it requires collecting data and training the model offline. Neural networks suffer from the same issues. That is, it requires collecting samples and periodically training the model. For complex models, training phase is typically computationally infeasible which is conflicting with requirements of thesis.

Tesauro et al. [41] proposes a hybrid approach to overcome poor performance of online training. The system consists of two components: an online component based on queuing system combined with Reinforcement Learning component that is trained offline. The offline component is based on *neural networks*. The authors model the data center as multiple applications managed under a single resource manager. Modeling streaming workloads as a queuing system has two problems. First, modeling is a complicated process and determining probability distributions requires domain knowledge. Second, it requires access to each node (so it can be modeled as a queue) which is currently not possible without modifying `spark-core` package. Since, it was one the requirements to provide a solution without making any modification to `spark-core`, this work has been abandoned.

Rao et al. [35] proposed to use Reinforcement Learning to manage resources consumed by virtual machines. It employs standard model-free learning, which is known as *Temporal Difference* [39] or *Sarsa* algorithm. The state space consists of metrics collected from virtual machines (CPU, RAM, Network IO, . . .). There is no global controller and each node decides based on its own Q-Table. As mentioned in literature, standard temporal difference has a slow convergence speed. In order to speedup bootstrap phase, Q-Table is initialized by values that were obtained during separate supervised training. Since this approach also relies on offline training, it wasn't adopted by this thesis.

Enda, Enda, and Jim [13] proposed a parallel architecture to Reinforcement Learning. Standard model-free learning (Temporal Difference) is used. No global controller is involved and each node decides locally. In order to speed up learning, all nodes maintain two Q-Tables (local and global tables). Local table is learned and updated by each node. Whenever, an agent learns a new value for a specific state, it broadcasts it to other agents. The global table contains values received from other agents. Additionally, agent prioritize local and global tables by assigning weights to each table. Weights are factors that are defined by application developers. The final decision is the outcome of combining local and global tables. Although each node learns some part of the state space (which may overlap with other nodes), it is not applicable in the context of Spark Streaming. The assumption in this architecture is that, each node is operating autonomously without intervention from other nodes (such as web servers). In contrast, Spark is a centrally managed system. That is, all nodes running Spark jobs are supervised by a single master node (probably with couple of backup masters).

Heinze et al. [19] implemented Reinforcement Learning in the context of FUGU [17] and compared it to threshold-based approaches. Each node, maintains its own Q-Table and imposes local policy without coordinating other nodes. This architecture can not be applied in the context of spark streaming, since Spark abstracts away individual nodes from the perspective of application developer. In order to decrease state space, the author applied two techniques. First, only system utilization is considered. Second, system utilization is discretized using coarse grained steps. To remedy slow convergence, the controller enforces a *monotonicity constraint* [24]. That is, if the controller decides to take scale-out action for a specific utilization, it may not decide scale-in for even worse system utilization. This feature has been adopted by this thesis.

Cardellini et al. [6] proposed a two level hierarchical architecture for resource management in Apache Storm [38]. There is a local controller on each node which is cooperating with the global controller. The local controller monitors each operator using different policies (threshold-based or Reinforcement Learning using temporal difference). In case, local controller decides to scale in or out an specific operator, it contacts the global controller and informs it about its decision. Then it waits to receive confirmation from the global controller. The global controller operates using a token-bucket-based policy [7] and has global view of cluster. It ranks requests coming from local controllers and either confirms or rejects their decisions. Although, this architecture seems to be a promising approach, however it has been implemented by modifying Storm's internal components. As mentioned above, this is in conflict with thesis's requirements.

In order to mitigate the problem of large state space in Reinforcement Learning, Lolos et al. [31] proposed to start the agent from small number of coarse grained states. As more metrics are collected (and stored as historical records), agent will discover *outlier* parameters (those parameters that are affecting agent more, CPU rather than IO as an example). Then, it partitions the affected state into two states and *re-trains* newly added states using historical records. Both Temporal Difference and Value Iteration methods can be used as learning algorithm. Gradually, agent only focuses on some specific parts of the state space, since all parameters are not equally important. This approach, effectively reduces

the size of state space. However, the trade-off is the storage cost in which historical metrics need to be stored. It worth noting that from the context of paper, storage cost (whether it is in-memory or on-disk and the duration of storing historical metrics) is unclear. Thus, this approach has been abandoned due to uncertainty.

Dutreilh et al. [12] proposed a model-based Reinforcement Learning approach for resource management of cloud applications. All virtual machines are supervised by a single global controller. Slow convergence is the bottleneck of model-free learning, in contrast to model-based learning. However, environment dynamics are not available at the time of modeling. Authors proposed to estimate these parameters as more metrics are collected and then switch to *Value Iteration* [39] algorithm instead of *Temporal Difference*. In short, statistical metrics are stored and updated for each visit of (old state, action, reward, new state) quadruple. As more samples are collected, statistical metrics become more accurate and can be directly used in *Bellman* equation. Until enough measurements get collected, a separate initial reward function is used which is essentially the original reward function but with penalty costs removed. Furthermore, In order to reduce the state space – tuple of [request/sec, number of VMs, average response time] – there exists a predefined upper and lower bound for state variables and average response time is measured at granularity of seconds. This approach has been partially adopted by this thesis.

Dutreilh et al. [11] proposed a model-free Reinforcement Learning approach (*Temporal difference* algorithm) with modified *exploration* policy. The standard exploration policy for Q-Learning is $1-\epsilon$. Under this policy, the agent performs a random action with probably of ϵ and with probability of $1-\epsilon$, it adheres to an action proposed by optimal policy. Although the random action is necessary to explore unknown states, but it has sever consequences under streaming workloads. In some cases, it leads to unsafe states where SLOs are severely violated. Since streaming is heavily latency sensitive, this property is undesirable. Thus, author sought toward a heuristic-based policy proposed by Bodik et al. [5]. This policy is based on couple of key observations which has been adopted by this thesis:

- It must quickly explore different states.
- It should collect accurate data as fast as possible, to speedup training.
- During exploration phase, the policy should be careful not to violate SLOs.

8.6 Summary

In this chapter prior work on auto-scaling scaling has been discussed and evaluated. First, threshold-based approaches are investigated. Simple threshold-based approaches are intuitive and simple to understand by application developers and are widely supported by cloud providers. However, adjusting thresholds is a tricky and error-prone process. Then, time-series analysis techniques are explored. As confirmed by other authors, advanced seasonal forecasting is a computationally intensive process, which makes it less suitable for streaming workloads. Queuing theory approaches are suitable for stationary networks with a known probability distribution for workload and service time. Reinforcement Learning techniques has the benefit that it requires zero knowledge about the environment which helps to gradually adapt to changes in environment.

9 Conclusion

Bibliography

[1] Amazon. *Amazon AWS Cloud*. Accessed July 16, 2018. 2018. URL: https://aws.amazon.com.

[2] Apache. *Apache Flink*. Accessed July 17s, 2018. 2018. URL: https://flink.apache.com.

[3] Apache. *Apache Spark*. Accessed July 17s, 2018. 2018. URL: https://spark.apache.com.

[4] A. Bifet and R. Gavaldà. "Learning from Time-Changing Data with Adaptive Windowing". In: *Proceedings of the 7th SIAM International Conference on Data Mining*. Vol. 7. Apr. 2007.

[5] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. "Automatic Exploration of Datacenter Performance Regimes". In: *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*. ACDC '09. Barcelona, Spain: ACM, 2009, pp. 1–6.

[6] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo. "Decentralized self-adaptation for elastic Data Stream Processing". In: *Future Generation Computer Systems* 87 (2018), pp. 171 –185.

[7] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. "Towards Hierarchical Autonomous Control for Elastic Data Stream Processing in the Fog". In: *Euro-Par 2017: Parallel Processing Workshops*. Ed. by D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer. Cham: Springer International Publishing, 2018, pp. 106–117.

[8] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 725–736.

[9] L. Cherkasova and P. Phaal. "Session-based admission control: a mechanism for peak load management of commercial Web sites". In: *IEEE Transactions on Computers* 51.6 (2002), pp. 669–685.

[10] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144.

[11] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. "From Data Center Resource Allocation to Control Theory and Back". In: *2010 IEEE 3rd International Conference on Cloud Computing*. 2010, pp. 410–417.

[12] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck. "Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a fully automated workflow". In: *7th International Conference on Autonomic and Autonomous Systems (ICAS'2011)*. Venice, Italy, May 2011, pp. 67–74. URL: https://hal-univ-paris8.archives-ouvertes.fr/hal-01122123.

[13] B. Enda, H. Enda, and D. Jim. "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud". In: *Concurrency and Computation: Practice and Experience* 25.12 (2012), pp. 1656–1674.

[14] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. "Exploring Alternative Approaches to Implement an Elasticity Policy". In: *2011 IEEE 4th International Conference on Cloud Computing*. 2011, pp. 716–723.

[15] Google. *Google Cloud*. Accessed July 16, 2018. 2018. URL: https://cloud.google.com.

[16] Google. *Kubernetes*. Accessed July 17s, 2018. 2018. URL: https://kubernetes.io.

[17] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. "Multi-resource Packing for Cluster Schedulers". In: *SIGCOMM Comput. Commun. Rev.* 44.4 (2014), pp. 455–466.

[18] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi. “Integrated and autonomic cloud resource scaling”. In: *2012 IEEE Network Operations and Management Symposium* (2012), pp. 1327–1334.

[19] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. “Auto-scaling techniques for elastic data stream processing”. In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. 2014, pp. 296–302.

[20] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. “Online Parameter Optimization for Elastic Data Stream Processing”. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC ’15. Kohala Coast, Hawaii: ACM, 2015, pp. 276–287.

[21] N. Herbst, A. Amin, A. Andrzejak, L. Grunske, S. Kounev, O. J. Mengshoel, and P. Sundararajan. “Online Workload Forecasting”. In: *Self-Aware Computing Systems*. Ed. by S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu. Cham: Springer International Publishing, 2017, pp. 529–553.

[22] N. R. Herbst, S. Kounev, and R. Reussner. “Elasticity in Cloud Computing: What It Is, and What It Is Not”. In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 23–27.

[23] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. “Self-adaptive Workload Classification and Forecasting for Proactive Resource Provisioning”. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE ’13. Prague, Czech Republic: ACM, 2013, pp. 187–198.

[24] H. Herodotou and S. Babu. “Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs”. In: *Proceedings of the VLDB Endowment*. Vol. 4. Jan. 2011, pp. 1111–1122.

[25] R. Hyndman and Y. Khandakar. “Automatic Time Series Forecasting: The forecast Package for R”. In: *Journal of Statistical Software, Articles* 27.3 (2008), pp. 1–22.

[26] M. Kielbowicz. “Online parameter optimization for Spark Streaming”. SAP, 2017.

[27] J. F. C. Kingman. “The Single Server Queue in Heavy Traffic”. In: *Proceedings of the Cambridge Philosophical Society* 57 (1961), p. 902.

[28] A. M. D. Livera, R. J. Hyndman, and R. D. Snyder. “Forecasting Time Series With Complex Seasonal Patterns Using Exponential Smoothing”. In: *Journal of the American Statistical Association* 106.496 (2011), pp. 1513–1527.

[29] B. Lohrmann, P. Janacik, and O. Kao. “Elastic Stream Processing with Latency Guarantees”. In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. 2015, pp. 399–410.

[30] B. Lohrmann, D. Warneke, and O. Kao. “Nephele streaming: stream processing under QoS constraints at scale”. In: *Cluster Computing* 17.1 (2014), pp. 61–78.

[31] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. “Elastic Resource Management with Adaptive State Space Partitioning of Markov Decision Processes”. In: (2017).

[32] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments”. In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592.

[33] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[34] Microsoft. *Microsoft Azure Cloud*. Accessed July 16, 2018. 2018. URL: `https://azure.microsoft.com`.

[35] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. “VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration”. In: *Proceedings of the 6th International Conference on Autonomic Computing*. ICAC ’09. Barcelona, Spain: ACM, 2009, pp. 137–146.

[36] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. “Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. New York, NY, USA: ACM, 2012, 7:1–7:13.

[37] RightScale. *Set up Autoscaling using Voting Tags*. Accessed June 20, 2018. 2018. URL: http://support.rightscale.com/12-Guides/Dashboard_Users_Guide/Manage/Arrays/Actions/Set_up_Autoscaling_using_Voting_Tags/.

[38] A. Storm. *Apache Storm*. Accessed June 21, 2018. 2018. URL: http://storm.apache.org/.

[39] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. The MIT Press, 1998. ISBN: 0262193981.

[40] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. "P-Store: An Elastic Database System with Predictive Provisioning". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 205–219. ISBN: 978-1-4503-4703-7.

[41] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation". In: *2006 IEEE International Conference on Autonomic Computing*. 2006, pp. 65–73.

[42] D. Warneke and O. Kao. "Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud". In: *IEEE Transactions on Parallel and Distributed Systems* 22.6 (2011), pp. 985–997.

[43] T. Ye and S. Kalyanaraman. "A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration". In: *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '03. San Diego, CA, USA: ACM, 2003, pp. 196–205.

[44] Q. Zhang, L. Cherkasova, and E. Smirni. "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications". In: *Fourth International Conference on Autonomic Computing (ICAC'07)*. 2007, pp. 27–27.