An Elastic Layers Pattern Approach with Dynamically Added Layers

Christian Zirkelbach and Marc Adolf
Software Engineering Group
Kiel University, Germany
{czi,mad}@informatik.uni-kiel.de

Abstract

Cloud environments often provide dynamic resource allocation techniques. These can be used to scale single components or even whole software systems according to their current workload. Fluctuating workloads can occur in different layers of the software architecture and need appropriate handling to meet performance requirements. Scaling workload-intensive components in combination with load-balancing can be used to deal with these issues. Therefore, we present a parallel layers approach, which extends an existing pattern regarding improved elasticity. Based on dynamically adding layers on top of bottleneck layers, we increase the flexibility and performance of related architectures. Furthermore, we describe a first design approach, implementation and an evaluation of the feasibility.

1 Introduction

In many applications the received workload varies dependent on different external influences. Often, this creates peaks or periods of time with fluctuating workloads. In order to fulfill performance requirements like *Service Level Agreements* (SLA), an application has to handle these occurrences. Therefore, software has to be scalable to utilize elastic resource allocation to avoid over-provisioning. Additionally, scaling only bottleneck related components can reduce needed resources and costs. Related approaches are architectures like Mapreduce [1] or Microservices [5], or self-adapting techniques like [4]. In this paper, we present a parallel layers pattern approach, which extends an existing pattern [2]. We reduce the amount of scaled components, and increase the flexibility and performance of related architectures by dynamically adding layers to reduce the workload in bottleneck layers. Furthermore, we describe a first design approach and implementation, which we successfully integrated in ExplorViz [6], our web-based tool for live trace visualization of large software landscapes.

In the following, we describe our approach. Then, we present a first implementation in order to show the feasibility of our approach. Finally, we draw the conclusions and illustrate future work.

2 Parallel Layers Pattern

In the parallel layers pattern [2], single components of a layer can be duplicated to enable parallelism and avoid bottlenecks. These duplicated components communicate with the same component in a higher layer than the original. Likewise, they can not use existing underlying components. Hence, the overall architecture forms a tree. If a single node is duplicated, it is necessary to replicate the whole subtree. Therefore, the related layers can be scaled in terms of width. The structure and layering is designed beforehand and the number of layers is fixed [2]. This behavior can lead to a higher resource allocation than necessary for a system to react to certain workloads. If one of the intermediate layers or even the root needs more (computational) effort than lower layers for processing results, the whole (sub-)tree needs to be duplicated. These effects can increase, if the parallel layers pattern is used to duplicate underlying layers.

For example, we register new sensors in a layered monitoring system. Here, the collection of data from sensors requires less effort than the processing. Even duplicating these sensors is not a solution.

3 Elastic Parallel Layers Pattern

For this reason, we propose an approach, which is based on the parallel layers pattern [2] and enables a system to scale dynamically. The approach increases the performance of a single layer by horizontally duplicating the related components and vertically by adding accumulation layers. In the following, we present the principle of our approach based on an example.

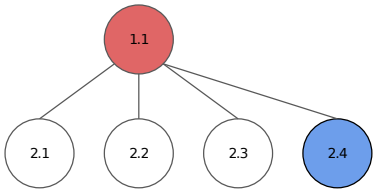

Figure 1: Layered Architecture - `Node 1.1` has high workload, `Node 2.4` was created

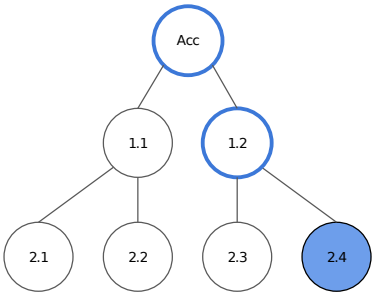

Figure 2: Extended Layered Architecture: `Node 1.2` and an `accumulator` were created

In Figure 1, we start with a layered architecture with two layers. `Node 2.4` is added to the system and generates additional load for the processing `Node 1.1`. Since the system does not meet the *SLA*s anymore, the load has to be rebalanced within the system. Thus, `Node 1.1` is duplicated. An accumulator component (`Acc`) is needed to combine the results of the parallel execution. This can be handled through a new component with the single purpose of accumulating received data. Alternatively, if the input and output of the accumulator require the same type of data, a duplicate of the node can be used. The resulting architecture is shown in Figure 2. The nodes of the second layer, which receive the data, are redistributed between the existing `Node 1.1` and the new `Node 1.2`. At the top, a new layer with a single Accumulator node is created, which collects the results of these two. If the workload further increases, more layers can be added. The architecture is still conform to the parallel layers pattern and all scaling operations behave as before. The changes can also be reverted to reestablish the original architecture.

Two prerequisites have to be considered to follow our approach. First, for every layer that should be able to scale using our approach, an accumulation component has to be defined. Second, a distribution strategy is needed to define how the workload should be rebalanced.

4 Implementation

As a first implementation, we developed an elastic, scalable monitoring approach to avoid the over-utilization of a single analysis node in the context of distributed application-level monitoring [3]. We successfully integrated the approach in our tool ExplorViz, which consists of three major components, *monitoring*, *analysis*, and *visualization*. The *monitoring* component gathers data of instrumented applications and passes the recorded data to the *analysis* component, where execution traces are reconstructed and aggregated. Finally, the *visualization* can be accessed through a web browser.

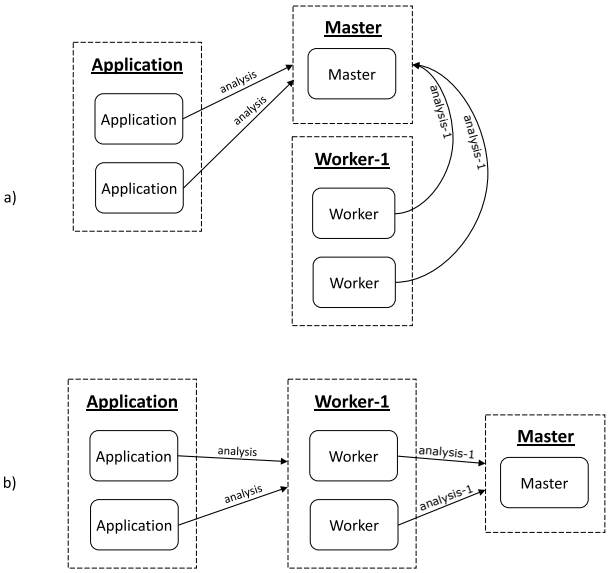

Figure 3: Upscaling process based on [3]

4.1 Design

Our approach focuses on the *analysis* component, as this element can be affected by over-utilization within our monitoring approach. If a deployed analysis node becomes over-utilized, we dynamically add a new worker level ahead of it. Furthermore, to circumvent this situation with newly created worker levels, we scale the associated worker applications within their worker level. In order to allow multiple worker levels, the *analysis* component offers two different modes. Deployed as a worker node, incoming monitoring data is preprocessed and passed to the next analysis node. If the analysis node is running as a master node, it adds up the preprocessed monitoring information. Hence, in this configuration the master node acts as the accumulator.

4.2 Scaling Process

The process of scaling analysis nodes during the monitoring of an scalable example software system is illustrated in Figure 3. A single scaling group (name is displayed on top), i.e., a group of applications, which are scaled independently, is illustrated as boxes with dashed lines. In the beginning, we have just two scaling groups – `Application` and `Master`. Arrows illustrate directed accesses and the related label is used to request an IP address for a specific target scaling group from a *LoadBalancer*, which is integrated in our implementation.

Initially, the scaled monitored software system (multiple instances of Application) directly communicates with the `Master`. Figure 3 shows the process of dynamically adding one worker level, i.e., a new scaling group. This scaling process is triggered, once the CPU utilization of the `Master` rises above a

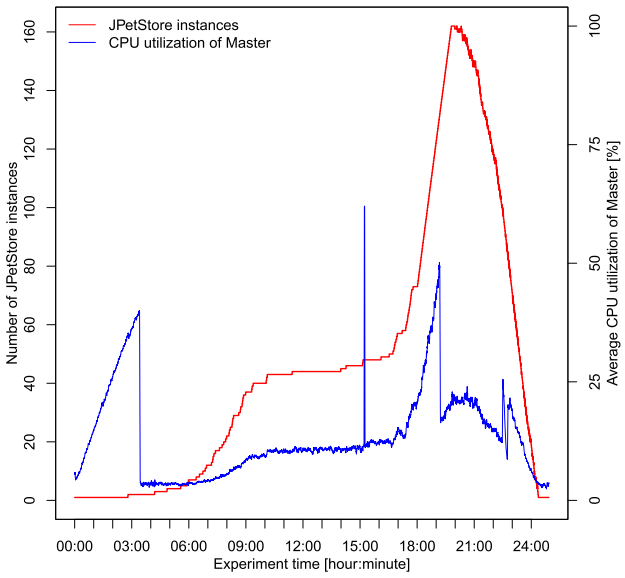

Figure 4: Instance count and average CPU utilization of the `Master` scaling group [3]

defined threshold. First, a new worker level is created (*worker-1*). Then, we start two new worker nodes with the same configuration within this level. The newly created worker nodes are connected to the `Master` in order to prepare the delivery of their preprocessed monitoring data. This state is shown in Figure 3a. Afterwards, the communication of the monitored applications is switched from the `Master` to the new worker level (*worker-1*). The resulting state is presented in Figure 3b. As the analysis should not be suspended during the scaling process, the order of changing the communication between levels is vital. Subsequently, if the workload decreases, worker levels are removed step-by-step. This down-scaling process works in reverse order of the upscaling process.

4.3 Evaluation

In order to evaluate our approach, we conduct an experiment and monitor the web application JPetStore[1] with ExplorViz. We employ a modeled workload, which characterizes a day-night-circle of a typical website, with rising workload until 6 p.m, a peak at around 9 p.m., and a decreasing load till midnight. The experiment utilizes our private OpenStack[2] cloud with a maximum of 216 virtual cores (VCPU). Our initial configuration contains only the `Master` with one VCPU.

Figure 4 shows the resulting number of instances and the average CPU utilization of the `Master` in our experiment. The count of JPetStore instances follows our workload curve and results in a maximum of 160 JPetstore instances at the peak with two intermediate worker levels. Once the workload rises above the defined threshold of CPU utilization (40%), another worker level is started. This situation can be specifically observed at 3 a.m. Thus, the CPU utilization of the `Master` drops to nearly 3%. Another worker level is added at 7 p.m. When the workload falls between 8 p.m. and midnight, the number of JPetStore instances decreases. Hence, the second worker level is removed at 10 p.m. and consequently the first worker level at midnight. Thus, our initial configuration is retrieved at the end of our experiment. The results confirm, that our implementation is able to handle an over-utilization of the `Master` with increasing workload. Further information of the experiment can be found in [3].

[1] `https://github.com/mybatis/jpetstore-6`
[2] `https://www.openstack.org`

5 Conclusion

In this paper, we present an extended, elastic parallel layers approach. Based on dynamically adding new layers, we are able to provide a high level of scalability, especially for cloud environments. Additionally, we developed a first design approach and implementation, which we successfully integrated in ExplorViz. Furthermore, we conducted an experiment to evaluate our implementation. The experiment results confirm, that our approach is able to handle fluctuating workloads. As future work, we plan (i) to further evaluate the pattern on scenarios with a higher number of worker levels and multiple layers with different tasks and (ii) to compare the presented approach with the parallel layers pattern [2].

References

[1] J. Dean and S. Ghemawat. “Mapreduce: simplified data processing on large clusters.” In: *Proceedings of OSDI*. 2004.

[2] J. L. Ortega-Arjona. “The Parallel Layers Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming.” In: *Proceedings of SugarLoafPLoP*. 2007.

[3] F. Fittkau and W. Hasselbring. “Elastic Application-Level Monitoring for Large Software Landscapes in the Cloud.” In: *Proceedings of ESOCC*. Springer, 2015.

[4] N. Huber et al. “Model-Based Autonomic and Performance-Aware System Adaptation in Heterogeneous Resource Environments: A Case Study.” In: *Cloud and Autonomic Computing (ICCAC)*. 2015.

[5] S. Newman. *Building Microservices*. O’Reilly Media, Inc., 2015.

[6] F. Fittkau, A. Krause, and W. Hasselbring. “Software landscape and application visualization for system comprehension with ExplorViz.” In: *Information and Software Technology* (2016). http://dx.doi.org/10.1016/j.infsof.2016.07.004.