Proposal: Dynamic Load Scaling via Distributed Trace & Metric Analysis

UIUC / CS 598 - Cloud Computing Capstone / Fall 2024

John Durkin Kevin Mooney Sushil Khadka Samson Koshy

Project Idea

For our project, we propose a software system which analyzes application generated distributed tracing and metric data to perform dynamic scaling of application work loads in cloud hosted and/or on-premises containerized environments. We envision a monitor that can make use of open technologies to probe said distributed tracing and metric data, that would also wire up with cloud or container orchestration APIs. Based on provided rules and thresholds, the monitor would call these APIs in order to scale computational units by operator configured intervals. Most current dynamic scaling systems are typically either closed systems or tightly coupled to specific providers or technology stacks. Often, these systems only make use of a limited set of metrics, typically focusing only on hardware resource utilization.

We are proposing a more customizable solution than the current state of the art. One that works well with multiple open source technologies as well as proprietary cloud APIs, thus making it more portable across different cloud providers or on-premises container orchestration platforms. Most development teams today are instrumenting their applications in some form for distributed tracing, metrics collection or both. These help provide data points which can be used for real time monitoring, trend research and perfomance analysis. However, these same data points could in turn be analyzed, and when combined with a rich set of thresholds and rules provided by development or operational teams, be the driver behind dynamic resource scaling.

Justification

Whether running workloads on remote cloud or on-premises clusters or compute grids, one of the most significant factors that can affect operational expense and capital expenditures is proper resource utilization. On the one hand, if you are under utilizing, then wasted cost is obvious in the form of used cloud resources or hardware. On the other hand, if you are overutilizing then this can negatively impact resopnse times or worse yet introduce failures within your systems. At the very least this can frustrate your client base, while in the worst case cause detrimental production outages, possibly causing permanent damage to your brand.

To combat this, software delivery teams are starting to lean heavily on dynamic workload scaling. This allows them to utilize resources efficiently and as needed. For fullscale cloud workloads, this promotes requesting the optimal amount of resources at any given time, maximizing system throughput when service requests are at their peak and relenquishing those resources during periods of low traffic. For on-premises, while the capital expense of buying hardware cannot be unwound, there is still benefit as you can properly scale varying workloads throughout different periods to better share resources across internal development teams and business units.

While technologies exist to varying degress to support dynamic load scaling, current implementations seem to have their limits. Many are simply focused directly on hardware resources such as CPU and memory. If the CPU utilization breaches a certain barrier, then new instances of some computational unit will be spun up. There are more advanced scalers out there. For instance AWS can auto-scale based on CloudWatch metrics. How-

ever, These can often be tedious to instrument and proprietary in nature.

To elaborate, imagine that some distributed application makes use of a serverless function. This function may have low memory and cpu utilization, but is taking a very long time in some blocking IO. By inspecting traces we can see the latency of the IO operation as the bottleneck, and scale up the amount of serverless workers. At the same time, we can also see that we aren't breaching high memory or CPU utilization or our serverless instances, thus when scaling up the monitor can request less resources for each instance, maximizing our operational expense while at the same time providing optimal performanceto our systems.

While likely outside of the scope of our research for this semester, this could also lead to other opportunities involving deeper analysis of the data using various ML or reporting mechanisms, helping you plan your allocations in a more thoughtful manner rather than only looking at things like CPU, memory, IO and network utilizaiton. The effects of the scaling and the analysis of the trace date and metrics that brought it about could also be combined with business based trend data. This overlaying of this data can allow technology teams to associate operational expense to real world factors, such as client usage trends or other business impacting events.

Preliminary Plan

After discussion on expertise amongst the team, we have converged on Python as our primary programming language to develop our proposed monitoring system. If there are performance or compatibility concerns, we will approach other languages as necessary. For data collection, we plan to focus initially on distributed trace data collected from OpenTelemetry compliant trace stores with possible candidates being Jaeger or Tempo. Based on operator provided rules and thresholds, the monitor will scale computational units based on much more than hardware resource utilization, which is often the norm in current systems.
At first this configuration will come in the form of a file utilizing some an arbitrary mark up or declarative language. While we intend the systems which can be scaled to be pluggable, we will begin our experimentation and scaling workloads run Minikube, a compact Kubernetes installation that is simple to run on localized commodity environments.

Our initial experimentation should focus on two main areas. Both of these should center around the comparison to existing scaling systems. The first area of focus should demonstrate if our system actually works improves latency in a distributed system over other scaling systems. The second area of focus should center around measuring amounts of resources that are requested when scaling. This would pertain mostly to determining if one system is over subsribing to resources to achieve the same latency improvements.

Literature Survey

A few important notes from some of the papers the team has researched are provided below. Other papers that we have drawn influence and insight from are included in the references section as well. While we have found many systems that make use of observability data for system analysis we have not yet come accross one that makes direct use of distributed traces for system scaling.

Research on DeepFlow [1] tells us that a variety of observability tools are currently in use for tasks such as performance monitoring, system diagnostics, and troubleshooting. Ongoing research aims to extend the capabilities of distributed tracing, including addressing network blind spots that are often overlooked by existing frameworks

From [2] we learned that identifying the appropriate metrics is often challenging, as most research predominantly focuses on CPU, memory, and disk usage. For services that cannot be effectively profiled using CPU or memory, other metrics or custom measures (such as the number of active sessions) should be considered. Additionally, there is a notable lack of generic, general-purpose auto scalers to address these needs.

STEAM [3] presents an effective telemetry sampling algorithm implemented on top of OpenTelemetry. Something we must consider is the sheer volume of trace data that a large system can produce. For example, Google generates approximately 1 PB of raw trace data daily. Obviously there comes a point that it's no longer feasible to collect / analyze all of the data. To handle such volumes, organizations often use head-based (uniform) trace sampling, randomly selecting a subset of traces to store. However, this method can com-

promise observability because it may discard rare but important traces due to the long-tail distribution of trace data.

Research in [4] shows that to gain insights into serverless applications, distributed tracing can be employed. However, for practical purposes, it is essential that distributed tracing does not significantly degrade application performance, thereby negating its benefits. When comparing serverless applications with and without tracing, some performance impact was observed. The extent of this impact—measured in terms of runtime, memory usage, and initialization duration varied across the tools analyzed (Zipkin, OTel, and SkyWalking). With appropriate tool selection, the advantages of distributed tracing can be maximized while minimizing performance drawbacks.

The architecture described in [5] extends a service mesh to support distributed tracing without requiring internal application changes. It essentially stitches together a graph of parent and child service calls based on URL parsing and the flow of requests between services. This architecture is useful in scenarios where modifications to the application code are impractical or undesirable. Although the service mesh architecture offers tracing without application modifications, there is overhead in parsing metadata from the traffic between services. This overhead can increase with the complexity of the service-to-service interactions, potentially affecting the performance of the mesh itself. In scenarios with a large number of microservices or highly dynamic traffic patterns, the mesh's ability to parse metadata efficiently could degrade, leading to performance bottlenecks.

While TraceWeaver [6] is an interesting approach, its limitations highlight the challenges of handling complex microservice architectures where workflows are not static. It's ideal for cases where the call graph is predictable, but less so for dynamic workflows, which are becoming more common. The system works best when services follow standardized or well-defined communication patterns, as this allows TraceWeaver to predict and trace service interactions. However, in cases where services expose non-standard endpoints or engage in more complex, non-deterministic communication, TraceWeaver's effectiveness diminishes. This creates a limitation in environments where microservices do not adhere to standard API patterns or have unpredictable routing logic.

Astraea [7] provides a data-driven way to identify performance issues, particularly for diagnosing new deployments or changes in service versions. The application of machine learning techniques allows for more accurate predictions and correlations in performance degradation, making this system powerful for long-term monitoring. Since the system relies on sampling, the focus is primarily on high-volume, frequent traces. However, rare but significant events, such as intermittent failures or performance issues that occur under specific conditions, may not be adequately captured or analyzed. This could result in missed opportunities to diagnose critical issues that do not happen regularly but are highly impactful when they do.

LatenSeer [8] provides a framework for using existing distributed trace data in order to build models that can help predict latency when introducing change to a software environment. The model uses trees whose roots are request entry points and children are groupings of similar call graphs. Grouping simliar traces eliminates noise and complexity that can arise from analyzing very diverse and high cardinality call graphs. Users can inject hypothetical latencies to specific tracepoints (which typically represent some microservice) in the graph to simulate real world events such as data center migrations, new service dependencies or new components to a workflow.

References

[1] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Shen, Z. Zhang, Y. Wu, X. Yin, J. Wang, M. Xu, Y. Li, J. Yin, J. Song, Z. Li, R. Nie, Network-centric distributed tracing with DeepFlow: Troubleshooting your microservices in zero code, in: Proceedings of the ACM SIGCOMM 2023 Conference, Association for Computing Machinery, New York, NY, USA, 2023: pp. 420–437. https://doi.org/10.1145/3603269.3604823.

[2] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, S. Kounev, Why is it not solved yet? Challenges for production-ready autoscaling, in: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, Association for Computing Machinery, New York, NY, USA, 2022: pp. 105–115. https://doi.org/10.1145/3489525.3511680.

[3] S. He, B. Feng, L. Li, X. Zhang, Y. Kang, Q. Lin, S. Rajmohan, D. Zhang, STEAM: Observability-preserving trace sampling, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, 2023: pp. 1750–1761. https://doi.org/10.1145/3611643.3613881.

[4] C. Eder, S. Winzinger, R. Lichtenthaler, A comparison of distributed tracing tools in serverless applications, in: 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE), IEEE Computer Society, Los Alamitos, CA, USA, 2023: pp. 98–105. https://doi.org/10.1109/SOSE58276.2023.00018.

[5] D. Cha, Y. Kim, Service mesh based distributed tracing system, in: 2021 International Conference on Information and Communication Technology Convergence (ICTC), 2021: pp. 1464–1466. https://doi.org/10.1109/ICTC52510.2021.9620968.

[6] S. Ashok, V. Harsh, B. Godfrey, R. Mittal, S. Parthasarathy, L. Shwartz, TraceWeaver: Distributed request tracing for microservices without application modification, in: Proceedings of the ACM SIGCOMM 2024 Conference, Association for Computing Machinery, New York, NY, USA, 2024: pp. 828–842. https://doi.org/10.1145/3651890.3672254.

[7] M. Toslali, S. Qasim, S. Parthasarathy, F.A. Oliveira, H. Huang, G. Stringhini, Z. Liu, A.K. Coskun, An online probabilistic distributed tracing system, (2024). https://arxiv.org/abs/2405.15645.

[8] Y. Zhang, R. Isaacs, Y. Yue, J. Yang, L. Zhang, Y. Vigfusson, LatenSeer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing, in: Proceedings of the 2023 ACM Symposium on Cloud Computing, Association for Computing Machinery, New York, NY, USA, 2023: pp. 502–519. https://doi.org/10.1145/3620678.3624787.

[9] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, C. Xu, The power of prediction: Microservice auto scaling via workload learning, in: Proceedings of the 13th Symposium on Cloud Computing, Association for Computing Machinery, New York, NY, USA, 2022: pp. 355–369. https://doi.org/10.1145/3542929.3563477.

[10] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, C. Xu, Characterizing microservice dependency and performance: Alibaba trace analysis, in: Proceedings of the ACM Symposium on Cloud Computing, Association for Computing Machinery, New York, NY, USA, 2021: pp. 412–426. https://doi.org/10.1145/3472883.3487003.

[11] H. Huang, X. Zhang, P. Chen, Z. He, Z. Chen, G. Yu, H. Chen, C. Sun, TraStrainer: Adaptive sampling for distributed traces with system runtime state, Proc. ACM Softw. Eng. 1 (2024). https://doi.org/10.1145/3643748.