Tucano: A Service Scheduler and Load Balancer in a Distributed System

Luiz F. Gonçalves, Eduardo L. Paschoalini, Gustavo D. Aguiar, Yuri Rousseff, Matheus A. Souza, Pedro H. Ramos, Ricardo C. Sperandio, Felipe D. Cunha

Depto. de Ciência da Computação – Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte – MG – Brasil

{luiz.frazao, eduardo.paschoalini, gdaguiar, yrousseff}@sga.pucminas.br,
phramoscosta@gmail.com, {ricardosperandio, matheusalcantara, felipe}@pucminas.br

Abstract. *As applications grow in complexity, their workloads grow heavier and more resource demanding. To address this challenge, a system of network-connected servers can be employed to distribute the processes from such applications, reducing performance loss caused by unequal resource sharing. This work introduces Tucano: a scheduler that leverages operating systems' techniques to manage workloads in a distributed, container-oriented context. Additionally, Tucano implements a load balancer, that ensures the correct mapping of requests to their corresponding host servers. Through physical distribution and containerization, Tucano aims to explore an efficient and scalable strategy for managing multiple workloads.*

1. Introduction

Early on, organizations ran applications on physical servers without dedicated resource boundaries, causing resource allocation issues [Anand et al. 2021]. When a server is shared by multiple applications, one of them may consume the majority of resources, leading to a performance loss for others [Mirhosseini et al. 2019]. A simple solution is to run each application on a server, but this led to resource underutilization and high maintenance costs for many physical servers [Speitkamp and Bichler 2010]. These drawbacks highlight the need to subdivide applications into smaller processes and schedule them as manageable units, optimizing resource distribution.

A possible solution is virtualization, which enables multiple applications to run on a single physical server by using Virtual Machines (VMs) that emulate entire computer systems. Using VMs improves cost-efficiency by consolidating multiple systems onto a host server instead of using separate hardware for each. It also simplifies application management as each VM operates in its isolated environment. However, virtualization earns significant performance overhead by requiring the replication of the entire Operating Systems (OSs) and simulate necessary hardware, leading to high memory and CPU usage [Bermejo and Juiz 2020, Shirinbab et al. 2017].

Another approach to address this challenge is containerization. Unlike VMs, containers virtualize only the OS rather than the entire computer system. Each container typically carries a single application alongside all necessary binaries, libraries, and

Eduardo, Gustavo, Luiz, and Yuri are undergraduate students in the Computer Science program, advised by the other authors.

configuration files [Watada et al. 2019]. Containers share the host OS kernel and often its binaries and libraries in a read-only manner, eliminating the need to replicate OS code multiple times. This streamlined abstraction results in lower costs and greater efficiency due to reduced resource requirements. However, despite their flexibility and lightweight nature, environments utilizing containers must ensure continuous uptime and scalability of applications [Bernstein 2014, Morabito et al. 2017].

The third approach, and the one targeted in this paper, introduces Tucano, a hybrid model that integrates multiple physical machines with a container-oriented design for application deployment. Services are scheduled across a multi-computer system using an interconnection network for communication. A controller node orchestrates the scheduling routine, while worker nodes execute services as processes. Tucano goes beyond container management by overseeing the nodes that run these containers. It ensures each node operates within capacity, dynamically allocating services to capable nodes, thus reducing resource overhead and enhancing scalability. By managing both containers and their execution environment, Tucano streamlines operations and balances resource availability with application demand.

The rest of this paper is organized as follows. Different techniques form scheduling systems are discussed in Section 2. Section 3 introduces the system architecture and Section 4 gives implementation details of its components. The gathered results are shown in Section 5 before conclusion in Section 6.

2. Related Work

Previous studies have evaluated various aspects of scheduling techniques and container management. For instance, [Kovvur et al. 2013] presented a general description of the phases required by a scheduler. These phases consist of resource discovery, resource selection, task selection, and task monitoring. The presented approach extends these phases by separating resource management and task execution control into separate components, with more methods for dealing with workload balance on worker nodes.

Containers provide lightweight virtualization by encapsulating applications and their dependencies into isolated environments. In light of that, [Merkel et al. 2014] discussed the benefits of Docker, a containerization platform. The authors highlighted Docker's ability to package applications with everything they need. Our approach leverages Docker's lightweight nature, allowing for rapid deployment and scalability.

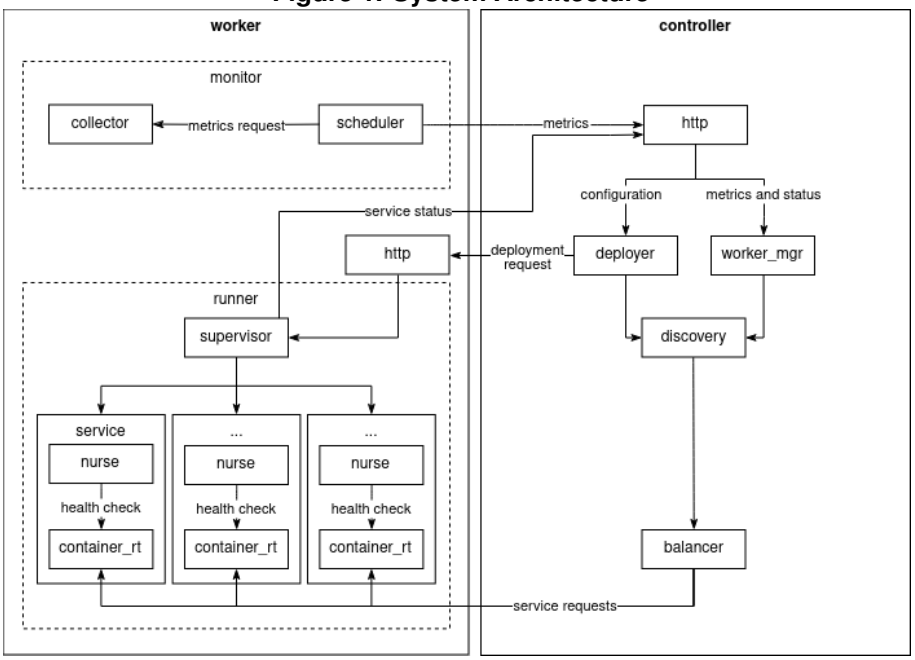
Finally, there is Kubernetes, a platform for automating deployment, scaling, and operations of container applications [Burns et al. 2016, Bernstein 2014]. The solution presented also aims for container cluster orchestration, providing high availability and scalability through a multicomputer ecosystem attached to a load balancer.

3. System Architecture

Some fundamental concepts are central to this work and need explicit definition. Tucano's architecture comprises two main components: a controller and a worker. Figure 1 presents how each sub-component of the controller and worker nodes has distinct responsibilities, also describing the communication establishment.

Within the controller, the **deployer** accepts a service's static configuration and starts its deployment. The **balancer** manages user service requests, distributing them

Figure 1. System Architecture



among system nodes. The **worker manager** receives and shares resource usage information from each worker. The **discovery** routine maintains mapping information for services and live worker nodes.

In a worker node, the **monitor** collects and sends metrics to the controller, while the **runner** deploys and executes container instances based on the controller's instructions, supporting multiple processes.

Inter-node communication uses the HTTP protocol, chosen for its simplicity. Previously, gRPC and custom TCP/UDP were considered, but a RESTful model reduced complexity. Intra-node communication draws insights from Erlang's actor model [Armstrong 2003] for message passing between actors within the same process. The following sections delve deeper into each component and their interactions within Tucano's architecture.

4. Implementation

4.1. Controller

An **HTTP server** establishes communication with the worker nodes and system administrator requests. As shown in Figure 1, the component receives metrics and processes' status from worker nodes and redirects those to other components.

The **Worker Manager** processes various messages, such as registering and deregistering workers, receiving metrics, and performing periodic liveness checks. Metrics management involves updating worker details and verifying their activity status. Periodic liveness checks ensure that workers are active by comparing the time elapsed since the last metrics update against the liveness timeout. If a worker fails to send metrics within the designated timeout, it is marked as dead and removed from the pool, thereby maintaining optimal system performance and preventing resource bottlenecks.

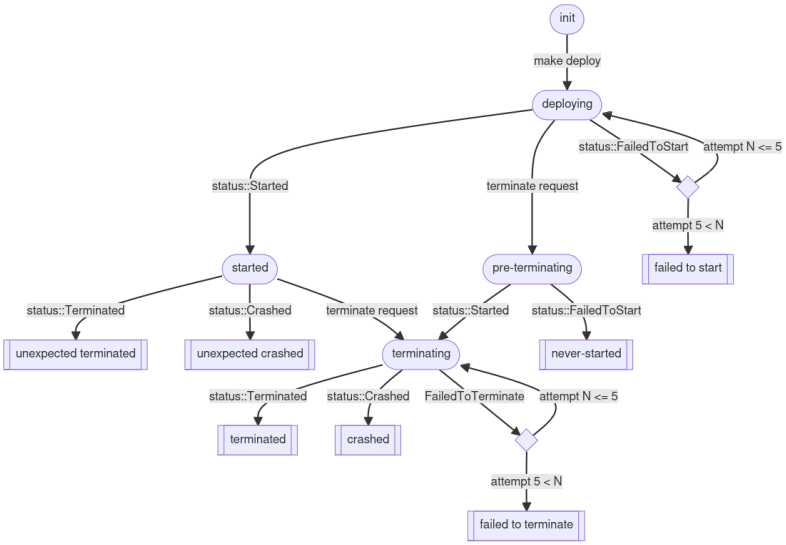
The **Discovery** component sustains the system's service discovery ability. It works as a central database, maintaining information about worker nodes, running services, deployment records, and overall system metrics. Additionally, it provides a message-passing-based API that other local components rely on.

To enhance the system's reliability and robustness in the face of failures, data is persisted on disk in an eventually-consistent manner using a SQLite database. This approach aims to minimize the latency of discovery operations by asynchronously organizing *most* data changes to disk in the background, rather than immediately upon receipt. This persisted data is crucial for recovering the system's state if the controller node needs to restart due to catastrophic failures. Even if not all data persisted before a crash, the remaining worker nodes can quickly provide the necessary information to rebuild the up-to-date system state.

Managing the lifecycle of services, the **Deployer** orchestrates the allocation of instances across available workers and ensures that services are deployed and terminated. Upon receiving a deployment request, it allocates suitable workers for the service instances based on the service configuration. This involves querying the discovery component for available workers and distributing the instances accordingly. The deployer ensures that the instances are deployed with the necessary resources and configurations, preparing the runtime environment and starting the containers.

Instance state transitions are managed through a state machine, depicted in Figure 2, which tracks states such as deploying, running, terminating, and crashing. The state machine handles retries for deployments and terminations as needed.

Figure 2. Deployer component state machine



Our **Load Balancer** distributes incoming HTTP requests using a round-robin algorithm. It extracts the service identifier from the `host` header to identify the target service. This ID is used to find available instances from a shared state. The round-robin mechanism evenly distributes requests by incrementing a counter and cycling through instances. After selecting an instance, the load balancer updates the request's URI (Uniform Resource Identifier) to the worker address and adds custom headers with the

instance ID and the client's original IP address. This ensures the request is routed correctly and distinguishes between multiple instances of the same service.

4.2. Worker

As workers communicate with the controller, fetch system information, and redirect requests, asynchronous programming becomes crucial. Instead of waiting for I/O tasks or network responses to be completed, it allows workers handling multiple tasks concurrently.

When a request is received, the **Proxy** first extracts the instance ID from the request header. It then searches the corresponding port for this instance ID in its internal map. If the instance exists, the proxy rewrites the request URI to direct it to the local port where the instance is running.

The **Monitor** is formed by the collector and scheduler components, it's purpose is to periodically fetch system information and send it to the controller by HTTP requests. Controller components later use the data acquired to manage active workers' states. System information is also used to decide which worker will receive future services.

The collector gathers system information e.g. CPU usage, memory usage, and total memory of a worker node. Since processing time continuously accumulates from system boot, assessing usage at a single point in time provides poor insight into system performance. To accurately gauge CPU usage, the difference in CPU time between two points within the desired interval is computed. The CPU usage in the percentage formula is described as $U = UT/TT$ where `UT` is the used processing time and `TT` is the total processing time measured in seconds.

The **Runner** manages the lifecycle of container instances. Upon receiving a deployment request, the runner allocates an available port on the machine and initiates the deployment via the Docker API. This involves preparing the runtime environment and starting the container with the given configuration. The runtime allocates necessary resources such as CPU shares and memory to each container, thus preventing resource contention.

Termination involves graceful shutdown by sending a `SIGTERM` signal. If the container does not terminate within a specified timeout, the runtime forcefully kills the container using a `SIGKILL` signal.

5. Results

With the purpose of demonstrating Tucano's capability of balancing workloads amongst multiple computer clusters, the system deployed a CPU bound web service responsible for factorizing large prime numbers and computing their SHA256 hashes, tasks known for their computational intensity. In this context, the performance evaluation for Tucano focuses on its behaviour against large amounts of processing requirements.

The experimental evaluation employed k6[1], a performance testing tool designed to assess systems' reliability and efficiency by simulating various levels of loads. In this experiment, k6 was used to send requests—ranging from small, medium, to

[1] `https://k6.io/`

large ones—to the web service. This enabled the generation of a performance report highlighting how well Tucano managed the distributed tasks under different conditions. The report provided insights into key metrics such as response times, throughput, and resource utilization, allowing for an in-depth analysis of the system's capacity to handle stress and scale effectively. All tests were performed using a MacBook Pro with M3 pro chip, 12 cores, 18GB of RAM and 512GB SSD. Three scenarios were examined:

- **Case 1** ran the service in a container with 1 vCPU and 512 MB of RAM, without use of Tucano.
- **Case 2** used the Tucano ecosystem with a container cluster of three workers and a controller, each allocated 0.25 vCPU and 128 MB of RAM.
- **Case 3** also used the Tucano environment with the same cluster configuration, but all containers had the same resource allocation as in Case 1.

Figure 3.

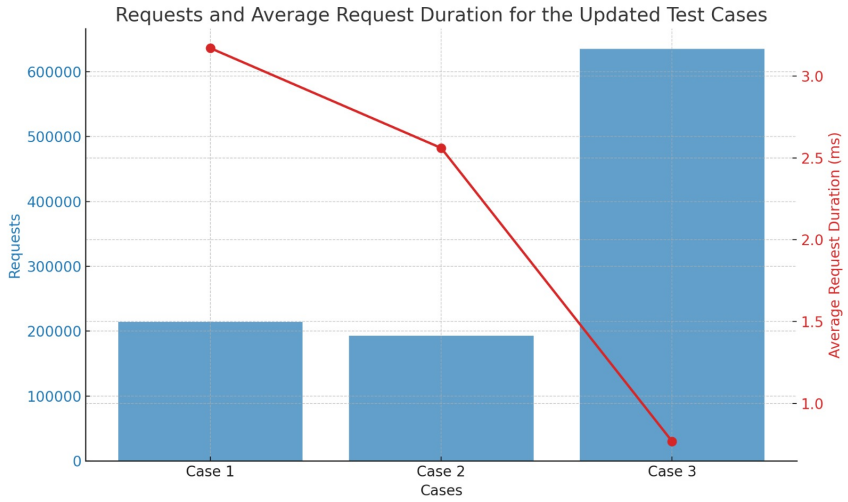


Figure 3 shows a significant improvement in performance from Case 1 to Case 3, not only the number of requests went from 214,484 in Case 1 to 635,084 in Case 3, and the average request duration decreased from 3.17 ms to 769.9µs. Case 2 also reduced the request duration, despite the lower individual resource allocation per container, resulting on a smaller amount of requests during execution.

The reduction in request duration underlines the capabilities of the system's resource management and load balancing mechanisms. Tucano's distributed architecture minimizes latency, even when the system handles large volumes of requests. Tucano is well-optimized to allocate resources across multiple nodes, leveraging parallelism to ensure lower response times.

While the increase in the number of machines naturally contributes to better performance, these results also demonstrate Tucano's efficiency in managing and distributing workloads. The system's ability to optimize resource usage and load balance across multiple nodes highlights its scalability and robustness. By leveraging parallelism and distributed computing, Tucano enhances system performance, making it a valuable solution for computationally intensive applications.

6. Conclusion

While virtualization gain resource isolation and management, it introduces overhead by emulating complete OSs and hardware environments. Containers offer a more lightweight solution by virtualizing only the OS, however, managing many containers needs scheduling and load-balancing mechanisms to ensure system performance.

This paper proposes Tucano, a service scheduler and load balancer. Tucano integrates the aforementioned principles by distributing workloads across a network of interconnected physical servers, each running containerized applications. Its architecture, with controller and worker nodes, simplifies service deployment, monitoring, and dynamic load balancing. By leveraging physical distribution and containerization, Tucano offers a scalable, efficient, and cost-effective solution for managing intensive workloads. It reduces resource overhead and performance issues of traditional virtualization, pointing to a promising future for distributed system management and application deployment. Tucano is available under the GPL-3.0 license [2]

Future work aimed at enhancing the performance and flexibility of Tucano can be outlined. Key areas for improvement include creating a multi-controller cluster to increase system availability; exploring advanced load balancing techniques to optimize resource allocation; and researching the adoption of broader container standards versus swapping containers for unikernels to minimize resource consumption and maximize safety. These enhancements will build on the foundational principles established in this work, advancing the capabilities of distributed service schedulers and load balancers in contemporary computing environments.

References

[Anand et al. 2021] Anand, A., Chaudhary, A., and Arvindhan, M. (2021). The need for virtualization: when and why virtualization took over physical servers. In *Advances in Communication and Computational Technology: Select Proceedings of ICACCT 2019*, pages 1351–1359. Springer.

[Armstrong 2003] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden. Final version with corrections — last update 20 November 2003.

[Bermejo and Juiz 2020] Bermejo, B. and Juiz, C. (2020). Virtual machine consolidation: a systematic review of its overhead influencing factors. *The Journal of Supercomputing*, 76(1):324–361.

[Bernstein 2014] Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

[Burns et al. 2016] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *ACM Queue*, 14:70–93.

[Kovvur et al. 2013] Kovvur, R. M. R., Ramachandram, S., Kadappa, V., and Govardhan, A. (2013). A distributed dynamic grid scheduler for mixed tasks. In *2013 3rd IEEE International Advance Computing Conference (IACC)*, pages 110–115.

[2]Tucano repository: `https://github.com/esfericos/tucano`.

[Merkel et al. 2014] Merkel, D. et al. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2.

[Mirhosseini et al. 2019] Mirhosseini, A., Sriraman, A., and Wenisch, T. F. (2019). Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198. IEEE.

[Morabito et al. 2017] Morabito, R. et al. (2017). Evaluating performance of containerized microservices for iot applications. *IEEE Transactions on Services Computing*, 10(5):1–14.

[Shirinbab et al. 2017] Shirinbab, S., Lundberg, L., and Casalicchio, E. (2017). Performance evaluation of container and virtual machine running cassandra workload. In *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*, pages 1–8. IEEE.

[Speitkamp and Bichler 2010] Speitkamp, B. and Bichler, M. (2010). A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE Transactions on Services Computing*, 3(4):266–278.

[Watada et al. 2019] Watada, J., Roy, A., Kadikar, R., Pham, H., and Xu, B. (2019). Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7:152443–152472.