Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration

Merlijn Sebrechts, Thomas Vanhove, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert and Filip De Turck
Department of Information Technology, Ghent University - iMinds
Tech Lane Ghent Science Park Campus A, 15, 9052 Ghent, Belgium
Email: merlijn.sebrechts@intec.ugent.be

***Abstract*—Both researchers and industry players are facing the same obstacles when entering the big data field. Deploying and testing distributed data technologies requires a big up-front investment of both time and knowledge. Existing cloud automation solutions are not well suited for managing complex distributed data solutions. This paper proposes a distributed service orchestration architecture to better handle the complex orchestration logic needed in these cases. A novel service-engine based approach is proposed to cope with the versatility of the individual components. A hybrid integration approach bridges the gap between cloud modeling languages, automation artifacts, image-based schedulers and PaaS solutions. This approach is integrated in the distributed data experimentation platform Tengu, making it more flexible and robust.**

I. Introduction

Big data is a fast-moving research field that has resulted in an explosion of distributed data technologies. The distributed nature and the complexity of these data technologies creates a lot of operational overhead. Deploying, managing and monitoring big data solutions is a very time-consuming and knowledge-intensive task. Hiring the right people to manage such an infrastructure is hard because of the data skills shortage [1][2]. The DevOps movement has spawned a number of tools that help reduce the operational overhead and empower small teams to manage complex cloud infrastructure.

There are a number of operational challenges specific to distributed data technologies that are not addressed in the state of the art cloud automation tools. The first issue lies in the **orchestration** of multiple components. The centralized and monolithic nature of most orchestration tools cannot handle the complex orchestration logic needed for deploying and managing distributed technologies. The second challenge resides in the **management** of an individual component. The current state-machine based approach for automating the management of an individual component is not flexible enough to meet the needs of adaptable services. The last issue addressed in this paper lies in the **integration** between model-based orchestrators and existing automation artifacts. There is a need for integrating a variety of existing PaaS and SaaS solutions, configuration management tools and model-based orchestrators.

This paper further explores these three challenges and propose a novel architecture to tackle them. Section II describes where existing state of the art techniques fall short. The distributed service orchestrator is explained in section III. Section IV goes into details about the Service Agent and Service Engine. Section V proposes the novel hybrid integration approach. Finally, Section VI reflects on the work and gives a peek into future work.

II. State of the art

A. Orchestration

With IT infrastructure growing ever-more complex, the dependencies between individual components are becoming hard to manage. One of the key properties of an orchestrator is that it enables isolation of components and autonomic management of dependencies between these components. The conventional way of orchestrating large clouds includes a central orchestrator such as seen in the OASIS standard Topology and Orchestration Specification for Cloud Applications (TOSCA) [3]. Research at Google has shown that orchestrating and scheduling a datacenter from a central monolith becomes too complex to maintain [4]. The Omega and Mesos [5] meta-schedulers are proposed as a solution for this complexity, implementing a decentralized scheduling approach. Their orchestrating capabilities, however, are not as extensive as needed for cloud modeling languages such as TOSCA or Ubuntu Juju[1]. The decentralized orchestrator proposed in this paper solves the complexity issues of a centralized approach while still maintaining the full features of a cloud modeling language.

B. State machines and lifecycles

Individual components in TOSCA are described using a lifecycle described by a state machine [6]. This approach has the disadvantage that it is quite inflexible because all possible states and transitions have to be known at design time. Moreover, the state machine of a component can become very complex when a component can have a number of different relationships with other components. Each possible combination of those relationships needs its own state. The amount of states needed to model such a tool grows exponentially in function of the number of

[1]https://jujucharms.com/

possibilities. This is especially relevant in the distributed data field where components such as an Extract, Transform, Load (ETL) tool can have a number of different relationships to a multitude of data sources and to many different data stores. The service engine proposed in this paper addresses the complexity problem of state machines.

C. Integration

Integration between model-driven cloud management and configuration management is key to automate every aspect of the management of a service [7]. Wettinger et al. propose two approaches to integrate model-driven cloud management and configuration management: *direct integration* and *transparent integration*. For direct integration, the service orchestrator has to have knowledge about the particular configuration management tool. Transparent integration can be accomplished using generic 'script' artifacts that translate generic calls of the orchestrator into configuration-management-specific calls. Wettinger et al. note that although the transparent integration approach is more portable, the direct integration makes it possible to fully leverage all features of the configuration management tool. To get the full benefits of both approaches, Wettinger et al. propose a combined approach. This however has the downside that it requires some code duplication and that it requires rigorous testing to ensure that both methods will yield the same result. This paper proposes a hybrid approach that combines the best of direct and transparent integration without the added complexity and duplicated code.

III. Distributed orchestration

To tackle the issues encountered in centralized, monolithic orchestrators, this paper proposes a novel decentralized service orchestration architecture. This architecture enables very complex orchestration logic without the complexity of a monolith. Moreover, this decentralized approach also tackles the scalability and fault-tolerance issues of centralized orchestrators. The orchestration layer consists of a number of service agents and a number of peer-to-peer relationships between them as shown in Figure 1. Each service agent is responsible for a particular service. Using the relationships, service agents communicate to achieve higher-level goals. An example is a Apache Hadoop[2] cluster. The namenode, datanode and slaves each have their own service agent. The service agents communicate with each other to setup and configure the Apache Hadoop cluster.

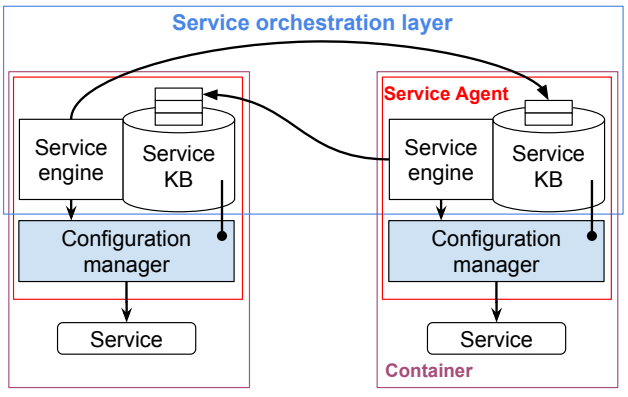

Figure 1. Decentralized service orchestration architecture.

Service agents operate event-driven and transactional, resulting in an eventually consistent model [8]. Service agents communicate by sending relationship events to each other. Each relationship event updates the knowledgebase shared between the two services. When a service agent receives a relationship event, it queues the event on a FIFO queue. One by one, each relationship event gets applied to the knowledgebase and is processed by the service engine in a transactional manner. The service engine can call a configuration manager to update the service or send events to the peers of the service agent. The event-driven approach allows for complex dependencies. Service agents can negotiate with each other on topics such as how the services should be configured and what libraries should be installed.

IV. The Service Engine

The actual management of the service and the communication with other service agents happens in the service engine. In a conventional configuration management approach, the operator requests a certain state and the configuration manager takes the correct actions to get into that state. This proactive approach requires the desired state to be known beforehand. In the presented approach, actions handle changes in the state of the service and its surroundings. The service agent monitors changes and executes the required actions (handlers) to react to those changes. Each handler modifies a small part of the service state, causing other handlers to be executed. When no more handlers are executed, the service has reached a consistent state, which causes the service agent to go back into monitoring mode until it detects another change. When a handler fails, a rollback of the transaction is issued and the service engine retries the processing of the relationship event. This convergence-based approach, as proposed by Wettinger et al. [9] eliminates the need for idempotency and convergence, and is more fault tolerant.

The state of the service is represented as a set of facts in a local knowledge base. The heart of the service agent is the service engine. The service engine contains a set of handlers and decides which handlers should be executed based on the facts in the knowledge base and the preconditions of the handlers. A handler can add facts to the knowledge base, communicate with related service agents and change the service state. There are four types of preconditions that a

[2]https://hadoop.apache.org/

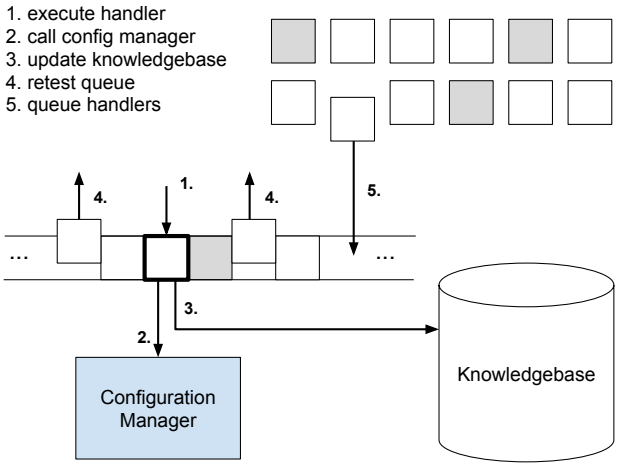

Figure 2. A service engine run. After the handler updates the knowledgebase, the queue is retested and handlers with false preconditions are removed. Handlers that react to events are automatically removed from the queue after execution. Handlers whose preconditions are met, are added to the queue. A handler that has an event precondition that is not active (greyed out) cannot be added to the queue.

handler can specify: a relationship fact, a relationship event, an internal fact and an internal event.

The difference between *events* and *facts* is that an event is only relevant during a single moment in time while a fact remains true until a handler changes it. An example of an event is a file change. When a file changes, all handlers that react to that event get queued to execute. After all events are queued, the event becomes inactive again. An example of a fact is the state of a package: installed or uninstalled. This means that if a handler reacts to two facts and one event, it will only get queued if both facts are true at the moment the event is fired. Queued handlers are retested after each handler run to ensure that their preconditions based on facts are still true as seen in Figure 2. Preconditions based on events are not retested.

The difference between *internal* and *relationship* is that relationship events are processed as a transaction. When a relationship change or event happens, the service engine starts a *run*. Relationship changes that happen during such a run get queued until the run completes. From the outside perspective, this run happens during a single moment in time. This means that the relationship information is guaranteed to be consistent, but not current. However, the queuing of relationship events guarantees eventual consistency over all service agents.

This eventually consistent model works great if the goal is to let the infrastructure converge to a globally consistent state. Where this falls short is when the actions of a service agent depend on the real-time current state of another service agent. An example of this would be a inter-cluster data copy between two Hadoop clusters using *distcp*. For this tool to work, both clusters need to be ready when the distcp is started. Since the state a service agent sees is not necessarily the current state, but could have been a current state in the past, a service agent cannot reliably orchestrate the start of a distcp. Future work will expand this architecture with distributed locking capabilities to tackle this challenge.

V. Hybrid Integration

The distributed orchestrator allows for a novel approach to integrate cloud modeling languages, configuration management tools, image-based orchestrators and PaaS solutions. This approach can be called *hybrid integration* since it combines the best of direct- and transparent integration without the downside of duplicated code and without the added complexity. This solution is portable, yet able to fully leverage advanced features of the configuration management tool. Service agents communicate using generic language-independent interfaces, making the implementation of a service agent transparent to other service agents. The internals of a service agent, namely a service engine, can however directly integrate into configuration management tools. This allows them to leverage the full capabilities of the specific tool. Existing automation artifacts can be easily encapsulated in a service agent allowing for reuse of the expertise in existing code. Figure 3 shows an example of a Chef[3] script encapsulated in a service agent. This service agent can communicate with other service agents independent of the underlying configuration management tool.

This approach is tested by applying this architecture to Tengu, a distributed data experimentation platform [10].

[3]https://www.chef.io/

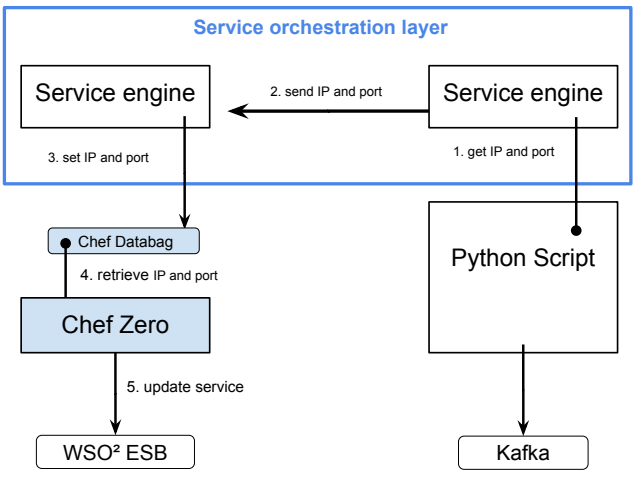

Figure 3. The distributed service orchestration layer handles the communication between service agents. The service engine translates the received information into a format the Chef Zero configuration manager can read: a Chef Databag. Chef then updates service configuration to enable communication from the ESB to Kafka.

Tengu is built to give researchers an easy way to spin up distributed data environments. The first version of Tengu was built using the configuration management tool Chef. The complexity of the dependencies between individual services caused Tengu to be very inflexible. Adding a service orchestration layer to Tengu, turned it into a flexible and composable big data experimentation platform[4].

The hybrid integration approach also allows for integration with image-based deployments such as containers and virtual machines, cloud schedulers, PaaS and SaaS solutions. The biggest drawback of image-based deployment is the lack of flexibility. This can be addressed by creating a service agent that deploys a prebuilt image. After the deployment, the service engine changes the configuration of the deployed image to accommodate the needs of related service agents. When a related service agent needs information about the deployed service, the service engine uses introspection to gather that information from the image. Integration with PaaS and SaaS solutions is similarly straightforward. The service agent can communicate to a PaaS or SaaS solution using an API. A service agent that has an API client as back-end can be interchanged with a service agent that has a configuration management tool as back-end if both service agents implement the same relationship interface. This allows users to choose the solution that fits their use-case best, be it self hosted, SaaS or PaaS. As an additional bonus, this further limits vendor lock-in problems.

VI. Conclusion and future work

The distributed service orchestrator presented in this paper tackles three problems. Its distributed nature reduces its complexity while maintaining the ability to orchestrate complex distributed interdependent services. The service engines allow for orchestrating very versatile service agents without the need for complex state machines. Lastly, the hybrid integration makes deep integration possible between the cloud modeling language and configuration management tools, image based deployers and PaaS solutions. Future work will further expand on and evaluate this architecture. This architecture will be extended with multi-tenant capabilities, a distributed monitoring framework, distributed reasoning about higher-level objectives, and real-time locking of resources.

Acknowledgment

The research described in this paper is partially funded by the IWT/VLAIO agency through the DeCoMAdS project grant.

References

[1] Gartner, "Gartner Survey Highlights Challenges to Hadoop Adoption," May 2015. [Online]. Available: https://www.gartner.com/newsroom/id/3051717

[2] TEKsystems, "Lowered Expectations for IT Budgets 2015 | TEKsystems," Dec. 2014. [Online]. Available: https://www.teksystems.com/en/resources/news-press/2014/teksystems-annual-it-forecast-2015?&year=2014

[3] OASIS, "Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification 01," 2013.

[4] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 351–364.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[6] P. Lipton, S. Moser, D. Palma, M. Rutkowski, and T. Spatzier, "TOSCA Simple Profile in YAML Version 1.0," Aug. 2015. [Online]. Available: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd01/TOSCA-Simple-Profile-YAML-v1.0-csprd01.html#_Toc430015763

[7] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbcher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, "Integrating Configuration Management with Model-driven Cloud Management based on TOSCA." in *CLOSER*, 2013, pp. 437–446.

[8] P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, May 2013.

[9] J. Wettinger, U. Breitenbcher, and F. Leymann, "Compensation-Based vs. Convergent Deployment Automation for Services Operated in the Cloud," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Springer Berlin Heidelberg, Nov. 2014, no. 8831, pp. 336–350, dOI: 10.1007/978-3-662-45391-9_23.

[10] T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck, B. Vermeulen, and P. Demeester, "Tengu: An Experimentation Platform for Big Data Applications," in *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Jun. 2015, pp. 42–47.

[4] https://github.com/IBCNServices/tengu-charms