Middleware for Dynamic Upgrade Activation and Compensations in Multi-tenant SaaS

Dimitri Van Landuyt(✉), Fatih Gey, Eddy Truyen, and Wouter Joosen

imec-DistriNet, Department of Computer Science, KU Leuven,
Celestijnenlaan 200A, 3001 Leuven, Belgium
{dimitri.vanlanduyt,fatih.gey,eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract. Multi-tenant Software as a Service (SaaS) is the cloud computing delivery model that maximizes resource sharing up to the level of a single application instance, servicing many customer organizations (tenants) at once. Due to this scale of delivery, a SaaS offering, once successful, becomes difficult to upgrade and evolve without affecting service continuity, and this in turn limits its capabilities to respond to the reality of changing customer requirements.

However, not all tenants are equal, and to some organizations such disruptions are more costly than to others. Supporting different quality trade-offs for different tenants is often a manual, error-prone task and far from trivial.

This short paper outlines our middleware design for fine-grained, gradual and continuous evolution of multi-tenant SaaS applications, providing automated and systematic support for (i) tenant-aware upgrade enactment, and (ii) compensations that allow recovering from negative side-effects of the upgrade enactment.

1 Introduction

In the Software as a Service (SaaS) delivery model, Internet services are offered to customer organizations (tenants) on a subscription basis. The SaaS provider and tenants typically agree on individual service quality levels that such an application must reliably provide.

A key advantage of SaaS applications is their *cost-efficiency* which is attained at large scale due to economies-of-scale effects [2]: Run-time resources (such as the hardware, platforms and supportive services) are shared among multiple tenants up to the level of application instances (an architectural tactic called multi-tenancy [7]). To minimize the costs per tenant, configuration and customization activities are commonly outsourced to tenant administrators, a principle called *self service* [25].

Such a multi-tenant SaaS application becomes difficult to change and evolve without affecting overall service continuity and thereby many tenant businesses. As a result, its capabilities to respond to the reality of changing customer requirements [21] (for example, through *continuous delivery* [24]) are limited. More specifically, a SaaS application that is expected to attain high levels of service

M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 340–348, 2017.
https://doi.org/10.1007/978-3-319-69035-3_24

continuity cannot be taken offline for maintenance, i.e. to enact an upgrade, but must continue servicing tenant requests even during upgrade enactment. In addition, due to the high level of resource sharing among tenants, it must be ensured that changes applied for one tenant do not negatively affect other tenants (tenant isolation). Furthermore, service continuity cannot always be ensured (e.g. during the enactment of an incompatible upgrade [4]), such that either the SaaS application becomes temporarily unavailable, or different service qualities are sacrificed, for example functionality and integrity.

An upgrade enactment that maintains one metric of service continuity at the cost of another provides a specific *quality compromise*. A multi-tenant SaaS application that traditionally evolves in one shot [4,11] has no room for considerations on a per-tenant basis. Moreover, in this context, the large scale of operation of the SaaS application has a multiplying effect, which leads to upgrades that potentially have a profound impact on many tenant businesses. This renders traditional approaches such as waiting for application-wide quiescence [19] unfeasible. As different compromises (in terms of quality or functionality) may be considered acceptable to different tenants (depending for example on the tenant SLA), systematic support is required for compromises on a per-tenant basis, both *during* the upgrade enactment and/or *after* the enactment, i.e. by supporting compensatory measures that are enacted after the fact (e.g. rolling back inconsistent transactions). This short paper presents middleware support for continuous evolution of multi-tenant SaaS applications that provides support for both types of compromises on a fine-grained, per-tenant basis.

The remainder of this paper is structured as follows: Sect. 2 derives and motivates the main requirements, whereas Sect. 3 presents our middleware. Section 4 discusses related work, and Sect. 5 concludes the paper.

2 Motivation and Requirements

The following key observations contribute to our motivation: (i) Incompatible software upgrades demand for different quality compromises with respect to the upgrade enactment process; (ii) related work on dynamic software upgrades and dynamic adaptations provides several alternative strategies [1,19,22,23,28], each involving fundamentally different quality compromises (e.g. consistency over availability or vice versa); (iii) to some tenants, software failures (as a cost of a quality compromise) are only harmful for tenants if their effects remain permanent, and when anticipated, such negative consequences can often be corrected though compensatory measures.

The above observations highlight the potential to perform evolution of multi-tenant on a per-tenant, customized manner, but current solutions either involve enacting upgrades in a single shot operation, or require manual effort and are therefore error-prone and expensive.

As such, we state the following requirements for supporting continuous evolution of multi-tenant SaaS applications in a systematic and maximally automated fashion:

R1 Customization support: The nature of the SaaS service degradation and service quality compromises should be customizable and controllable by tenants. This entails:

R1a Tenant-isolated upgrade enactment: Allowing the activation of an upgrade for one tenant without affecting other tenants (tenant isolation) is a key enabler for fine-grained per-tenant customization, as this allows the activation of an upgrade for one tenant to be timely and functionally decoupled from other tenants [14]. It enables, moreover, approaches such as phased cut-off, i.e. to overlap the phase-out of the current version and the phase-in of the upgraded version of service components.

R1b Awareness of the upgrade compatibility: Alternative upgrade paths that each involve different compromises (in terms of quality and functionality) must exploit the compatibility nature [4] of upgrades. This implies in particular that different upgrade activation mechanisms must be developed and supported by the SaaS developer.

R2 Compensation support: For each quality compromise made during upgrade enactment for which significant service degradations are anticipated, automated compensation facilities should be provided that revert or counteract these, again on a per-tenant basis, in isolation and tailored to the nature of the upgrade at hand (thus, provided by the SaaS developer and configured by the tenant administrator).

3 Middleware Support

Fig. 1 provides an overview of the proposed middleware solution. The top of the figure represents the SaaS application which is structured as a service-oriented application (SOA). Section 3.1 first introduces the `DSlookup` component, which supports tenant- and context-aware dynamic service composition [14,29]. The `Activation Controller` and `Compensation Controller` components both rely extensively on this component and are discussed in Sects. 3.2 and 3.3 respectively.

3.1 Dynamic Multi-tenant Service Composition

Our middleware relies extensively on the underlying ability to manipulate service compositions at run time, and we leverage this mechanism for tenant-aware customization of service bindings [29].

Dynamic service lookup is accomplished by the `DSlookup` component that implements a lazy service composition approach: it resolves only to a specific service binding of a composition at request time. In addition, `DSlookup` allows manipulating its service lookup logic through changing the transaction context of the triggering service request. Multi-tenant customization is accomplished by defining a set of service compositions that serve specific variants of services specific to a tenant (these are part of the tenant configurations, stored in the `Tenant Configuration Repository`) (step (0) in Fig. 2). Tenant context tokens [18,29]

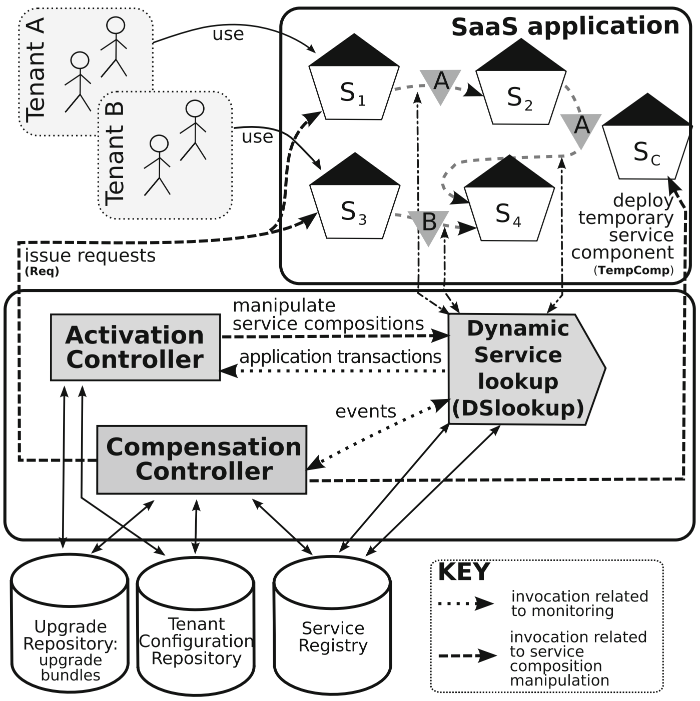
Tenant A
Tenant B
use
use
SaaS application
S_1
S_2
S_3
S_4
S_C
A
A
B
deploy temporary service component (TempComp)
issue requests (Req)
Activation Controller
manipulate service compositions
application transactions
Dynamic Service lookup (DSlookup)
events
Compensation Controller
Upgrade Repository: upgrade bundles
Tenant Configuration Repository
Service Registry
KEY
invocation related to monitoring
invocation related to service composition manipulation

Fig. 1. High-level overview of our middleware, which is comprised of the `DSlookup` component, the `Activation Controller`, and the `Compensation Controller`.

are attached to the call chain when new application transactions are started, and the tenant identity is derived, for example, from authentication data [29].

To allow ensuring version-consistent behavior [23], the dynamic service composition must additionally be aware of the end-to-end application transaction context. This is done with a transaction context token that is attached to all service requests of an application transaction. A service component instance that issues outbound service requests r_O in the course of processing an inbound request r_I must copy the tenant and transaction context of r_I to r_O [18].

Figure 2 illustrates the workings of the `DSlookup` component. A service component instance addresses `DSlookup` to lookup another service component instance that provides a specific interface, attaching the tenant and transaction context token (step (1) in Fig. 2). To fulfill the request, `DSlookup` consults the corresponding tenant configuration from the tenant configuration repository (step (2*b*)), specifically to find a matching *service binding*. If successful, the reference to an instance[1] of the specified target service is returned to the caller (return arrow for step (1)) who now is able to invoke that service call ((3) in Fig. 2). It is worth noting that service instances are identified by an identifier/type *and* a version.

[1] This involves consulting the `Service Registry`, which is omitted here for simplicity.

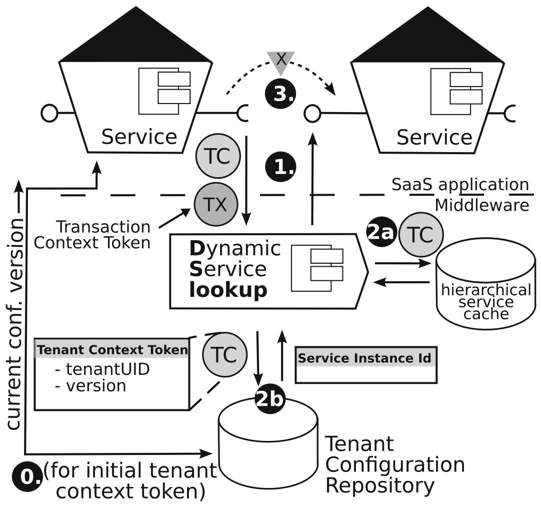

Fig. 2. Dynamic context-aware service composition using **D**ynamic **S**ervice **lookup**.

Service cache. For performance and scalability reasons, each service component instance caches service instance identifiers in a hierarchical cache. The cache is queried in the reverse hierarchy order (step (2*a*) in Fig. 2): only if transaction-specific service instance references cannot be found, generic references are searched.

3.2 Activation Support

Our middleware enables configuration of upgrade activations dynamically and on a per-tenant basis. Upgrade activation is accomplished by dynamically manipulating service compositions, to reroute service lookups to new service versions. As shown in Fig. 1, the key component for coordinating these upgrade activations is the `ActivationController`.

More specifically, the possible upgrade paths for a specific upgrade are encoded in a set of *Activation scripts* (provided by the SaaS developer), and these can be selected or configured by the tenant or SaaS operator. These are code artifacts that are defined in terms of pre-defined service composition manipulation primitives. The following manipulation primitives are currently supported by the middleware:

`InitVer` **Change version for initial tenant context**: this primitive provides the capability to change the tenant configuration version used for initial tenant contexts which is stored in the tenant configuration repository. This effectively means that for all tenants that do not refer to a specific version in their service compositions, the newer version will picked as a default.

`TokenVer` **Change version of tenant context token:** with this primitive, at `DSlookup`, the configuration version entry of a tenant context token can be manipulated for specific service-lookup queries *before* the actual lookup. This effectively overrides the selected version.

`FailLookup` **Deliberately fail service lookup:** using this primitive, `DSlookup` can be set to fail a specific lookup deliberately, i.e. to return that no service component instances are available.

`FlushSC` **Flush service reference cache:** the service cache (maintained by every service component instance locally) can be cleared for transaction-specific or generic service instance references, e.g. to immediately effectuate a version upgrade.

As depicted in Fig. 1, the `Activation Controller` monitors new application transactions (beginning and end), and coordinates the execution of Activation scripts, which in turn entails the invocation of the service composition manipulation primitives discussed above.

3.3 Compensation Support

A compensation is essentially an additional behavior to prevent or recover from a negative side-effect of upgrade enactment. Similarly to the `Activation Controller`, the `Compensation Controller` actively monitors the application transactions and perform actions in response to specific events. As with the activation controller scripts, a compensation is put together with *Compensation Primitives.* The following compensation primitives are currently supported:

`ManipSC` **Manipulate service composition instance:** this primitive is equivalent to the `TokenVer` primitive discussed earlier.

`FailReq` **Deliberately fail service requests:** similarly, this primitive is already supported by the `FailLookup` primitive.

`TempComp` **Deploy temporary service components:** an upgrade may be shipped with temporary service components that are only deployed during the activation of an upgrade by a compensation artifact (for example, to attain graceful degradation).

`Req` **Issue service requests:** a compensation may issue additional service requests, for example to start new transactions on behalf of the end user.

A *Compensation script* consists of two key elements: one for specifying events it may have interest in (the **monitor**), and one for defining the appropriate reaction to these events (**action**). Event filters are installed at the `DSlookup` component at the start of an upgrade. Event filters may refer to service component instances involved, tenant context used at the beginning and the end of the service lookup[2], and the application transaction context.

Relevant events are propagated from `DSlookup` to the `Compensation Controller`, which in turn coordinates the execution of the corresponding Compensation script.

[2] Note: these two may differ when `TokenVer` is used.

4 Related Work

We first discuss the broader set of related work on dynamic software updates (DSU), then we focus on existing support for evolution or customization of cloud applications and finally, we discuss related work w.r.t. compensation support.

Dynamic Software Updates. Updating an application at run time has been studied for decades [3,15,19], increasingly reducing the impact on its normal operation. There are two dominant and fundamentally different approaches:
(i) *Dynamic software updates* [16] score well on service continuity and focus on update safety, but are limited to specific types of upgrades; moreover, they usually depend on memory-invasive operations which are not applicable in a cloud context.
(ii) *Dynamic Adaptation* techniques [1,17,19,22,23,28] are applied in terms of components and connectors and are therefore applicable for any type of upgrade and technology-independent. These techniques can be further divided in two classes: those that require a safe state (e.g. quiescence) of the application before performing the upgrade [19,28], and those that support a mixed mode where old and new versions co-exist [1,17,26].

Although showing this in further detail is part of our future work, using the manipulation primitives for service composition presented in Sect. 3.2, we can effectively support these different classes of upgrade strategies simultaneously. Similar to [17,22,27], our middleware provides an open and versatile platform for upgrades, specific to the domain of cloud-scale evolution [4].

Middleware Support for Evolution of Cloud Applications. Dumitras et al. [11] propose a middleware that moves an entire application to a "parallel universe" to avoid inconsistencies of otherwise incremental upgrades of enterprise-sized cloud applications. Opposed to theirs, our approach promotes a service component as the smallest unit for evolution. Others [13,20] support adaptation and evolution of a SaaS application for anticipated upgrades. Ertel et al. [12] present a framework to support dynamic evolution of dataflow programs. While their support is based on types of applications that are different from multi-tenant SaaS applications, their work is complementary to ours as it focuses on algorithms for automated enactment, accounting for state-transfer, referential integrity and timeliness of dependent upgrades.

Compensations. In relational database transactions [8,9], supporting compensations is a strategy for *forward error recovery* which is an alternative to *backward error recovery* (i.e. roll-back). *Automatic workaround* [5,6,10] as a related self-healing tactic on the other hand provides a computed recovery strategy as a compensation.

Although showing this in further detail is part of our future work, using the compensation primitives defined in Sect. 3.3, we support the following types of compensations: (i) inverting or repeating service requests (`Req`), (ii) changing behavior for specific requests (`ManipSC` and `TempComp`), (iii) deliberately failing service requests (`FailReq`).

5 Conclusion

We have presented dedicated middleware support for continuous evolution of multi-tenant SaaS applications that essentially implements two measures to reduce the impact of –or at least to increase control over– an upgrade enactment: customization and compensation. Our middleware allows per-tenant, customized and fine-grained service continuity compromises when enacting different types of software upgrades. Compromises that entail a significant sacrifice are complemented by a *compensation* to alleviate their effect in an automated fashion. Our approach allows the SaaS developer to implement upgrade activation and compensation scripts that are based on common manipulation and compensation primitives respectively that are built into the middleware.

The systematic support for both types of measures allow controlling the overall cost of enacting a change (in the course of software evolution) in multi-tenant SaaS applications that are subject to continuous service delivery guarantees, and as such these mechanisms may contribute greatly in reducing the time-to-market of new features.

Acknowledgements. This research is partially funded by the Research Fund KU Leuven, the ADDIS research program funded by KU Leuven GOA, and the DeCoMAdS SBO strategic research project.

References

1. Ajmani, S., Liskov, B., Shrira, L.: Modular Software Upgrades for Distributed Systems. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 452–476. Springer, Heidelberg (2006). doi:10.1007/11785477_26
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
3. Bloom, T., Day, M.: Reconfiguration and module replacement in argus: theory and practice. Softw. Eng. J. **8**(2), 102–108 (1993)
4. Brewer, E.: Lessons from giant-scale services. IEEE Internet Comput. **5**(4), 46–55 (2001)
5. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds for web applications. In: FSE 2010, pp. 237–246. ACM, New York (2010)
6. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: SEAMS 2008, pp. 17–24. ACM, New York (2008)
7. Chong, F., Carraro, G.: Architectural strategies for catching the long tail (2006). http://msdn.microsoft.com/en-us/library/aa479069.aspx
8. Colombo, C., Pace, G.J.: Recovery within long-running transactions. ACM Comput. Surv. **45**(3), 28:1–28:35 (2013)
9. Davies Jr., C.T.: Recovery semantics for a db/dc system. In: Proceedings of the ACM Annual Conference, pp. 136–141. ACM, New York (1973)
10. de Lemos, R., et al.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35813-5_1

11. Dumitraş, T., Narasimhan, P.: Why Do Upgrades Fail and What Can We Do about It? In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 349–372. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10445-9_18
12. Ertel, S., Felber, P.: A framework for the dynamic evolution of highly-available dataflow programs. In: Middleware (2014)
13. García-Galán, J., Pasquale, L., Trinidad, P., Ruiz-Cortés, A.: User-centric adaptation of multi-tenant services: Preference-based analysis for service reconfiguration. In: SEAMS (2014)
14. Gey, F., Van Landuyt, D., Joosen, W., Jonckers, V.: Continuous evolution of multi-tenant saas applications: a customizable dynamic adaptation approach. In: PESOS, May 2015
15. Gupta, D., Jalote, P., Barua, G.: A formal framework for on-line software version change. Softw. Eng. **22**(2), 120–131 (1996)
16. Hayden, C.M., Magill, S., Hicks, M., Foster, N., Foster, J.S.: Specifying and verifying the correctness of dynamic software updates. In: Verified Software (2012)
17. Hillman, J., Warren, I.: An open framework for dynamic reconfiguration. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 594–603. IEEE Computer Society, Washington (2004)
18. Jørgensen, B.N., Truyen, E.: Evolution of Collective Object Behavior in Presence of Simultaneous Client-Specific Views. In: Konstantas, D., Léonard, M., Pigneur, Y., Patel, S. (eds.) OOIS 2003. LNCS, vol. 2817, pp. 18–32. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45242-3_4
19. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. Softw. Eng. **16**(11), 1293–1306 (1990)
20. Kumara, I., Han, J., Colman, A., Kapuruge, M.: Runtime Evolution of Service-Based Multi-tenant SaaS Applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 192–206. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45005-1_14
21. Lehtonen, T., Suonsyrjä, S., Kilamo, T., Mikkonen, T.: Defining metrics for continuous delivery and deployment pipeline. In: Symposium on Programming Languages and Software Tools (2015)
22. Li, W.: Evaluating the impacts of dynamic reconfiguration on the qos of running systems. JSS **84**(12) (2011). http://www.sciencedirect.com/science/article/pii/S0164121211001439
23. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., Lu, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: FOSE (2011)
24. Neely, S., Stolt, S.: Continuous delivery? easy! just change everything (well, maybe it is not that easy). In: Agile Conference (AGILE), 2013, pp. 121–128 (2013)
25. Sun, W., Zhang, X., Guo, C.J., Sun, P., Su, H.: Software as a service: Configuration and customization perspectives. In: Services (2008)
26. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jorgensen, B.: A dynamic customization model for distributed component-based systems. In: Distributed Computing Systems Workshop, pp. 147–152, April 2001
27. Truyen, E., Janssens, N., Sanen, F., Joosen, W.: Support for distributed adaptations in aspect-oriented middleware. In: AOSD (2008)
28. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. Softw. Eng. **33**(12), 856–868 (2007)
29. Walraven, S., Truyen, E., Joosen, W.: A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 370–389. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25821-3_19