Resilience in dynamic component-based applications

Kiev Gama
Recife Center for Advanced Studies (C.E.S.A.R)
Recife, Brazil
kiev.gama@cesar.org.br

Walter Rudametkin, Didier Donsez
Université de Grenoble
Laboratoire d'Informatique de Grenoble
Grenoble, France
{firstname}.{lastname}@imag.fr

Abstract— **Increasingly, software is required to accommodate new features after the design and deployment stages. Applications are designed to improve their adaptability and flexibility. Software needs to evolve at runtime with minimal interruptions and, when possible, never stop running. Different motivations push software design to allow such evolution at runtime. For example, production systems with critical availability requirements need to be updated with little perceived execution interruption. This paper enumerates challenges in the construction of dynamic component-based applications that are capable of undergoing changes during execution, with minimal impact.**

Keywords: Component-based development, Dynamic reconfiguration, runtime software evolution

I. Introduction

Increasingly, software needs to accommodate new features after being already in use in production environments. It requires the ability to evolve at runtime with minimal interruptions because of a true need for providing non-stop systems, or simply for avoiding users to be annoyed by application restarts [23]. This requirement is especially true with high availability software, which needs to evolve at runtime with minimal interruptions and, when possible, never stop running. Depending on the frequency of updates, patches, changing business requirements, new features, and so forth, this can become a daunting task to handle.

Some applications with critical availability requirements (the so-called critical systems [7]) need to be updated with little perceived execution interruption because application unavailability would lead to consequences such as loss of business, data, infrastructure, etc. These updates may be for different reasons such as changes on business requirements, new functionality added or even bug fixes. Non-critical applications may also present requirements for evolving software at runtime, like end-user applications such as Web browsers, office application suites and mobile applications that need to have the user experience improved with the possibility to easily add new functionality (i.e., plugins) without interrupting application usage. In domains such as ubiquitous computing [27], systems and applications must adapt to continuously changing contexts in an opportunistic manner. Devices, services, and connectivity may appear and disappear at anytime. In such highly dynamic scenarios, applications should be able to adapt their behavior autonomously, being ready to handle failures and unavailability, as well as the appearance of new services, performing the necessary configurations at runtime [8].

Resilience is a desired characteristic for such types of systems that need to evolve during runtime. It consists of maintaining system`s dependability, even when facing changes (e.g., reconfigurations, system updates). By structuring software into modular units with clearly defined roles and interfaces, we facilitate the construction of compatible implementations. A key factor for providing high availability is to modularize the system so modules can be the unit of failure and replacement [10]. By having well separated modules the application can give the impression of having instantaneous repair. By having a tiny mean time to repair (MTTR) the failure can be seen as a delay instead of a failure.

Component-based Development [21] allows using such a modular approach. Replacing parts of an application comes down to choosing a compatible building block and integrating it. Initial approaches using components defined the architecture of the application at design-time and compile-time introducing tight coupling between components, making it difficult to replace them at runtime. Newer techniques have provided mechanisms for achieving this at runtime but still lack a level of flexibility that the component approach tends to inhibit. Although we can find many examples of components that allow dynamic reconfiguration [1][3][17][19][20][26], the challenges of dynamic updates remain the same for all of those approaches. A myriad of questions arises in the utilization of this approach in production systems. Some of the questions that we would like to be able to answer are: What is the impact of a dynamic update? How many components does it directly and indirectly affect? What happens to the components' states? When should we update?

In this paper we are particularly interested in the challenges around practical aspects of creating resilient component-based applications that can be unpredictably reconfigured. That is, those that remain dependable even after changes at runtime. Throughout the text we will use the general term *dynamic reconfiguration* to describe runtime changes to an application's architecture, which may include performing a component update, changing a component binding, installing new components and so forth.

The next sections of this paper are as follows: section II provides motivations and background, section III focuses on the challenges around dynamic updates and section IV draws some conclusions about the discussed issues.

II. Motivations and Background

Before delving into the challenges enumerated in this article, this section presents some motivations and background concerning dynamic component-based applications.

A. High availability requirements

Research reports [5][6] show that systems downtime and data recovery represent major revenue losses for organizations in Europe and in the United States (U.S.). In terms of availability measured by "nines", in the year of 2009, unavailability in Europe represented 99% while in the U.S. 99.9%.

Either due to outages because of failures or because software had to go under maintenance (e.g., module updates, bug fixes), these numbers demonstrate the importance of keeping critical systems up and running without interruption as much as possible. Criticality can be of different types — safety-critical, business-critical, mission-critical or security-critical — but in general, systems are considered as critical when failure or malfunction will lead to significant negative consequence [7]. The increasing complexity and ubiquity on software are transforming critical software into software that is designed to be easily changed, extended and reconfigured [11].

Runtime software evolution (RSE) is appropriate for such types of systems with high availability requirements. The principles behind RSE are of key importance when autonomous critical systems encounter errors during operation, as they must be capable of identifying, detecting, and recovering from errors, potentially without human assistance (error processing) [11]. Fault treatment and error processing are priority tasks in critical systems. Even though eventual operational errors that may be originated during application execution, the frameworks or applications that support RSE also carry potential problems that are inherent of the dynamic update process performed during runtime.

B. Resilience and Runtime Software Evolution

The term resilience [15] has been used in dependable computing as a synonym of fault tolerance. However, in other fields like psychology, ecology or business administration the notion of resilience is related to the capacity of accommodating unforeseen changes. The definition given to resilience in the context of dependable computing is:

> "The persistence of the avoidance of failures that are unacceptably frequent or severe, when facing changes."

Resilience can be seen a sort of scalable dependability, where the goal is continuous dependability when facing changes. This need for resilience is a growing requirement of today's applications that increasingly need to run non-stop. Eventually, applications need to be fixed to accommodate new features or introduce changes in their current behavior. This ability to successfully accommodate changes is referred by Laprie [15] as the *evolvability* property of a system and it is crucial for systems that have to be resilient.

Self-adaptive software is able to provide such desired evolvability, since it is a capable of modifying its own behavior when facing changes in its environment [16]. A system can be closed-adapted, where the predefined adaptive behavior is embedded in the system. In this case the system has a limited number of adaptations and does not allow new behaviors to be introduced at runtime. Systems that permit such runtime flexibility, where new adaptation plans can be added during execution are said to be open-adapted. Taylor [23] refers to runtime software evolution (RSE), as an alternative term to dynamic adaptation, which constitutes the ability of a software system's functionality to be changed during runtime, without requiring a system reload or restart.

The current trend of ubiquitous computing and critical applications with high availability requirements lead to ever changing scenarios where applications need to constantly adapt. Dependability is always necessary in such contexts, but upon eventual adaptations systems must ensure that they continue to be dependable. Therefore, resilience can be seen today as the ultimate objective of dependable applications that take adaptivity into account.

C. Dependencies

Two types of inter-component dependencies have been previously identified [13] for systems that allow loading components at runtime: prerequisites and dynamic dependencies. They would be equivalent to our definitions of static and dynamic dependencies, respectively. In addition, we specify a third level of dependency which we call resource dependency, which is not limited to inter-component dependencies since it may depend on things provided by the environment.

Static dependencies exist when a reconfiguration requires restarting and reinitializing the module, causing its full state to be lost and all its components instances to be destroyed. Because the unit of deployment is a module, and at the module level is where implementation dependencies are handled, the module is clearly the granularity that is directly affected. State-loss and instance destruction are required when a module imports implementation code from another, and the provider module changes. For example, if module A requires classes from module B, and B is updated, we must also update A to use the newer implementation of B. This type of dependency is common for datatypes specified in service specifications and for modules that provide libraries. Implementation dependencies are always mandatory for a module to operate correctly (i.e., they are prerequisites) and are costly because they cause the destruction of dependent modules' component instances (which hold the application's state) when changes are applied.

Dynamic dependencies are those where a reconfiguration is possible without restarting the module and loosing state. These dependencies occur at the service level and benefit directly from the principles of service-oriented computing. Required services may be optional, degrading functionality of client components when not available. Dynamic dependencies affect the component instance and cause rebinding to a compatible service if a change occurs. If no compatible dependency is found and the service is mandatory, then the component instance is stopped, and its provided services removed from the registry until its dependencies can be once again resolved.

Resource dependencies, generally regard configuration, and can be either static or dynamic. For example, a communication port, according to how the component is implemented may be static, and require reinitialization of the module to change, or may be dynamic having the component internally handle the change. Also, a port may not be used by two components simultaenously, so declaring these dependencies helps avoid conflicts at runtime. Other examples include hardware devices and files. In general, these dependencies specify if the resource they require can be shared or not (e.g., a file might be read simultaneously) and if the dependency is static or dynamic. The effects at runtime are the same as for static dependencies if the resource is static and dynamic dependencies if the resource is dynamic.

D. *Dynamic Reconfigurations*

Dynamic reconfigurations may be costly and have a stronger impact than one might consider if only looking at service dependencies, even if using even following the principles of service-oriented computing that ensure loose coupling. When changes to a dependency occur, the impact on dependents can be that of total state-loss for the dependent, in the case of static dependencies, or simply rebinding to another service in the case of dynamic dependencies. Regarding impact, for each dynamic reconfiguration that is applied, we take note of what the impact will be on existing components and modules. There are two types of impact:

Dynamic dependencies affect component instances. If a dynamic dependency exists on a component's service that disappears, the dependent component will require a rebind to find another suitable service (or may wait for the same service in the case of an update). If no suitable service is found, the component is stopped. In either case, the impact is rebind or stop, and it is important it be made explicit. An impacted component may affect other dependent components, causing them to stop or rebind themselves, and so on, impacting others in a domino effect which may bring down the whole application.

Static dependencies affect both modules and components, because components are provided by modules. If an implementation-dependency changes, all component instances of the dependent modules are destroyed causing state-loss. Since this affects dependent modules and components the effects are propagated across the system. All statically dependent modules are destroyed (leading to further state-loss), and components with service-dependencies are rebound or stopped.

III. THE IMPACT OF DYNAMIC RECONFIGURATIONS

When dealing with RSE, the typical units of replacement are components that are interconnected to form an application. Indeed, the possibility of dynamically performing updates on parts of the application while it is still running brings a lot of flexibility. Component-based software development and service-oriented computing offer replaceable building blocks for realizing the goal of runtime software evolution. These approaches can be employed in different techniques for constructing adaptive components and services for constructing flexible and evolvable applications. However, this flexibility comes at a cost since such dynamic reconfigurations have a significant impact in application execution. Different considerations concerning this dynamism have to be taken into account when developing software infrastructure and components targeting an approach where runtime software evolution is possible.

Dynamic updates may be overlooked by others but there is a complex series of events that are involved with such mechanism. Despite different perspectives on component deployment lifecycle (e.g., install, start, install, update) [4][17][22], for the sake of simplicity we utilize a general and temporal perspective on the phases that are present in a lifecycle state transition. These phases consist on stages before, during and after a transition, which we will generally refer to as an update. The possibility of updates performed during application execution introduces a myriad of consequences, which are of different nature and impact for each of those stages, being a potential risk to application dependability. Some of these issues, grouped by the corresponding phase, are briefly discussed next.

Before. As component-based applications are comprised by a set of components with interrelated dependencies, inter-component dependency asks for a verification of the requirements (e.g. required hardware) — also called prerequisites — in order to check if a component can be installed in the runtime [13]. If a component is to be replaced, verification mechanisms should ensure type versioning consistency by not allowing type compatibility to be broken [2]. The fact of adding or removing components during application execution may change (or refresh) the set of interconnected dependencies. Therefore the system is lead to a reconfiguration that can impact other components in the application.

During. An update should not avoid interruptions of on-going operations that would be directly or indirectly related to such update. Some systems disregard such issue while others try to put constraints regarding updates. Maintaining component state is another issue when components are updated and their state needs to be preserved while its behavior is updated to a new version. A transactional update mechanism should ensure restoration of a previous component version in the case of unsuccessful updates, so the system is able to perform a rollback and restore component's behavior and state as it was before the update.

After. The process of a component update can be successful but after it takes place, there may be inconsistencies such as dangling objects left or executing tasks belonging to the component that were not properly terminated. Concerning the inter-component dependencies, the system at this stage must verify the dynamic dependencies among loaded components in a running system [13]. In some dynamic platforms, the fact of loading a component does not mean that it is ready to execute. Other issues are rather related with regular application execution, but may be directly affected after the update of a component that eventually introduces faulty behavior. Fine-grained resource monitoring allows the application to keep monitoring component performance in order to identify which components are consuming resources (e.g., CPU, memory) more than expected. By identifying which component is responsible for that, corrective measures can be directly addressed to it. Besides excessive resource consumption, other errors (e.g. programming errors, non-deterministic faults) may be caused by components updated at runtime. Fault containment mechanisms should prevent errors introduced by one component from being propagated to others. The continuous verification of non-functional attributes conformance can be seen as another issue to be considered after dynamic updates. In SOA they typically take the form of quality of service (QoS) attributes (e.g., performance, availability) represented in a service-level agreement (SLA). If the monitored QoS diverge from expected values the system should perform dynamic optimizations [18][9], which could also include the update or selection of other components or services.

Among many questions that can be raised around this subject, there are two important issues that concern all of these phases of an update: the update cost and when to update.

Update cost: The total cost of a dynamic reconfiguration becomes a function of the impacted modules, the component instances destroyed (causing state loss), the (re)creation of component instances, the number of re-bindings, the modules installed, the modules restarted and the components and modules that do not start because of unsatisfied dependencies. If resource monitoring at the component level is available, we can calculate the total resources used in a dynamic reconfiguration (e.g., memory, CPU, disk) by using the individual resource consumption for each reconfigured element and adding them up. But for critical systems, we need to calculate the cost before the dynamic reconfiguration.

When to update: It is difficult to define the appropriate moment to reconfigure. The update cost information could be combined with statistical information of application usage in order to perform "expensive" updates only at times when the system is not being used by (many) users, minimizing the perceived impact. For instance, Software rejuvenation [12] uses that approach for performing strategic resets on parts of the applications in order to reset application state so inconsistencies can be removed or avoided. As another example, the criteria of quiescence [14] and tranquility [25] are introduced as safe update states where the node (i.e., component) to be updated should not be engaged in transactions fired by the node itself or by nodes that may call it. This sort of safe update state may not be certain in environments where the application provider is not able to control all the components, such as in a service-oriented architecture. In such cases the system must cope with temporary unavailability [24] of services in case of updates.

IV. CONCLUSIONS

Currently, there are platforms that allow the construction of dynamic applications albeit in limited ways, being aware that runtime updates may bring inconsistency to software. Spontaneous system reconfigurations, which can typically take place in ubiquitous and other highly dynamic environments, can be hard to support if the difficulties we have mentioned are not taken into account. Regarding production systems where the risk of failure must be minimal, updates still require a controlled and predictable environment, where testing and validation has been thorough.

Component-based development helps us tackle the complexity of applications and provides us with smaller units of functionality that can be individually replaced. However, even if some component frameworks provide the ability to replace small parts of the application during runtime, in general, highly dynamic applications are still difficult to construct and handle, even in the most advanced frameworks.

One must often be on the defensive when developing components in order to avoid misbehavior especially when many component dependencies are involved. There are also few guarantees that an application can adapt and still correctly execute or that the dependencies will be available for a component to be enabled. Many issues regarding dynamic applications remain. To mention a few, it is hard to know what the impact of an update on an application will be because the side-effects are not known and the information needed is implicit or non-existent. Updates and reconfigurations still lack many features such as being atomic or transactional in order to ensure they are applied entirely or not at all, which is crucial when ensuring an application's correctness. Another issue concerns state transfer, which comes down to how to update a component and have it to continue where the old version left off.

We can write applications that introduce and tolerate dynamic behavior, meaning the applications internal structure can adapt, but the fact is that even with the most advanced technologies available, there are limits. These limits include the collateral impact of an update, which can bring down the whole application, or the fact that we still lack models for handling "sparkling architectures", where components or services disappear intermittently, even frequently, as is the case in pervasive and ubiquitous environments, as well as applications with high availability requirements.

REFERENCES

[1] T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-Based Applications. In Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, pages 32–39. IEEE Computer Society, June 2000

[2] P. Brada and L. Valenta. "Practical Verification of Component Substitutability Using Subtype Relation. In Proceedings of the 32nd EUROMICRO Conference on Softwar"e Engineering and Advanced Applications (EUROMICRO '06). IEEE Computer Society, Washington, DC, USA, 38-45

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma , J.B. Stefani. "The FRACTAL component model and its support in Java", SPE, v.36 n.11-12, p.1257-1284, September 2006

[4] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, A. L. Wolf. "A Characterization Framework for Software Deployment Technologies," Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998

[5] CA Technologies. The Avoidable Cost of Downtime . Research Report, September 2010. http://www.ca.com

[6] CA Technologies. The Avoidable Cost of Downtime . Research Report, November 2010. http://www.ca.com

[7] L. Coyle, M. Hinchey, B. Nuseibeh , and J.L. Fiadeiro. “Guest Editors' Introduction: Evolving Critical Systems”. In Proceedings of IEEE Computer. 2010, 28-33.

[8] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou & K. Pohl. “A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engineering, 15:313-341

[9] V. Grassi, R. Mirandola, and A. Sabetta. “A model-driven approach to performability analysis of dynamically reconfigurable component-based systems”. In Proceedings of the 6th international workshop on Software and performance (WOSP '07). ACM, New York, NY, USA, 103-114.

[10] J. Gray, "Why do computers stop and what can be done about it?" In Symposium on Reliability in Distributed Software and Database Systems, 1986, pp. 3-12.

[11] M. Hinchey, L. Coyle. “Evolving Critical Systems”. Lero Technical Report Lero-TR-2009-00. http://www.lero.ie/sites/default/files/Lero-TR-2009-00-20090727.pdf

[12] N. Kolettis and N.D. Fulton. “Software Rejuvenation: Analysis, Module and Applications”. In Proceedings of the Twenty-Fifth international Symposium on Fault-Tolerant Computing (June 27 - 30, 1995). FTCS. IEEE Computer Society, Washington, DC, 381.

[13] F. Kon and R. H. Campbell. “Dependence Management in Component-Based Distributed Systems”. IEEE Concurrency 8, 1, 26--36 (2000)

[14] J. Kramer and J. Magee. “The Evolving Philosophers Problem:Dynamic Change Management,” IEEE Trans. Software Eng., vol. 16, no. 11, pp. 1293-1306, Nov. 1990.

[15] J.C. Laprie. “From dependability to resilience”. In 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks, 2008

[16] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf. “An Architecture-Based Approach to Self-Adaptive Software”. IEEE Intelligent Systems 14, 3 (May 1999), 54-62

[17] OSGi Alliance. OSGi Service Platform. http://www.osgi.org

[18] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann. “Service-oriented Computing: State of the art and research challenges”. IEEE Computer, 11, 2007.

[19] F. Plasil, D. Balek, R. Janecek: SOFA/DCUP: architecture for component trading and dynamic updating. In: 4th Intl. Conf. on Configurable Distributed Systems, pp.43--51 (1998)

[20] M.J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A.L. Wolf. “Reconfiguration in the Enterprise JavaBean Component Model” In Proceedings of the IFIP/ACM Working Conference on Component Deployment, Berlin, 2002, pp. 67-81

[21] C. Szyperski. “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley Longman Publishing Co., Inc., 2002

[22] C. Szyperski. “Component technology: what, where, and how?”. In Proceedings of the 25th International Conference on Software Engineering (ICSE '03). IEEE Computer Society, Washington, DC, USA, 684-693.

[23] R. N. Taylor, N. Medvidovic, P. Oreizy. “Architectural styles for runtime software adaptation”. In 3rd European Conference on Software Architecture (ECSA) (September 2009), pp. 171-180

[24] L. Touseau, D. Donsez, and W. Rudametkin. “Towards a SLA-based Approach to Handle Service Disruptions”. In Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 1 (SCC '08), Vol. 1. IEEE Computer Society, Washington, DC, USA, 415-422.

[25] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”. IEEE Trans. Softw. Eng. 33, 12 (December 2007), 856-868

[26] R. Van Ommering, et al.: “The Koala component model for consumer electronics software,” Computer, vol. 33, no. 3, 2000, pp. 78-85.

[27] M. Weiser. The computer for the 21st century. Scientific American (September 1991).