Architectural Stability and Middleware: An Architecture-Centric Evolution Perspective

Rami Bahsoon[1] and Wolfgang Emmerich[2]

[1]School of Engineering and Applied Science, Computer Sc, Aston University Birmingham, UK, {r.bahsoon@aston.ac.uk}

[2]London Software Systems, Dept. of Computer Science, University College London, UK, {w.emmerich@cs.ucl.ac.uk}

Abstract. *Architecture stability refers to the extent to which a software system can endure changes in requirements, while leaving the architecture of the software system intact. We argue that changes in non-functional requirements are critical to threat the stability of a software architecture over its projected lifetime. We claim that focusing the analysis on the "coupling" of middleware and software architectures is a step towards understanding the ramifications of the change in the so called middleware-induced architectures. Middleware-induced architectures follow an architecture-centric evolution approach, as the emphasis is placed on the induced architecture and the provided middleware primitives to simplify the construction of distributed systems, realize many of the non-functional requirements (e.g., scalability, fault tolerance, etc.) and facilitate their evolution over time. To support the claim, we use a case study and we observe how a software architecture, when induced by distinct middleware, differs in coping with changes in non-functional requirements. We conclude by hinting on future research directions in the area.*

1. Architectural Stability and Middleware

Software requirements, whether functional or non-functional, are generally *volatile*; they are *likely* to change and evolve over time. The change is inevitable as it reflects changes in stakeholders' needs and the environment in which the software system works. The change may "break" the software system architecture necessitating changes to the architectural structure (e.g., changes to components and interfaces), architectural topology (e.g., architectural style), or even changes to the underlying architectural infrastructure (e.g., middleware). It may be expensive and difficult to change the architecture as requirements evolve [10]. Consequently, failing to accommodate the change leads ultimately to the degradation of the usefulness of the system. Hence, there is a pressing need for flexible software architectures that tend to be *stable* as the requirements evolve. By a stable architecture, we refer to the extent to which a software system can endure changes in requirements, while leaving the architecture of the software system intact. We refer to the presence of this "intuitive" phenomenon as architectural stability [1].

Ongoing research on relating requirements to software architectures has considered the architectural stability problem as an open research challenge and *difficult* to handle [7, 10, 13, 18]. van Lamsweerde [18] acknowledges that: "the conflict between requirements volatility and architectural stability is a difficult one to handle". Nuseibeh [13] noted that many architectural stability related questions are difficult and remain unanswered. Examples include: what software architectures (or architectural styles) are stable in the presence of changing requirements, and how do we select them? What kinds of changes are systems likely to experience in their lifetime, and how do we manage requirements and architectures (and their development processes) in order to manage the impact of these changes? In [2, 7], we reflected on the architectural stability problem with a particular focus on developing distributed software architectures induced by middleware. Specifically, we considered the architecture stability problem from the distributed components technology in the face of changes in *non-functional requirements*. We argued that addition or changes in functional requirements could be easily addressed in distributed component-based architectures by adding or upgrading the components in the business logic. However, changes in non-functional requirements are more critical; they can stress an architecture considerably, leading to architectural "breakdown". Such a "breakdown" often occurs at the middleware level and is due to the incapability of the middleware to cope with the change in non-functional requirements (e.g., increased load demands). This may drive the architect/developer to consider ad-hoc or propriety solutions to realize the change, such as modifying the middleware, extending the middleware primitives, implementing additional interfaces, etc. Such solutions could be problematic, costly, and unacceptable.

We argue that changes in non-functional requirements are critical to threat the stability of the software architecture over its projected lifetime. We claim that "coupling" middleware with software architectures is a step towards understanding the ramifications of the change in distributed systems that are built on top of middleware. To support the claim, we observe how a software architecture, when induced by distinct middleware, differs in coping with changes in non-functional requirements. We hint on "tactics" that requirements engineering must consider to proactively engineer stable architectures to facilitate evolution of the system and its environment.

2. Middleware-Induced Architectures

The requirements that drive the decision towards building a distributed system architecture are usually of a non-functional and global nature. Scalability, openness, heterogeneity, and fault-tolerance are just examples. The

current trend is to build distributed systems architectures with middleware technologies such as Java 2 Enterprise Edition (J2EE) [17] and the Common Object Request Broker Architecture (CORBA) [14]. Middleware-induced architectures follow an *architecture-centric* approach to evolution, as the emphasis is placed on the induced architecture for simplifying the construction of distributed systems by providing high-level primitives, which shield the application engineers from the distribution complexities, managing systems resources, and implementing low-level details, such as concurrency control, transaction management, and network communication. These primitives are often responsible for realizing many of the non-functional requirements (e.g., scalability, fault tolerance, etc.) in the architecture of the system induced and facilitating their evolution over time. Despite the fact that architectures and middleware address different phases of software development, the usage of middleware can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware [6]. Once a particular middleware system has been chosen for a software architecture, it is extremely expensive to revert that choice and adopt a different middleware or a different architecture. The choice is influenced by the *non-functional requirements*. Unfortunately, the requirements tend to be unstable and evolve over time and threat the stability of the architecture. Non-functional requirements often change with the setting in which the system is embedded, for example when new hardware or operating system platforms are added as a result of a merger, or when scalability requirements increase as a result of having to build web-based interfaces that customers use directly [8]. Hence, as the non-functional requirements of the software system evolve, "coupling" middleware and architectures becomes the focal point for understanding the *stability* of the distributed software system architecture in the face of the change.

3. Case Study: The Evidence

We observe how a software architecture, when induced by distinct middleware, differs in coping with changes in non-functional requirements. The experiment is fully documented in [2]. We use the Duke's Bank application, an online banking application provided by Sun [17], as part of the J2EE reference application. Given the software architecture of the Duke's Bank, we have instantiated from the core architecture two versions, each induced by a different middleware: one with CORBA and the other with J2EE. We have observed how a likely future change in scalability could impact the architectural structure of each version. Scalability denotes the ability to accommodate a growing future load, be it expected or not. We look at the changes in scalability demands as a representative of a critical change in non-functional requirements that could impact the architecture at its various levels: structure, topology, and infrastructure. The ability to scale the software system of a given architecture is revealing to its stability, for the change may break the architecture and/or ripple to impact other non-functionalities such as fault-tolerance, performance, reliability, availability, when poorly accommodated by the system. Further, the challenge of building a scalable system is to support changes in the allocation of components to hosts without breaking the architecture of the software system, or changing the design and code of a component [8]. We note that the stability notion is relative to the change. Hence, what we observe is how the architecture of the given system, when induced by a particular middleware cope with the scalability change.

Though the experiment is conducted in a controlled environment, we regard the Duke's bank application to be adequately representative of medium-size component-based distributed application. The architecture of the Duke's Bank application is a 3-tier style, given in Figure 1. The architecture has two clients: an application client used by administrators to manage customers and accounts, and a Web client used by customers to access account statements and perform transactions. The server-side components perform the business methods: these include managing customers, accounts, and transactions. The clients access the customer, account, and transaction information maintained in a database.

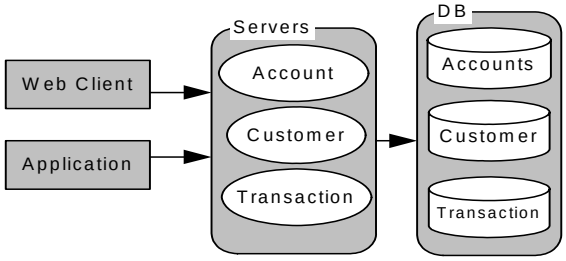

Figure 1. The Architecture of the Duke's Bank

We have assumed that the Duke's Bank system needs to scale to accommodate the growing number of clients in one-year time. We have considered scalability as a goal that needs to be achieved by the architecture of the software system. We have adopted a goal-oriented approach to refining requirements (e.g., [5]). We have refined the goal, using guidance on how it could be operationalised[5] by the architecture, when induced by a particular middleware. In more abstract terms, the guidance was given through the knowledge of the domain; vendor's specification, and previous design and implementation experience. We note that different architectural mechanisms may operationalise the scalability goal. As an operationalisation alternative, we use replication as way for achieving scalability. The reason is due to the fact that both CORBA and J2EE do provide the primitives or guidelines for scaling a software system using replication, which make the comparison between the two versions feasible. In particular, the Object Management Group's CORBA specification defines a fault tolerance and a load balancing support, both when combined provide the core capability for implementing scalability through

replication. Similarly, J2EE provides the primitives for scaling the software system through replication. We have then estimated the structural impact upon achieving the scalability goal on both the CORBA and the J2EE versions. We have estimated the SLOC to be added for implementing the change on both versions. We have calculated the expected relative savings in maintenance including development, deployment, and configuration efforts. We have applied the ArchOptions model [1,2,3,4] to quantify the flexibility of the architecture of each version relative to the change. An observable advantage of scaling the software architecture when induced by EJB is that no development effort is required to realize the scalability requirements through replication, as when compared to the CORBA version. J2EE does provide the primitives for scaling the software system, which result in making the architecture of the software system more *flexible* in accommodating the change in scalability, as when compared to the CORBA version. Our claim that middleware induced software architecture differs in coping with changes is verified to be true for the given change.

Evidence 1. *Understanding architectural stability has to be done in connection with the solution domain.*

For the category of distributed software systems that are built on top of middleware, the observations affirm the belief that investigating the stability of the distributed software architecture could be fruitless, if done in isolation of the middleware. This is because the middleware constraints and dominates much of the solution that relate to the non-functionalities, and consequently constraint how smoothly the non-functionalities may evolve over the projected lifetime of the software system. Hence, the development and the analysis for architectural stability should consider the "coupling" between the architecture and the middleware. This addresses pragmatic needs and is feasible even at earlier stages of the software development life cycle, where a considerable part of the distributed system implementation could be available, when the architecture is defined during the Elaboration phase of the Unified Process. We also note that the change in requirements could have been addressed by other architectural mechanisms. However, the middleware has guided the solution for evolving the software system. For instance, the choice of replication as an architectural mechanism for scaling the software system, was guided by the clustering primitives provided by J2EE and the core capabilities provided by CORBA to support load balancing and fault tolerance. Interestingly, [6] state that "despite the fact that architectures and middleware address different phases of software development, the usage of middleware and predefined components can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware used in the implementation phase". In abstract terms, Rapanotti et al. [15] advocate the use of information in the solution domain (e.g., the middleware-to be induced for our case) to inform the problem space:

> "Whereas Problem Frames are used only in the problem space, we observe that each of these competing views uses knowledge of the solution space: the first through the software engineer's domain knowledge; the second through choice of domain-specific architectures, architectural styles, development patterns, *etc*; the third through the reuse of past development experience. All solution space knowledge can and should be used to inform the problem analysis for new software developments within that domain" [15].

Evidence 2. *Understanding Architectural Stability: Intertwined with changes in non-functional requirements, style, and the middleware*

Following the definition of [6], a *style* defines a set of general rules that describe or constrain the structure of architectures and the way their components interact. Styles are a mechanism for categorizing architectures and for defining their common characteristics. Though both versions have exhibited similar styles (i.e., three-tier), they have differed in the way they cope with the change in scalability. The difference was not due to the architectural style, but due to the primitives that are built in the middleware to facilitate scaling the software system. The governing factor, hence, appears to be to a large extent dependent on the flexibility of the middleware (e.g., through its built-in primitives) in supporting the change. The intuition and the preliminary observations, therefore, suggest that the style by itself may not be revealing for the analysis of architectural stability when the non-functional requirements evolve. It is, however, a factor of the extent to which the middleware primitives can support the change in non-functional requirements. Interestingly, [16] claims that for a system to be implemented in a straightforward manner on top of a middleware, the corresponding architecture has to be compliant with the architectural constraints imposed by the middleware. [16] supports this claim by demonstrating that a style, that in principle seems to be easily implementable using the COM middleware, is actually incompatible with it. Following a similar argument, adopting an architectural style that is in principle appear to be suitable for realizing the non-functionality and supporting its evolution, may not be complaint with the middleware in the first place. And if the architectural style happens to be compliant with the middleware, there are still uncertainties in the ability of the middleware primitives to support the change. In fact, the middleware primitives realize much of the non-functional requirements. Hence, the architectural style by itself may not be revealing for potential threats that the architecture may face when the non-functional requirements evolve. The evolution of non-functionality maybe in principle easily supported by the style, but could be uneasily accommodated by the middleware. An observable advantage of scaling the software architecture induced by J2EE, for example, is that no development effort required to realize the scalability requirements through replication, as when compared to that of CORBA, knowing that in principle the style of both versions exhibit similar capabilities.

4. Discussion and Conclusion

The evidence is appealing to employ the "coupling" between middleware and software architectures in developing, analyzing, and facilitating evolution. This is because the solution domain can guide the development and evolution of the software system; provide more pragmatic and deterministic knowledge on the potential success (failure) of evolution; and consequently assist in practically understanding the stability of the software architectures in the face of the change.

Identifying and documenting possible future changes is important in order to manage software evolution [11] and evaluate architectural choices [12]. Managing the change is a process which involves recognizing the change through continued requirements elicitation, requirements evaluation of risk, and evaluation of systems in their operational environments [12]. Eliciting and dealing with the change in requirements, however, is still one of the major research challenges facing the requirements engineering community [9]. Engineering requirements for evolution, we advocate adjusting requirements elicitation and management techniques to elicit not just the current non-functional requirements, but also to assess the way in which they will develop over the projected lifetime of the architecture. These ranges of requirements may then inform the selection of candidate distributed components technology, and subsequently the selection of application server products. Further, requirements engineering has not only to be aware of the architecture (e.g., the style), but also of the underlying middleware. For example, if we take a goal-oriented approach to requirements engineering (e.g., [5]), we advocate adjusting the non-functional requirements elicitation and their corresponding refinements to be aware of both the architectural style and the constraints imposed by middleware. Hence, the operationalisation of these requirements in the software architecture have to be guided by both the architectural style, the complaint middleware for the said style, and guided by previous experience. Architecture representations (e.g., ADLs), in turn, may need to be populated by middleware-related information (e.g., metadata) which could facilitate analysis and traceability from requirements. This, we believe, is a pragmatic need towards managing the change and focusing the analysis for change on high level of abstraction and hence developing "evolvable" software in face of changes in non-functional requirements.

Despite the fact that ongoing research on the "coupling" of middleware and architectures could have an impact on understanding the relation between architectures and non-functional requirements, their contributions to such understanding is still insufficient. For example, there is minimal research effort on understanding the evolution of non-functional requirements in relation to both the architecture and the middleware when coupled, with our work being the notable exception [2]. Though the reported observations reveal a trend that agrees with the intuition, research, and the state-of-practice, confirming the validity of the observations is still subject to careful further empirical studies. These studies may need to consider other non-functional requirements, their concurrent evolution, and their corresponding change impact on different architectural styles and middleware, which we are currently investigating as part of our ongoing research agenda.

5. References

[1] Bahsoon, R.: Evaluating Architectural Stability with Real Options. PhD Thesis, University of London(2005)

[2] Bahsoon, R., Emmerich, W., and Macke, J.: Using ArchOptions to Select Stable Middleware-Induced Architectures. In: IEE Proceedings Software, Special issue on Relating Requirements to Architectures, IEE Press 152(4) (2005) 176-186

[3] Bahsoon, R., Emmerich, W.: ArchOptions: A Real Options-Based Model for Predicting the Stability of Software Architecture. In: Proceedings of the Fifth ICSE Workshop on Economics-Driven Software Engineering Research (2003)

[4] Bahsoon, R., Emmerich, W.: Evaluating Architectural Stability with Real Options Theory. In: Proc. of the 20th IEEE Int. Conference on Software Maintenance, Chicago, Illinois, IEEE CS Press (2004)

[5] Dardenne, A., van Lamsweerde A., and Fickas, S.: Goal-Directed Requirements Acquisition, Science of Computer Programming, 20, pp. 3-50 (1993)

[6] Di Nitto, E. and Rosenblum, D.: Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In: Proc. of the 21st Int'l Conf. on Software Engineering, (1999) 13-22

[7] Emmerich, W.: Distributed Component Technologies and their Software Engineering Implications. In: Proc. of the 24th Int. Conference on Software Engineering, Orlando, Florida (2002) 537-546

[8] Emmerich, W.: Software Engineering and Middleware: A Road Map. In: A. Finkelstein, ed., Future of Software Engineering. ACM Press (2000)

[9] Finkelstein, A., and Kramer, J.: Future of Software Engineering. In: A. Finkelstein (ed.): The Future of Software Engineering, ACM Press (2000) 5-21

[10] Finkelstein, A.: Architectural Stability. http://www.cs.ucl.ac.uk/staff/a.finkelstein/talks.html (2000)

[11] Lehman, M.M.: The Future of Software – Managing Evolution. IEEE Software (Jan. 1998)

[12] Nuseibeh, B., and Easterbrook, S.: Requirements Engineering: A Roadmap. In: A. Finkelstein (ed.): The Future of Software Engineering, ACM Press (2000) 35-46

[13] Nuseibeh, B.: Weaving the Software Development Process Between Requirements and Architectures. In: Proc. of the 1st. International Workshop From Software Requirements to Architectures (2001)

[14] Object Management Group: The Common Object Request Broker: Architecture and Specification, 2.4 ed., OMG, Needham, Mass. (2000)

[15] Rapanotti, L., Hall, J., Jackson, M., and Nuseibeh, B.: Architecture Driven Problem Decomposition. In: Proc. of 12th IEEE International Requirements Engineering Conference (RE'04), Kyoto, Japan (2004)

[16] Sullivan, K. J., Socha, J., and Marchukov, M.: Using Formal Methods to Reason about Architectural Standards. In: Proc. of the 19th International Conference on Software Engineering, Boston, MA (1997)

[17] Sun MicroSystems Inc: Enterprise JavaBeans Specification v2.1 (June 2002)

[18] van Lamsweerde, A.: Requirements Engineering in the Year 00: A Research perspective. In: Proc. 22nd International Conference on Software Engineering, Limerick, Ireland (2000)