Hints to Address Concurrency in Self-Managed Systems at the Architectural Level

A Case Study

Francesco Mazzei and Claudia Raibulet
Università degli Studi di Milano-Bicocca
DISCo - Dipartimento di Informatica, Sistemistica e Comunicazione
Milan, Italy
e-mail: f.mazzei3@campus.unimib.it, raibulet@disco.unimib.it

***Abstract*—Self-managed systems are able to perform changes by themselves on themselves during their execution due to variations occurred internally or in their execution environment. Current solutions for self-managed systems address one change at a time. In the real world, two or more changes may raise contemporaneously. Hence, they should be addressed concurrently and not sequentially. This would improve the overall performances of a system and would avoid delays in addressing changes. In this paper, we propose architecture level mechanisms to address concurrency in self-managed systems. We investigate available concurrency solutions used in non self-managed systems and adopt and adapt them for self-managed systems. We also introduce novel concepts such as: adaptation zone, adaptation need, and adaptation level. To apply and validate our solution a case study in a Web banking context has been developed.**

Keywords-self-adaptivity; concurrency; architecture.

I. INTRODUCTION

Over the years, systems have grown in size and complexity and many of them are required to run continuously. Therefore, it has become important to develop systems that are able to manage and adapt themselves at runtime in response to changing requirements and environmental conditions. Examples of situations in which self-managed systems may offer a valid solution are identified in [12]: (1) systems should run continuously in the presence of components' faults, variability in resources, or variability of users' needs, (2) administrative overheads should be reduced allowing smooth operation with minimal human oversight, and (3) systems should provide various levels of services to different users depending on their needs and context.

Current solutions consider that systems address one change at a time. In the real world, two or more changes may occur contemporaneously. Several issues may raise here: is it possible to address them concurrently? If yes, under which conditions (considering that modifications are performed at runtime and systems should provide the functionalities for which they have been designed independent of the computation performed for their self-management issues). If no, how to establish their priority for changes' execution? Hence, how to address two or more changes at a time in a self-managing system?

Engineering self-managed systems is not a trivial task [3][5][9]. Addressing one change at a time is complex. However, it is easier to reason about one change at a time: no synchronization or consistency issues should be considered. Addressing two or more changes at a time becomes a significant issue. Each change should leave the self-managed system in a stable state. A question may raise here: is it worth the effort considering the possible benefits of addressing concurrent issues? David Garlan mentions two possible benefits: (1) improvement of the performances of the self-managed systems through the parallelization of the changes, and (2) provision of an immediate feedback when change needs raise. For example, if a self-optimization task is running in the system, and a self-healing issue occurs in the meantime, it is desirable that the last is addressed immediately independent of the fact that the first continues its execution or should be stopped or finished immediately.

As far as concerns our knowledge, there is no available solution for engineering self-managed systems that implements concurrency mechanisms to manage contemporaneously two or more changes. The closest solutions take into consideration multiple objectives when deciding which change to perform in the system [2]. However, the multiple objectives are summarized into a single change.

The objective of this paper is to investigate the available concurrency mechanisms which have been defined for traditional systems and which can be adopted and/or adapted for self-adaptive systems. Examples of mechanisms for addressing concurrency issues include prioritization, scheduling, architectural and design patterns. This work aims also to validate the identified solutions through a common and actual case study.

The rest of the paper is organized as follows. Section II presents the main concepts used in our solution. Section III presents a case study using the previously introduced concepts. The paper ends with the Conclusions and Future Developments presented in Section IV.

II. MAIN CONCEPTS OF OUR SOLUTION

Prof. David Garlan mentioned concurrency [7] as one of the future challenges of self-managed systems at SEAMS 2013. The potential benefits of exploiting concurrency concern performance and rapid response when new self-managing issues arise. He also suggested three ideas on how to manage concurrency in such systems:

 ISBN: 978-1-61208-463-3

- non-interference guarantee between concurrent adaptations;
- possible interruptions of ongoing adaptations, when higher-priority adaptations occur;
- possible finish of unproductive adaptations.

Starting from these course-grained ideas, we try to expand them and to provide fine-course possible hints.

In the remaining of this section, we introduce the main concepts on which our solution is based: Monitor-Analyze-Plan-Execute (MAPE) loop, adaptation zone, adaptation process, adaptation need, and adaptation level.

A. The MAPE Loop

Architectural-based solutions for self-managed systems exploit feedback concepts and control mechanisms [3][5]. The feedback enables a system to understand its current state and its execution environment by monitoring itself and its surrounding world. To achieve this the system collects meaningful data through various sensors and/or mechanisms, data which is further analyzed by the system. The control enables a system to be active and perform changes on itself in correspondence of variations in its execution environment. To achieve this a system chooses the most appropriate changes to be performed in its current state and implements mechanisms to apply the identified changes.

The feedback control loops consist in the following four steps: Monitoring, Analyzing, Planning, and Executing (MAPE) [5].

B. Adaptation Zone

An *adaptation zone* indicates a part of a system which may be subject to changes at runtime. In other words, an adaptation zone indicates the co-related elements of a system which may be involved in a type of adaptation at runtime. A zone is a mutable, a dynamic part of a system.

At the architectural level, an *architectural adaptation zone* may be composed of components, packages, classes, interfaces. Each architectural adaptation zone is identified using a name that reminds the type of adaptation in that zone. It is possible to use, for example, the name of each adaptation use case for the corresponding zones.

From the concurrency point of view, if two or more adaptations occur in disjoint architectural adaptation zones they may be executed in parallel without any further concurrency issue. Otherwise, if two or more adaptations occur in the same architectural adaptation zone, concurrent issues, such as shared resources problem, should be considered and a mutual exclusion solution is needed. To address this issue we define the *runtime adaptation zone*. At runtime, many instances of a system's element may be created (e.g., objects). An adaptation may use only part of the instances available in an architectural adaptation zone. Hence, we introduce the concept of runtime adaptation zone to group together the instances of the system's elements of an adaptation zone used actually in an adaptation.

A runtime adaptation zone may be in one of the following states:

- green, meaning that in the runtime adaptation zone every object is unlocked, or rather no adaptations are running in the zone; when an adaptation finds the green color, it will run and use objects without any constraint;
- yellow, meaning that in the runtime adaptation zone one or more objects are in use by other adaptation(s); an adaptation may run in this zone only if it uses objects that are not locked by the other adaptation(s);
- red, meaning that in the runtime adaptation zone every object is locked, or rather one or more adaptations are running in this zone using all objects; when an adaptation finds the red color, it will wait until the objects it plans to use will be unlocked.

A runtime adaptation zone may modify its state going in one of the remaining two states without following any particular sequence.

C. Adaptation Process

An adaptation consists in a change in a system. It is the result of a feedback control process composed of four main steps: monitoring, analyzing, planning, and executing. Each of these steps may be complex and may require several entities for its implementation. In our solution, we have defined four managers, each supervising one step of the adaptation process. The Adaptation Monitor gathers the data describing the current state of the execution environment of a system (both the system itself and its external world). The Adaptation Analyzer verifies the current state of a system and identifies the variations which may require a change in a system. These two entities must run for the entire lifetime of the system and they continuously or periodically check the state of a system.

From the concurrency point of view, there are no particular issues in these two steps. The monitoring step may gather concurrently data from various sources regarding various aspects. The analyzing step may verify concurrently various data to reveal variations.

The planning step is managed by the Adaptation Planner. The planning step identifies the change to be performed and the strategy to apply the identified change in a system. A strategy is composed of a set of operations required to adapt a system to a need starting from its current state. The Adaptation Planner is the manager of the adaptation strategies. To develop interchangeable strategies, the Strategy design pattern is used.

From the concurrency point of view, the Adaptation Planner may manage in parallel two or more adaptations each considered separately. It works similarly to the Adaptation Monitor and Adaptation Analyzer with the difference that it is activated only when an adaptation is needed in a system.

When the Adaptation Planner has decided which is the most appropriate strategy to be applied for the current adaptation, it has to be performed. This is the most

 ISBN: 978-1-61208-463-3

important part of the concurrent adaptation mechanism considered in this paper. In self-managed systems in which there is no concurrency, the adaptation is performed without any particular issues and complexity. But, in this case, more than one adaptation may be needed at the same time. Therefore, this part of the solution must have specific attributes that characterize every adaptation in order to compare them. Hence, we have introduced an entity that represents the execution of an adaptation: the Adaptation Performer.

The Adaptation Performer is the entity that has the responsibility to apply the strategy selected by the Adaptation Planner. It is the element that applies in a system those operations that compose the strategy (see Figure 1).

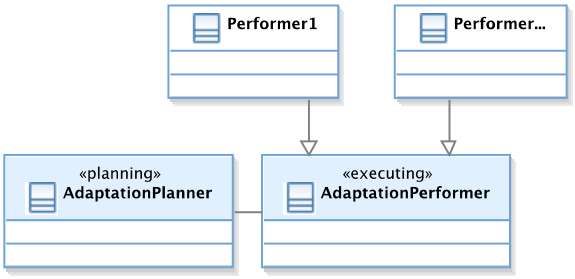

Figure 1. The Adaptation Performer and the Adaptation Planner

From the concurrency point of view, it has been asserted that each adaptation involves only a specific architectural zone, hence one of the Adaptation Performer's attribute is the architectural adaptation zone in which the adaptation has to be applied (see Figure 2). Two or more Adaptation Performers can adjust a system concurrently if they involve different architectural adaptation zones. Furthermore, due to the runtime adaptation zones, if performers involve the same architectural zone, but not the same set of objects, they can run concurrently. However, when they need the same zones and the same objects, they must be compared among them, to determine which should be run, interrupted, or stopped. An entity like a scheduler can do the comparison. To achieve this goal, it is necessary to understand the characteristics of each adaptation and which of those it is useful for the comparison. It has been defined a particular attribute that groups some important characteristics, and it is called Adaptation Need (see Section II.D).

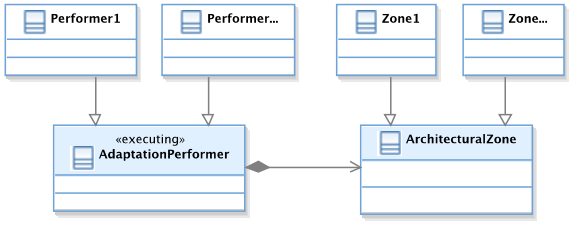

Figure 2. Adaptation Performer with the Architectural Adaptation Zones

D. *Adaptation Need*

An adaptation may be of various types: self-configuration, self-optimization, self-protection, self-healing. and so on. Each of these types has a different importance for a system. Generally, a self-healing or a self-protection adaptation have the greater importance. Obviously, this importance depends strongly on the application domain. Based on these affirmations, the first attribute that characterizes an adaptation is its *type*. Based on the *type* attribute it is possible to define a hierarchy of priorities for adaptations for each system.

With the *type* attribute a first comparison is done among Adaptation Performers. However, if two or more performers have the same type, other attributes are needed to prioritize them. A second attribute consists in the adaptation *Strategy* which has been chosen by the Adaptation Planner. A strategy has associated a static priority (e.g., as the priority of the create, update, insert, delete operations in a database). Further, a strategy spends an estimated time to perform its operations, so it has a considerable importance for the comparison. Thus, *Time* is the third and last element that characterizes an adaptation need. It estimates how much time a strategy needs to be completely performed (see Figure 3). Based on the application domain, the highest priority may be assigned to the strategy having the minor estimated time, or the major estimated time.

To summarize, two steps are performed to compare two or more Adaptation Performers:

- step 1: use type to compare two or more Adaptation Performers;
- step 2: if type is identical, use strategy and time to compare two or more Adaptation Performers.

Due to this two steps comparison, it is possible to decide which process has to be run, interrupted, or stopped.

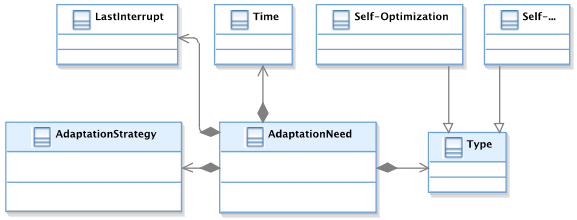

Figure 3. The Adaptation Need with Type, Strategy and Time

If a performer has to be interrupted, a mechanism to interrupt and resume it, is needed. A scheduler can do the comparison between the performers that will run in the same architectural adaptation zone with the same objects and it can choose if a performer has to be interrupted and then resume it. It is possible that the scheduler interrupts more than one Adaptation Performers, so it must have a queue of interrupted processes. To allow the scheduler to choose which performer has to be resumed, the Adaptation Need requires another attribute, called *last interrupt*.

 ISBN: 978-1-61208-463-3

Last interrupt (see Figure 3) defines the last time that a performer has been interrupted (e.g., the timestamp). Every time the scheduler has to choose which Adaptation Performer has to be resumed, it sorts the queue of performers by *last interrupt* and chooses the performer that has the oldest timestamp.

However, in an application there could be more aspects to take into account to resume a performer. Therefore, for each application the last interrupt can be used in combination with other application's elements.

E. Adaptation Level

If one or more processes have to be interrupted or stopped, a stability problem may occur in the system. An adaptation cannot be interrupted or stopped anytime, but it must do this leaving a system in a stable state.

The Adaptation Performer adapts the system performing the Adaptation Strategy's operations. To guarantee the stability, any operation can become an atomic step, in order to allow the interruption (or the stop) after the ending of an operation and before the beginning of another. Clearly, if some sub-steps compose an operation, they must be all executed without stopping. In Stitch [1] language, strategies are composed by tactics. For our solution, we consider that those atomic operations, which compose an Adaptation Strategy, are called tactics.

Furthermore, it is also possible to estimate how much a system has been adapted, according to the number of tactics performed. This is an adaptation value, and it was named *Adaptation Level*.

For example, if three tactics compose the Adaptation Performer strategies, the Adaptation Level can be:

- Low-level: the performer has completed only one tactic;
- Medium-level: the performer has completed two tactics;
- High-Level: the performer has completed three tactics.

This is an easy example, but the Adaptation Level can be usually divided into the three levels. To estimate at which level a performer has adapted a system, it is possible to calculate with percentage notation the operations performed (e.g., 34%, 67%, 100%) or any other solution that the system's engineer considers appropriate. The Adaptation Level is useful to provide information about a system and a system's adaptation. Furthermore, a system may take information about its adaptation, to dynamically add, modify or remove an Adaptation Strategy and modify the estimated Time of an adaptation.

The objects in each zone are in a stable state if there are no Adaptation Performers running tactics over them. Hence, there are three definitions of stable states, object for the objects, local for the zone, and global for the entire system.

- Object stable state: an object is in a stable state if it is not used by an Adaptation Performer;
- Local stable state: a zone is in a stable state if no object is used by an Adaptation Performer (the runtime zone is in a green state);
- Global stable state: the system is in a stable state if all the runtime zones are in a green state.

Therefore, an Adaptation Performer can be interrupted between a tactic and another, when it leaves all the involved objects in a stable state. It is important to understand that each performer has not the responsibility of the entire system's stability (or the zones' stability), but it only has to care about the objects that it involves.

So with all of these definitions, an Adaptation Performer can perform the strategy's operations and it can be interrupted or stopped when necessary.

III. THE UNIBANK CASE STUDY

The solution presented in the previous section has been validated through a case study, a home banking system, called Unibank. The solution implements two possible adaptation types: an architectural one (e.g., addition/removal of servers) and a content one (e.g., visualization of textual and/or multimedia content). The current version of the solution enables (1) the concurrent execution of two or more adaptation processes if they involve different architectural zones, (2) the interruption of an adaptation process if another one arise in the same adaptation zone with a higher priority, or (3) the ending of an adaptation process under certain conditions.

Unibank is a home banking system that provides several services to its customers (e.g., register, create a bank account, visualize the history of the operations performed, do a transfer, require a credit card). In such a system, there are various aspects which can be properly addressed through self-managed solutions. For example, the variations in the system may concern the number of clients' requests, the number of replicated servers the system uses, the available bandwidth, the type of the clients' devices used to communicate with the system.

In the remaining of this section, we present two types of adaptations in which concurrency based concepts are efficiently exploited. Both types of adaptations are triggered by the variations of the number of clients' request. To ensure a constant level of quality, the system may decide to add/remove a replicated server, and/or, vary the quality of the content visualization (e.g., visualize multimedia information and/or only textual information). Further, we introduce briefly the architectural aspects of our solution.

A. Variation of the Number of Servers

The number of clients' requests continuously change, from a low level to a high level. It means that if the number of requests is high, the system needs a greater number of replicated servers to handle them. Vice versa, if the number of requests is low, the system needs a lower number of servers, to reduce the services' costs. It is also possible that one or more servers crash, so other servers are needed to handle the current number of requests.

Every home banking system must guarantee a 365/24 service, regardless of the requests' level and servers status. However, every system has a cost and a budget limit, so

 ISBN: 978-1-61208-463-3

they have also a number of running servers limit. In fact, there is a limited server pool from which to add or remove a specific server.

Hence, in an architectural server-based adaptation, there are two strategies: add one or more servers, and remove one or more servers. This type of adaptation and these two strategies are very similar in their execution. In the following, one of them is described step-by-step as an example:

1) Adding servers according to clients' requests

1. The Adaptation Monitor supervises the number of the clients' requests.
2. The Adaptation Analyzer reveals an increase of the number of clients' requests.
3. The Adaptation Planner chooses a strategy: add a specific number of servers.
4. In the execution step, the Adaptation Performer adds every server needed (if the pool has available the requested number of servers).

B. Variation of the Quality of Content Visualization

For many reasons, the architectural server-based adaptation is not always available (e.g., all the available servers are functioning). Therefore, if the number of clients' requests increases and the adaptation process cannot add replicated servers, it may vary the quality of the content visualization to allow the handling of every request. Vice versa, if the number of requests decreases the adaptation process can improve the quality of the content visualization.

Hence, in a variation of quality of content visualization adaptation there are two strategies: improve the quality of the content visualization, and reduce the quality of the content visualization.

This type of adaptation and these two strategies are very similar in their execution. In the following, one of them is described step-by-step as an example:

1) Improve the quality of contents visualization according to the number of clients' requests

1. The number of client's requests has decreased;
2. When the Adaptation Process starts in the monitoring step, it reveals the number of requests;
3. The process passes to the analyzing step and it now knows that the actual number of servers and quality of contents visualization are too much to handle requests;
4. In the planning step, the Adaptation Process chooses a strategy: improve the quality of contents visualization;
5. In the execution step, the process puts the quality of contents visualization in a higher level.

C. Architectural Aspects

The architecture of the Unibank system is presented in Figure 4. The two grey elements indicate two different architectural adaptation zones. Currently, these architectural adaptation zones are defined statically at design time.

The Adaptation Monitor and the Adaptation Analyzer have been implemented as Singletons because they are quite simple in this case study: they monitor and analyze the number of the clients' requests.

Once that the data is checked and an adaptation is required, the Adaptation Process goes to the planning step. The Adaptation Planner is the element that represents the planning step, thus it decides how to adapt the system, according to the result of the Adaptation Analyzer.

To allow the handling of multiple adaptations, the Adaptation Process creates an independent (asynchronous) instance of the planner. Hence, after the analyzing step, the Adaptation Planner continues the adaptation process. The Adaptation Planner is a thread that is created only if an adaptation is needed.

When the Adaptation Planner is created, it has to choose how the adaptation has to be performed. It has been initially decided that there are only two types of adaptation in this case study (architectural server-based adaptation and quality of the content visualization) and each type has only two strategies to be performed (add/remove server and improve/reduce quality). Therefore, the Adaptation Planner has a link to those strategies, and it chooses (1) the kind of adaptation required, and (2) the suitable strategies.

To choose the strategy, the Adaptation Planner uses the result that the Adaptation Analyzer has returned after the analyzing step. Even if, in this case study, there are only four strategies, it was designed a reflection mechanism to allow a dynamic update of strategies. Once the strategy has been selected, the planner creates an instance of it. The Adaptation Strategy prepares the object that will perform the adaptation, the Adaptation Performer.

Such as for the strategies, there is not only one Adaptation Performer. Each performer has to run in a particular zone of the system and it uses only that zone's objects. For this reason, there was designed a Strategy design pattern to implement the performers. Each performer is associated to one zone, so in this case study we have two Adaptation Performers: ServerAdaptationPerformer and JspAdaptationPerformer.

The Adaptation Performer executes the strategy tactic-by-tactic [1][6], so that it can be stopped between a tactic and another. Tactics are atomic operations that compose a strategy and the number of tactics performed by a performer gives the Adaptation Level. Therefore, appropriate tactics were defined for the Adaptation Strategies. For the quality of content visualization strategies (improve quality and reduce quality), a tactic was implemented for each reachable visualization quality level, which improves or reduces the quality of the contents. This case study was designed with three levels of quality of content visualization. Hence, the Adaptation Level is incremented for each improved or reduced quality level gained. For the architectural server-based structural strategies (add server and remove server) it was defined a tactic based on the number of servers to add or remove. If a performer has to add three servers, the tactics that compose this strategy are three. Hence, the Adaptation Level is incremented for each added or removed server.

To handle the concurrency adaptations and so multiple performers, which will run concurrently to adapt the system,

 ISBN: 978-1-61208-463-3

in Unibank it was designed an Adaptation Scheduler. The scheduler has the objective of handle every Adaptation Process; it compares them to choose which adaptation process has to run, to stop or to interrupt. Two processes that involve two different architectural zones can run concurrently, each scheduler handles only the processes that run in a specific zone.

Each scheduler is a Singleton. The Adaptation Planner receives the Adaptation Performer from the Adaptation Strategy and then it communicates with the appropriate scheduler to add the performer. The Adaptation Scheduler uses the performer's Adaptation Need to compare it with the other performers that are running in a specific area.

The routine used to compare and manage the performers is based on the runtime adaptation zones and Adaptation Need concept. To simplify, only the performers' stops were considered and not the interruptions. However, it is only one of the multiple solutions that are achievable with these notions.

When the Adaptation Planner adds a performer to the scheduler, the scheduler starts to check the performer's attributes. The Adaptation Scheduler checks first how many performers are running in its zone. If no performer is running (so the runtime zone is in a green state) the scheduler can run the new Adaptation Performer, otherwise it has to check which objects the new performer has to use to adapt the system (the runtime is in a yellow or red state). If the objects, which the new performer has to involve, are in use by other performers, the Adaptation Scheduler has to compare the priority of the new performer with those of the other performers.

If the priority of the other running performers is lower, then the scheduler has to interrupt or stop them and to start the execution of the new performer. Recall that an Adaptation Performer cannot be interrupted or stopped anytime, but it can be stopped after the end of a tactic and the start of another, that is because every performer must leave the objects in a stable state.

If the priority of the other running performers is higher, then the scheduler has to reject the adaptation, because other most important performers (and so adaptations) are running.

If the priority is the same, the scheduler has to compare the Time and the Strategy to choose which performer has the highest priority. This application has a strategy's priority hierarchy for each type of adaptation. For server structural adaptation: add server and remove server. And for quality of contents visualization adaptation: improve quality of contents and reduce quality of contents.

Therefore, when the type of two performers is the same, the Adaptation Scheduler compares the strategy and then uses the same previous routine. When also the strategies are the same, the scheduler gives priority to the shorter performer according to its Time attribute. When two performers are completely equal, the Adaptation Scheduler chooses the performer that was added first.

IV. CONCLUSIONS AND FURTHER WORK

The work presented in this paper has addressed concurrency issues in self-adaptive systems by focusing on how two or more adaptation needs which occur in the same time interval can be properly managed. The issues raised by this topic are mainly due to the fact that the system should address two or more adaptations by itself during its execution. Each adaptation should leave the system in a stable state. Making two changes in a system may be risky also when the system is in the development phase, while during its execution is a challenge.

Several concepts have been used in this paper such as adaptation need, adaptation process, adaptation level, adaptation strategies and tactics, and priorities. Further novel concepts have been introduced for addressing concurrency: architectural adaptation zone and runtime adaptation zone. Our solution will be further refined by considering issues and solutions proposed in various fields such as Self-Organizing Networks (SON) mechanisms for future wireless networks, i.e., LTE [4].

The solution has been validated through a case study called Unibank, implementing a home banking application. Two types of adaptations have been considered: architectural (e.g., the changing number of the used servers) and content-based (e.g., the changing of the content type - textual and multimedia - displayed to the user). The solution enables (1) the concurrent execution of two or more adaptation processes if they involve different architectural zones, (2) the interruption of an adaptation process if another one arise in the same runtime adaptation zone with a higher priority, or (3) the ending of an adaptation process under certain conditions. In the further work, we plan to validate our solution in other case studies and application domains and to use available tools and approaches for formal validation.

In this paper, we have defined architectural zones, architectural levels, adaptation strategies, and adaptation types statically at design time. A future work plans to introduce flexibility in the definition of architectural zones, and enable their definition and/or modifications at runtime.

Another further work concerns the availability of various access devices. Today every person has more types of Internet-connected devices ranging from smartphones and tablets to laptops and desktops. To overcome the visualization problems for different types of devices, we use Bootstrap, which supports responsive Web design. The layout of Web pages adjust dynamically, taking into account the characteristics of the device used for the access of Unibank. This kind of adaptation is different from the ones presented in this paper, being not jet included in the adaptation process of Unibank.

Finally, we plan to measure the performances of our solution addressing concurrent issues considering quality attributes [8][10] and software metrics [11].

 ISBN: 978-1-61208-463-3

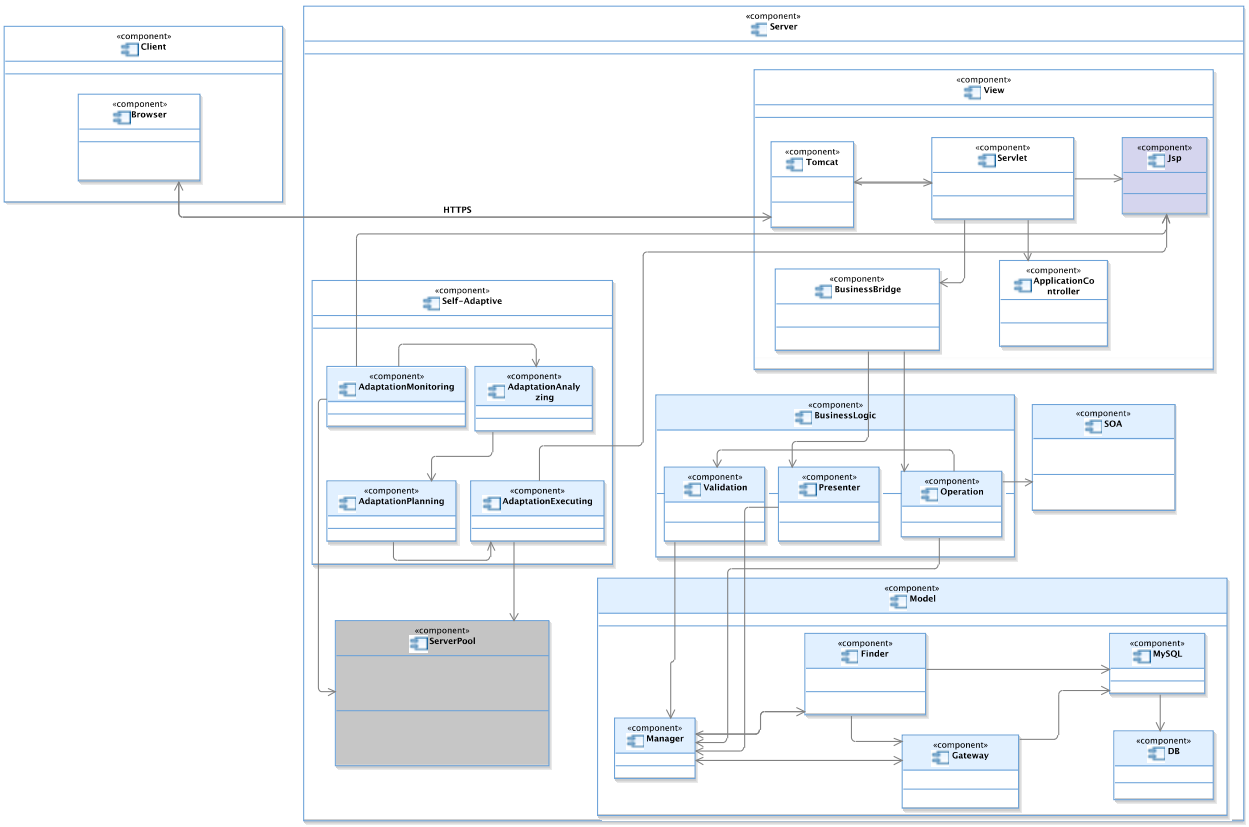

Figure 4. Unibank Architecture

REFERENCES

[1] S. W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation". Journal of Systems and Software, vol. 85, 2012, pp. 1860-2875.

[2] S. W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based Self-Adaptiation in the Presence of Multiple Objectives", SEAMS 2006, pp. 2-8.

[3] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Software Engineering for Self-Adaptive Systems. LNCS 5525, Springer, 2009.

[4] L. Ciavaglia, Z. Altman, E. Patouni, A. Kaloxylos, N. Alonistioti, K. Tsagkaris, P. Vlacheas, and P. Demestichas, "Coordination of Self-Organizing Network Mechanisms: Framework and Enablers", Mobile Networks and Management, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol 97, 2012, pp 174-184.

[5] R. de Lemos, H. Giese, H. Muller, and M. Shaw, Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science 7475, Spinger, 2012.

[6] D. Garlan, S. W. Cheng, A. C. Huang, and B. Schmerl, P. Steenkiste, "Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure". IEEE Computer, Vol. 37, No. 10, IEEE Computer Society Press, 2004, pp. 46-54.

[7] K. E. Harper, J. Zheng, and S. Mahate, "Experiences in Initiating Concurrency Software Research Efforts". 32nd International Conference on Software Engineering, vol. 2, 2010, pp. 139-148.

[8] S. Neti, and H. Müller. "Quality Criteria and Analysis Framework for Self-Healing Systems", ICSE Workshop on Software Engineering for Adaptive and Self-Management Systems, 2007.

[9] C. Raibulet, "Facets of Adaptivity". 2nd European Conference on Software Architecture, LNCS 5292, 2008, pp. 342-345.

[10] C. Raibulet. "Hints on Quality Evaluation of Self-* Systems", 8th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, London, UK, September 8th-12th, 2014.

[11] P. Reinecke, K. Wolter and A. Van Moorsel. Evaluating the Adaptivity of Computing Systems, Performance Evaluation Journal, Vol. 67, pp. 676-693. 2010.

[12] T. Seceleanu and D. Garlan, "Synchronized Architectures for Adaptive Systems". 29th Annual International Computer Software and Applications Conference. Edinburgh, UK, 2005, pp. 146-151.

 ISBN: 978-1-61208-463-3