Towards a Framework for Building Adaptive App-Based Web Applications Using Dynamic Appification

Ashish Agrawal(✉) and T.V. Prabhakar

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, Kanpur 208016, Uttar Pradesh, India
{agrawala,tvp}@cse.iitk.ac.in

Abstract. Appification, the process of building app-based web applications, can help in improving various quality attributes of the application and reduce consumption of resources at server side. A major challenge in ensuring quality attributes of such applications is run-time variations in availability of client resources like battery power. A generic architecture-based approach for building applications that can not only accommodate the dynamic environments by ensuring multiple quality attributes but can also opportunistically exploit the client resources, is missing in the literature. This paper presents a technique called *Dynamic Appification* using which an application can manage its expectations on the environment at run-time. Findings of our investigation on building adaptive applications using this technique are formulated as a methodological framework called *Appification Framework*. Using our framework, we implemented an application that can not only handle the scenarios of low client resources but can also opportunistically exploit the client resources to improve its capacity by more than 100% of the initial capacity.

Keywords: Mobile apps · Dynamic architecture · Adaptive applications

1 Introduction

Appification, the process of building app-based web applications, can also be seen as an opportunity to exploit resources available at the mobile clients. However, mobile devices operate in dynamic environments and availability of resources at client devices can vary with time like battery level, network connectivity, etc [7]. Such environmental changes cause issues in ensuring quality attributes of the application and also limit the application's ability to exploit client resources. For example, if an application is designed with light computation on client devices, it will not be able to fully use the client resources when possible. Existing approaches in the literature to handle dynamic environments at run-time (e.g., *Cyber Foraging* [1] and *Fidelity Adaption* [8]) are focused towards ensuring only a specific set of quality attributes. Also, these solutions accommodate the dynamic environments only from the client perspective (low availability of client

DOI: 10.1007/978-3-319-23727-5_3

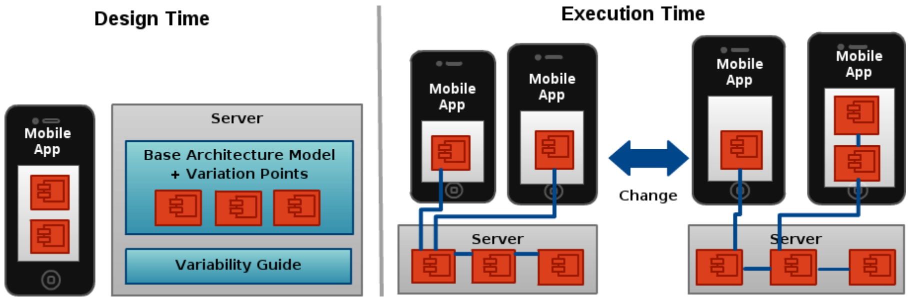

Fig. 1. An example of application adapting using our methodology

resources) and do not consider the server perspective of exploiting the client resources when possible.

Aim of this work is to provide a generic architecture-based approach that can accommodate the dynamic environments while considering both client and server perspectives. Our focus is on investigating the architecture-level decisions that are specific to the appification process. In this process, an important architectural decision is to divide the application components into two groups, one to be executed on the client device and the other to be executed on the server. This decision has an impact on the quality attributes of the application. We call this design decision as ***"Appification Strategy (AS)"*** for that application. An application can have several possible *ASs* which have different impacts on quality attributes. In current approaches, an *AS* is decided during the design phase. This leads to a one-time selection of what the quality attribute trade-offs are likely to be.

We present a technique called *Dynamic Appification* where the *AS* of the application is not fixed at design-time and can vary at run-time. This technique can help in handling dynamic environments from both client and server perspectives. From the client perspective, an application can select an *AS* more suitable for a particular type of client at that time. From a server perspective, among possible *ASs* for a particular client at a given time, an application can choose the one with maximum reduction in operational load on the server. Thus, at run-time, the application can have different *ASs* for different clients and can migrate from one *AS* to another *AS* for any client. Figure 1 depicts an example of adaptation in the application using our methodology. It shows that *AS* for a particular client can be changed by executing more components on the client device.

Findings of our initial exploration to realize *Dynamic Appification* for building adaptive applications are formulated as a methodological framework called *Appificaton Framework*. We have also implemented a simple application as a proof of concept for the *Appification Framework* and conducted experiments to accommodate scenarios of low battery power, intermittent network connectivity and high load on the server. Rest of the paper is structured as follows: Section 2 describes *Appification Framework* in detail. Section 3 explains implementation

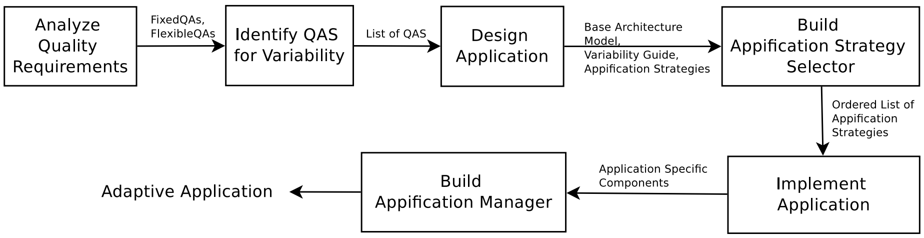
Analyze Quality Requirements
FixedQAs, FlexibleQAs
Identify QAS for Variability
List of QAS
Design Application
Base Architecture Model, Variability Guide, Appification Strategies
Build Appification Strategy Selector
Ordered List of Appification Strategies
Implement Application
Application Specific Components
Build Appification Manager
Adaptive Application

Fig. 2. Steps of the Appification Framework

of our prototype and results of the experiment conducted for handling changes in the environment. Section 4 concludes the work with directions for future work.

2 Appification Framework

This section presents a methodological framework for building applications with the ability to adapt in a dynamic environment at run-time using *Dynamic Appification*. The framework abstracts out the responsibilities involved in building such applications, and provides guidelines for architectural design, implementation and deployment of the application. Figure 2 depicts steps involved in the framework that are further explained in the following subsections.

2.1 Analyze Quality Requirements

The first step is to analyze the application requirements to identify the quality attributes that can be changed at run-time during adaptation. An application may have freedom in parameters of some quality attribute requirements like performance (response-time) may vary between 0 to 10 seconds. Thus, there can exist multiple *ASs* for that application which comply with the restrictions imposed by the quality attribute requirements. In this step, quality attributes desired from the application are categorized into two sets: 1) *FixedQAs* contains quality attributes for which variations are not allowed; 2) *FlexibleQAs* contains quality attributes for which application can have some variations.

2.2 Identify QAS for Variability

In this step, we need to identify the scenarios for which application needs to adapt. The main idea of our approach is to achieve variability quality attribute [4] in the application architecture and requirements for the same are captured through *Quality Attribute Scenarios (QAS)* [3] for variability. Documenting QASs for variability helps in identifying in what situations application needs to adapt and what should be the desired result after adaptation. For example an application can have the following QAS: "If client battery is less than 30%, reduce the energy consumption of the application on that client device".

2.3 Design the Application

In this step, first, all useful *ASs* of the application are identified by analyzing the application components. Among all possible *ASs*, strategies which do not fulfil the constraints on the execution location of the components (e.g., functional, dependency on specific hardware/software components), or do not comply with the quality requirements (*FixedQAs* and *FlexibleQAs*), are discarded. An *AS* is also discarded if there is another *AS* which will always give better quality attributes compared to this one.

Finally, a base architecture model of the application is designed which represents execution location of the components. If a component has same execution location (client or server) in all useful *ASs*, it has a static execution location in the base model. If a component has different execution location in any two useful *ASs*, its execution location is modeled as a variation point. Such a variation point has two choices, either to execute that component on the client device or on the server. This base architecture model will have a set of possible reconfigurations (*ASs*) for run-time execution. Details of variation points (e.g., choices, variation-time, etc.) are stored in the variability guide. These strategies are also evaluated in terms of their impact on quality attributes. Such evaluation results will be used in the process of selecting the best possible *AS*. In case it is not possible to fully quantify such an impact, it is captured in relative terms.

2.4 Build Appification Strategy Selector

In this step, we build a component to facilitate selection of an *AS* for given quality attributes. This component will be used to decide initial *AS* of the application and which *AS* to be used during adaptation. Such selection is a complex problem as there may be a situation in which no single strategy is giving better values for all quality attributes. For example, one *AS* may provide poor performance and better security and another may provide better performance but poor security. We have formulated this problem as a multi-criteria decision making problem and used *Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS)* [5] framework as it selects the solution closest to the best possible solution. Due to page limitation, details of our TOPSIS model are omitted here and the readers are referred to [2].

Output of the TOPSIS model is an ordered list of *ASs*, let's say *ASOrderedlist*, where the *AS* at the top is most preferable (most closest to the desired quality attributes). The application maintains a copy of *ASOrderedlist* for every client and initially, the topmost strategy is selected for current execution. On the occurrence of a QAS, the application selects an *AS* with the highest rank that fulfills the desired QAS response. Such selection will have the minimum adverse effect on other quality attributes. In case the adaptation is requested by the server environment (e.g., load on the server), then the application might have to change *ASs* for a set of client devices. Depending on the desired quality improvement, the number of client users, for whom *ASs* have to be changed, is defined using methods like stress-testing.

2.5 Implement the Application

In order to have the ability to dynamically change the *AS* at run-time, application components having variation points, have to be implemented in a way such that their execution location (client or server) and communication pattern (local or remote) can be changed at run-time. Our technique for building such components is based on the following tactics:

1. **Code Redundancy:** Deploy such components on both server and mobile device to reduce the overhead of transferring components on the fly. It may increase implementation cost as such components may have to be built in both the technology stacks of server and client.
2. **Encapsulate:** Provide an explicit interface of the components such as application programming interface (API). It is required for easy interaction with other components.
3. **Defer Binding:** As the exact execution flow of the application will be decided at run-time, components have to be designed in such a way that they can dynamically decide whether to make a remote call or a local call. Thus, to change the *AS*, application just need to change the call graphs of the components.

2.6 Build Appification Manager

In this step, we implement the functionality of dynamically changing the *AS* for client devices. The application should be able to monitor the environment to trigger a QAS, select *ASs* for clients, and migrate the client(s) to their selected *AS*. For handling these issues, we present a conceptual architecture of such component, *Appification Manager*. Architecture of *Appification Manager* is based on MAPE-K control loop [6] for adaptive systems which clearly abstracts out the responsibilities of an adaptive system. Following describes components of *Appification Manager* in detail.

1. **Context Monitor**: This component monitors the contextual variables (e.g., CPU load) and reports the data to the *Analyzer*.
2. **QAS Analyzer**: By analyzing the data provided by the *Context Monitor*, this component checks if any variability QAS has occurred in the system. For example, it notifies the *Planner* component if CPU load on the server becomes more than 80% continuously for 15 minutes.
3. **Strategy Planner**: This component decides what changes should be incorporated in the application. By using the results from *Appification Strategy Selector*, it identifies the number of users and their new *ASs*.
4. **Executor**: This component is responsible for finally changing the *ASs* of the clients identified by *Planner*. For each user, depending upon the *AS*, it selects the call graph (stored in the *Knowledge Base*), and updates both server and client with this information.
5. **Knowledge Base**: This component maintains a repository of architectural information used by other components of the manager.

3 Case Study

We have implemented a simple application as a proof of concept for the *Appification Framework* and conducted experiments to show the feasibility of two kinds of adaptations: 1) *Server-driven adaptation* in which server changes *ASs* of a set of clients to handle high operational load, and 2) *Client-driven adaptation* in which a single client adapts its *AS* to handle low availability of resources at the client-side. Our application facilitates image-based searching of products and has mainly three components; **TakeImage** for capturing an image, **ImageToText** for extracting text from an image, and **Search** for searching the textual content in a product database. We identified two *ASs* for the application as:

– as_1: Component **TakeImage** executes on mobile client; **ImageToText** and **Search** execute on server.
– as_2: Components **TakeImage** and **ImageToText** execute on client; **Search** executes on server.

For client devices, components are implemented in Java for the Android platform. For server, components are implemented in Python using Django framework. **ImageToText** is build using Tesseract-OCR library [9] for both the Android platform and the server. The server part is deployed on a virtual machine with 1 CPU core and 2 GB RAM. Here, the application can have some variations in performance, energy-efficiency on the clients, and capacity. However, the average response-time of requests should not be more than 10 seconds. Following sections explain the two kinds of adaptations investigated by us:

3.1 Server-Driven Adaptation

We demonstrate the ability of the application to accommodate dynamic changes in server environment (user-load) by opportunistically exploiting the resources available at the client devices. QAS for such adaptation is:

– **QAS1**: "If the server reaches 85% of its capacity, reduce the load on the server".

In our case, capacity is represented as the number of clients the application can serve while maintaining the average response-time to less than 10 seconds. We selected as_1 as the initial *AS* for the application in order to minimize the energy consumed on client devices. Without dynamic appification, the server has a capacity to handle around 46 simultaneous users. Compared to as_1, strategy as_2 has less operational load on the server as **ImageToText** is executed on client. Thus, to accommodate high user-load, the application changes the *ASs* for a set of clients. In order to identify the number of such clients for a given user-load, we performed profiling of the application. For a given user-load, average response time is calculated with varying number of users moved to as_2, as shown in Figure 3(a) and 3(b). For example, in case of 60 users, for reducing average response time to less than 10 seconds, at least 10 users should be moved to as_2.

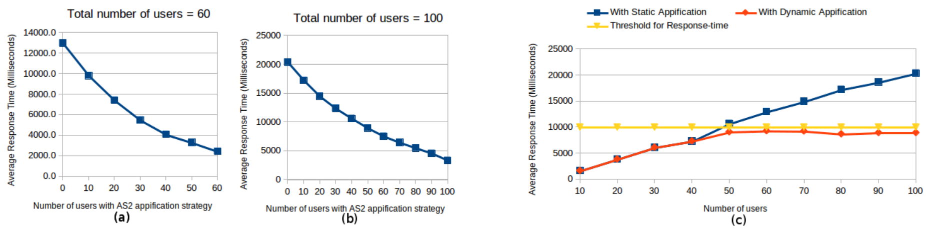

Fig. 3. Experimental results: (a) & (b) depict profiling results for different user-load. Here, average response-time is depicted with varying number of users appified to as_2 strategy. (c) depicts application behaviour with and without dynamic appification

Figure 3(c) presents results of dynamically varying capacity of the application to 100 users. Here, with static appification (with strategy as_1), average response-time goes more than 10 seconds after 46 users. With dynamic appification, the application behaves similar to static appification till 40 users. After 40 users, the application dynamically scales to increase its capacity. In case of 100 users, application changes ASs of 50% users to maintain the average response-time to less than 10 seconds. This adaptation has an adverse impact on energy consumptions of the client devices. By changing AS dynamically, an application can increase its capacity only up to a limit. For example, in our case, the application can not handle more than 165 users even by using as_2 strategy for all users. Here, contrary to the traditional/cloud approach, capacity is improved without adding new server-side resources. Thus it does not increase operational-cost of the application. One thing to note here is that the improvement in the capacity will directly depend upon the amount of computation off-loaded to the client devices. Aim of this experiment was to show feasibility of our approach. Exact improvement in capacity may vary with application.

3.2 Client-Driven Adaptation

Here, to handle the environmental changes, a client adapts its AS at run-time. The QASs for such adaptations are:

- **QAS2**: "If the battery power at a client is less than 30% of full power, reduce energy consumption at the client device". To test this scenario, initially the client is configured with as_2 strategy. By moving to as_1, client can reduce the energy consumption by removing the overhead (0.95 seconds of execution time) of executing component **ImageToText**.
- **QAS3**: "If the client is having intermittent network connectivity, reduce response-time from the server so that the dependency on a stable network is narrowed". To test this scenario, initially the client is configured with as_1 strategy. By moving to as_2, the application can reduce server part of response-time from 0.24 seconds to 0.11 seconds.

4 Conclusion and Future Work

Dynamic Appification can help in solving various quality related issues such as; unpredicted and dynamic quality requirements, energy constraints, intermittent network connectivity, etc. In this paper, we explored how to realize *Dynamic Appification* in the application. In our approach, variability is introduced in the application architecture by modeling the appification-specific design decisions as variation points. Such architecture supports multiple variants that differ in terms of their impact on the quality attributes and consumption of resources. Thus, depending upon the environmental context, an application can adapt at runtime by migrating to a suitable variant. Selection of the best suitable variant is done in a manner such that the adverse effects on other quality attributes are minimum. We presented a methodological framework, called *Appification Framework* to provide guidance on building adaptive applications with *Dynamic Appification*. Experiments conducted on a prototype implementation showed that the application can not only handle scenarios of low client resources, but can also dynamically scale by exploiting resources available at the client devices. In the future, we would like to explore on automating the framework activities in order to reduce the design and development overhead.

Acknowledgments. Acknowledgments The authors gratefully acknowledge the financial support from Tata Consultancy Services and MHRD, Govt. of India for this work.

References

1. Avgeriou, Paris, Zdun, Uwe (eds.): ECSA 2014. LNCS, vol. 8627. Springer, Heidelberg (2014)
2. Agrawal, A., Prabhakar, T.V.: Using topsis for decision making in software architecture. http://www.cse.iitk.ac.in/users/agrawala/topsis.html (retrieved June 2015)
3. Bachmann, F., Bass, L., Klein, M.: Deriving architectural tactics: A step toward methodical architectural design, technical report, CMU/SEI-2003-TR-004 (2003)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3rd edn. Addison-Wesley Professional (2012)
5. Hwang, C., Yoon, K.: Multiple Attribute Decision Making: Methods and Applications. Springer, New York (1981)
6. Jacob, B., Lanyon-Hogg, R., Nadgir, D.K., Yassin, A.F.: A practical guide to the to the ibm autonomic computing toolkit, April 2004
7. Messer, A., Greenberg, I., Bernadat, P., Milojicic, D., Chen, D., Giuli, T.J., Gu, X.: Towards a distributed platform for resource-constrained devices. In: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS 2002), p. 43 (2002)
8. Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K.R.: Agile application-aware adaptation for mobility. In: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP 1997 (1997)
9. Smith, R.: An overview of the tesseract ocr engine. In: ICDAR, vol. 7, pp. 629–633 (2007)