A Framework for Using Component Redundancy for self-Optimising and self-Healing Component Based Systems

Ada Diaconescu[*], John Murphy[*]
Performance Engineering Laboratory, Dublin City University
{diacones,murphyj}@eeng.dcu.ie

Abstract

The ever-increasing complexity of software systems makes it progressively more difficult to provide dependability guarantees for such systems, especially when they are deployed in unpredictably changing environments. The Component Based Software Development initiative addresses many of the complexity related difficulties, but consequently introduces new challenges. These are related to the lack of component intrinsic information that system integrators face at system integration time, as well as the lack of information on the component running-context that component providers face at component development time.

We propose an addition to existing component models, for enabling new capabilities such as adaptability, performance optimisation and tolerance to context-driven faults. The concept of 'component redundancy' is at the core of our approach, implying alternate utilisation of functionally equivalent component implementations, for meeting application-specific dependability goals.

A framework for implementing component redundancy in component-based applications is described and an example scenario showing the utility of our work is given.

1. Introduction

Extensive employment of software systems in various domains raised the concern for the dependability guarantees provided by such systems (e.g. performance, reliability, robustness). Nevertheless, the ever-increasing size and complexity of modern software systems leads to more complicated and expensive system design, testing and management processes, decreasing system flexibility and making it difficult to control dependability characteristics of such systems [1].

In this context, Component Based Software Development (CBSD) has emerged as a new solution that promises to increase the reliability, maintainability and overall quality of large-scale, complex software applications. In the CBSD approach, software applications are developed by assembling made or bought (i.e. commercial off-the-shelf - COTS) components, according to a well-defined software architecture.

Consequently, the dependability of component-based software applications is determined by both the dependability of the individual components involved, as well as by the adopted software architecture. Considerable research efforts towards determining optimal software architectures ([2], [3], [4]) with respect to the system quality attributes [5], as well as towards achieving dependability guarantees for COTS components ([6], [7]), support this idea. The impact software architecture has on the overall software system performance is also demonstrated in [8]. In this paper, it is shown how different software architectures, providing the same functionalities, yielded different performance results while running in identical environmental conditions.

Information on the context in which a software component or application will run (e.g. hardware and software resources, workloads and usage patterns) is vital when taking architectural, or design decisions. At software development or integration time though, it is impossible to predict with sufficient accuracy, the environmental conditions in which software components or applications will be deployed. In addition, the initial deployment conditions can dynamically change at runtime. Using COTS components exacerbates the problem, by increasing the level of indetermination and making it hard to provide dependability guarantees for the running system [6], [7].

Assuming that different architectural, design and implementation-related choices proved optimal in different environmental circumstances, we argue that it would be beneficial for system quality if the software application could accordingly adapt at runtime, when accurate information were available. We propose the use of redundancy in order to enable such capabilities for component-based software systems. Our intent is to enhance one of the existing component platforms (e.g. EJB, .NET, or CCM) with support for software component redundancy. The predicted benefits of this approach include constant, automatic performance optimisation for running applications, as well as tolerance to certain categories of non-functional, integration-specific faults (e.g. deadlocks, data corruption). By non-

[*] The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2001

functional faults, we mean faults that are not related to an application's expected functionality and therefore do not imply any application-specific behavioural knowledge or extra implementation effort to detect.

The rest of the paper is structured as follows. Section 2 provides an overview of our research proposal. An example scenario, indicating the benefit of our work, is presented in Sections 3. A general architecture for our proposed framework is described in Section 4. Section 5 places our approach in the context of similar work in the area. We conclude and present future work in Section 6.

2. Research overview

Our research goal is to enable dynamic adaptability capabilities in complex, component-based software systems, running in unpredictably changing environments, in order to automatically optimise and maintain their dependability characteristics.

Central to our solution is the concept of (software) *component redundancy*. By this concept, we mean that a number of component implementation variants, providing the same or similar services, are available at runtime. We refer to these component variants as *redundant components* and say that a set of redundant components providing an equivalent service constitutes a *redundancy group* (with respect to that service). Any component variant in a certain redundancy group can be functionally replaced with any other component variant in the same redundancy group.

Only one of the redundant components providing a service is assigned, at any moment in time, for handling a certain client request for that service (i.e. an instance of that component is forwarded the client request). This differs from other approaches (e.g. N-version programming; agent-based systems [9]), where a number of the available redundant variants work in parallel, towards a common result. We refer to a component variant that the application is currently using (i.e. sending client requests to instances of that component version) as an *active* component variant. Component variants that are not currently considered for handling client requests are referred to as *passive* component variants.

If instances of an active component variant fail, or perform poorly in a certain context, the component variant can be *deactivated* and replaced with an alternative member of the same redundancy group. This is the main means by which redundancy groups continually optimise themselves, while dealing with changing execution contexts, or context-driven faults.

We do not constrain the component redundancy concept to the level of atomic components [10] [Figure 1-a]. This concept can also be applied to composite components [Figure 1-b] (i.e. composites [10], 'containing' a number of sub-components) or to component sets, or groups (i.e. components 'using' other components) [Figure 1 – c]. Therefore, through the rest of the paper, references to redundant components can imply atomic, composite, or sets of components.

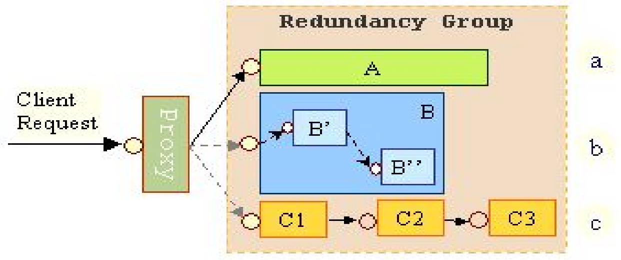

Figure 1: Redundancy granularities

We intend to implement all functionalities that are required to support and benefit from component redundancy at the component platform level. No development effort overhead is to be placed on the developers of software components that are to be deployed and run on such platforms. Of course, in order for redundancy to be enabled, alternative variants would have to be provided. However, our approach does not require that multiple redundant components be available at software application deployment or runtime. The only constraint is that at least one component version must be available for each external interface, at all times. While complying with this constraint, redundant components can be dynamically added or removed from the software system, at runtime.

We propose that a formal component description be available for every deployed component variant. The description includes information on both functional (e.g. provided and required services) and non-functional (e.g. quality attributes, recommended resources) characteristics of the component (e.g. similar to contracts as in [10], or [11]). Most system quality characteristics depend upon the execution context (e.g. response time is influenced by workload and available resources). These variations are represented in component descriptions as a list of [environment related parameters, corresponding values] pairs. Initially, component non-functional characteristics can optionally be provided by component developers, based on estimations, test results, or previous experience with the supplied components. While a component variant is active, its initial quality description is updated with run-time monitoring information, for the precise application configuration and execution environment.

3. Example

In this section, we provide an example of a possible scenario in which our approach proves to be beneficial. For this example, we opted for the EJB component technology. However, we believe our framework is generic enough to be applied to other component models.

The example involves two different component implementations providing the same functionality: repeatedly retrieving information from a remote database. The two components differ at the design level. The first design variant involves a single Session Bean, containing SQL code for directly accessing the database. We will refer to this variant as the Direct DB variant. In the second design variant, a Session Bean uses an Entity Bean as means of interacting with the database. We will refer to this variant as the Using Entity Bean variant. A client Session Bean is used for calling these two variants, repeatedly requesting information.

We deployed our EJB example on an IBM WebSphere application server, on Windows2000, running on an Intel Pentium4, with 1.6GHz CPU and 512 MB RAM. We used a DB2 database, running on Windows2000, Intel Pentium 4, 1.6 GHz CPU and 256 MB RAM. A third machine was used for generating traffic and loading the network link to the remote database, to various degrees. We used the Tfgen traffic generator for this purpose. The three machines were connected through a switched 100 Mbps Ethernet LAN, completely separated from other traffic.

We measured the response delays for each version, in different environmental conditions (i.e. available bandwidth on the network links) and usage patterns (i.e. number of repetitive read requests per client transaction).

When the network is lightly loaded, we experience smaller delays in the Direct DB variant than in the Using Entity Bean variant, regardless of the number of repetitive client requests (e.g. 1, 10, 100, 1000 [requests per transaction]). This can be accounted for by the overhead incurred (in the Using Entity Bean variant) by the extra inter-process communication and Entity EJB management.

However, increasing the load on the network link to the remote database has significant impact on the Direct DB approach, while hardly affecting the Using Entity Bean. This can be explained by the fact that the Direct DB variant needs to access the database for each individual (client) read request. The Using Entity Bean variant, involves a single database access per client transaction (i.e. only for the first read request in the transaction), as the data is then locally stored at the Entity Bean instance level and retrieved from there for subsequent requests. Therefore, for increased network loads (e.g. 90% load) and number of read requests, the Direct DB design choice produces higher delays than the Using Entity Bean does. Using an Entity Bean to read from the database becomes, in these circumstances, the optimal choice.

The optimal variant switching point between the two implementations is reached when the inter-process communication and CPU overhead (i.e. in the Using Entity Bean variant) is exceeded by the repeated remote database access overhead (i.e. in the Direct DB variant). Figure 2 shows the response-time curves corresponding to the two redundant variants, for various network loads, when 1000 read requests were made per client transaction. For obtaining these curves, we repeatedly measured the response delays of such repetitive client requests, for different network loads. We then calculated the average delay value, for each network load.

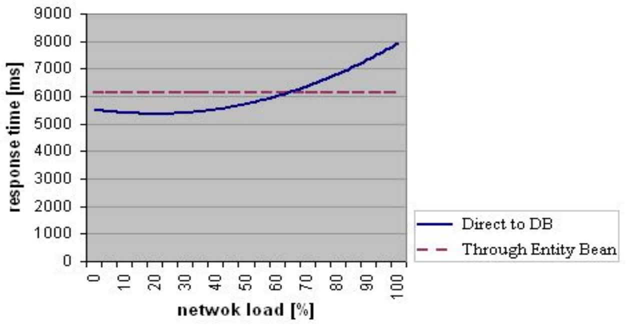

Figure 2: response-time curves

Even though simple, this example shows how alternating the activation of two redundant variants can ensure better performance than either variant could, at all times. We argue that it is hard, if not impossible to devise a component version that exhibits optimal characteristics in all possible running contexts. The optimal component variant depends on the component execution environment, which can frequently change. Our focus is on the adaptation logic for automatically determining optimal component variants and optimal combinations of component variants, in different running contexts.

4. Framework general architecture

We propose implementing component redundancy as a new service provided by component platforms (i.e. besides already provided services, such as security, transaction support, or life-cycle management). Three main functionalities were identified as needed for the support, utilization and management of redundant components and were associated with three logical tiers in our framework [Figure 3]: i) Monitoring tier; ii) Evaluation tier and iii) Action tier. In this section, we present the main roles and functionalities of each of these tiers and discuss the way they interact in order to provide the component redundancy service.

The Monitoring tier is concerned with acquiring run-time information on the software application as well as on its execution environment. Run-time monitoring implies that information is collected exclusively for the active component variants. It is also the responsibility of the Monitoring tier to analyse the collected information and identify any potential ‘problem’ components [1], [12].

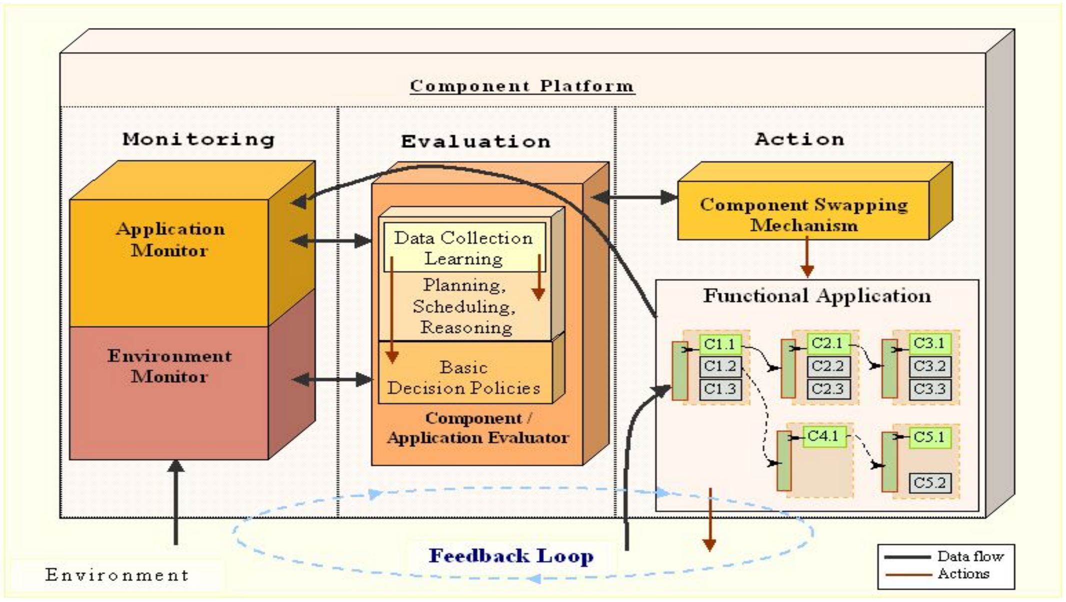

Figure 3: Framework architecture

The Evaluation tier is responsible for determining the optimal redundant component variant(s) in certain contexts, using adaptation logic, component descriptions and monitoring information on the current environment and application state. It also updates the descriptions of active component variants, with runtime information from the Monitoring tier. This helps the Evaluation tier to 'learn' in time about the performance characteristics of the software application it has to manage.

Adaptation logic, for deciding which redundant component(s) to activate (and deactivate respectively), is reified in the Evaluation tier in the form of decision policies. These are sets of rules, dictating the actions to be taken in case certain conditions are being satisfied. Decision policies can be customised for each deployed application (e.g. requested quality attributes values, default redundant components to activate) in order to serve the specific application goals and can be dynamically added, modified or deleted at runtime.

We split decision policies into two layers, based on their complexity. The bottom layer comprises basic decision policies, of the condition-action type. These policies are used to remedy poor performance or critical situations (e.g. response time thresholds are being exceeded) and take immediate effect. The top layer is reserved for decision policies concerned with application optimisations, in conditions in which the application is not necessarily evaluated as under-performing or faulty. These policies are designed for activities such as reasoning, predicting, planning, or scheduling, in order to optimise application performance, anticipate and prevent failures or emergencies. Policies in the top layer are also used to control the adaptation process. They can decide when to stop an optimisation evaluation or enforcement operation, in case it becomes too costly (e.g. in time, or resources), or it seems to have entered an infinite loop (e.g. oscillating state, chain reaction).

The Action tier encompasses the actual software application and a component-swapping mechanism. Based on optimisation decisions, the Evaluation tier sends corresponding configuration commands to the Action tier, indicating the redundant component variant(s) to be activated or deactivated respectively. The component-swapping mechanism performs the requested operations. As stated in related research on component hot swapping, two main issues occur when replacing component variants at runtime. One issue is concerned with state transfer from an executing component instance to a replacement component instance. This is only needed in case instances of different component variants handled the same client request or session, one after the other. Since in the targeted problem domain client calls are usually short-lived, we believe such action would bring little performance benefit to requests already being handled (when component replacement occurred). Therefore, in a first phase, we do not attempt to transfer state between instances of different component variants. Rather, incoming client calls are directed to an instance of the appropriate component variant, upon arrival. Instances of component variants to be deactivated finish handling current requests before being removed. This allows for instances of different redundant components to coexist. In a future phase, we will consider one of the solutions proposed in the ongoing research in this area (e.g. [13],

[14]). The other issue is maintaining client references consistency. We adopt a proxy-based solution to address this issue. Component technologies based on contextual composition frameworks [10] provide a straightforward way of implementing this. That is because clients can only call component instances through the component container, in which the component was deployed and run. The component container can consequently be modified so that to transparently (re)direct client requests to instances of active component variants. In brief, in a first phase of our research, we adopt a client request indirection strategy for implementing the component hot-swapping mechanism.

In our framework, the three presented tiers operate in an automated, feedback-loop manner [Figure 3]: the application performance is monitored and evaluated, the optimal redundant component(s) are identified and activated and the resulting application is monitored and (re-)evaluated. Decision policies at both layers can be dynamically tuned in effect. It is important to note that as these are logical tiers, the boundaries between them may not be as clearly marked when implemented.

4.1. Hierarchical adaptation mechanism

When considering large-scale component-based applications, global optimisations may not always be desirable. Evaluating an overall application, potentially consisting of hundreds of components, whenever an individual component or a group of components does not behave as expected, might induce unnecessary overhead and not scale well. We propose distributing the adaptation mechanism. That is, if a problem is detected at an individual component level, the problem is dealt with locally, by means of redundant component replacement. Nevertheless, exclusively concentrating on local optimisations might not globally optimise the system. Therefore, our framework employs (three-tiered) adaptation mechanisms with different scopes (e.g. local, group, global), organised in a hierarchal manner. Detected problems can be dealt with locally or/and signalled upwards the hierarchical tree, up to the global level. Adaptation mechanisms can be dynamically activated or deactivated, in order to reduce overhead, when possible. This idea is also presented in [12], in the context of non-intrusive, EJB system monitoring.

5. Related Work

Redundancy for increased robustness or reliability has been successfully used in various domains (e.g. hardware, mechanics, or constructions). The same concept was introduced in the software domain (e.g. [9], or as ‘design diversity’ in [6], [15]), in order to achieve fault-tolerance capabilities for software systems. A few examples of fault tolerant schemes implementing this concept are N-version programming, N self-checking software, recovery blocks [16], or exception handling approaches. However, as these schemes target system fault tolerance, they imply both the presence of knowledge of the correct system behaviour, as well as of methods for assessing system behaviour at runtime, in order to detect faults. We target a different problem domain, encompassing performance-related problems and non-functional faults, which can generally be detected without needing application semantics information. Our framework can consequently be implemented as part of the component platform layer, for the benefit of all applications deployed on such platforms.

Similar to our performance optimisation related intent, the Open Implementation initiative [17] allows clients to decide which implementation variant to use (i.e. instantiate) for optimal performance, in a specific context. We propose that the component platform automatically take such decisions. In our view, it is very expensive, or even impossible for a system manager to optimally perform such activities in due time, in the case of complex systems or frequent environmental changes.

Redundancy as a means of achieving dependability for Internet systems (i.e. Web Services based) is proposed in the RAIC [13] project. The addressed problem domain in this case however, is different in scope from our work. This is because such systems rely on Internet services offered by different providers, from different locations. No single authority owns, or has complete control over the entire system. The Internet system developer has no knowledge of, or access to the implementation, deployment platform, or supporting resources of the services it needs to use. Redundancy support cannot be implemented in this case at the component deployment platform level. Instead, redundancy support for the services that Internet systems use is implemented at the software application level of such systems.

Research in the area of dynamic component versioning presents certain similarities to our work. However, the main intent of the two research directions is different, emphasising different aspects. Component versioning is concerned with verifying whether new versions are better than old ones, before dynamically upgrading the system. In [14] for example, the best component version is determined by means of online testing. Even though the possibility of multiple versions being kept is considered, the way such versions are to be used is not elaborated.

A significant research area, closely related to our work, is concerned with specifying and building dynamic adaptability capabilities for self-repairing systems. Mostly related to our work are approaches based system architectural models [18], [19]. A feedback-loop mechanism (separated from system business logic) is employed for adapting running systems to changing requirements, or environmental conditions. This mechanism is designed in a centralised manner. Monitoring information is centralised, evaluated using analytical methods (e.g. queuing theory) and the system is globally optimised. Our approach adopts a hierarchical adaptation approach, where global system optimisation can generally be avoided. We focus on adaptation operations related to redundant component replacements.

An important aspect of our research is the fact that we exclusively target component-based applications based on contextual composition frameworks [10]. The unique nature of such applications (e.g. soft inter-component bindings; unpredictable number of component instances) might make approaches devised for component-based systems in general (i.e. in which 'components' can mean clients, servers, or software modules), difficult to apply.

6. Conclusions and Future Work

This paper proposed the use of component redundancy for enabling self-optimisation, self-healing and dynamic adaptation capabilities in component-based software systems. A component redundancy related terminology was defined. We argued that system complexity, lack of sufficient information and changing execution conditions make it impossible to create and ascertain components that exhibit optimal dependability characteristics at all times. An example was presented to support this idea. In this example, different strategies were selected for implementing two distinct component variants providing the same functionality. Each implementation variant proved optimal (with respect to response delays) in different environmental conditions. As these results indicate, knowledgeably alternating the usage of redundant components, optimised for different running contexts, ensures better overall performance than either component variant could provide.

A framework for implementing the component redundancy concept was described. We identified the main roles and functionalities this framework needs to provide and categorised them into three logical tiers: monitoring, evaluation and action. We proposed distributing the three logical tiers, organising them (each) in a hierarchical manner, in order to reduce overhead.

As future work, we intend to provide a proof-of-concept implementation of our framework and test it against our example scenario. In addition, further scenarios and case studies will be identified and documented. The cost of acquiring multiple redundant components, as well as the impact of using redundant components on the overall application performance and resource usage will have to be analysed.

7. References

[1] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing", IEEE Computer, January 2003

[2] C. U. Smith, L. G. Williams, "Software Performance Engineering: A Case Study with Design Comparisons", IEEE Trans. Software Eng., Vol. 19, No 7, July 1993

[3] F. Aquilani, S. Balsamo, P. Inverardi, "Performance Analysis at the Software Architectural Design Level", Performance Evaluation, Volume 45, Number 2-3, July 2001

[4] J. Bosch, P. Molin, "Software Architecture Design: Evaluation and Transformation", IEEE Conference and Workshop on Engineering of Computer-Based Systems, Nashville, Tennessee, March 1999

[5] M. Klein et al., "Attribute-Based Architecture Styles", in Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, 1999, pp 225-243

[6] P. Popov, L. Strigini, A. Romanovsky, "Diversity for Off-The-Shelf Components", International Conference on Dependable Systems&Networks, NY, USA, 2000, pp. B60-B61

[7] P. A. C. Guerra, C. M. F. Rubira, R. de Lemos, "An Idealized Fault-Tolerant Architectural Component", Workshop on Architecting Dependable Systems, Orlando, FL, May 2002

[8] E. Cecchet et al., "Performance and Scalability of EJB Applications", Proc of 17th ACM Conference on Object-Oriented Programming, Seattle, Washington, 2002, pp 246-261

[9] M.N. Huhns, V.T. Holderfield, "Robust Software", Agents on the Web, IEEE Internet Computing, March/April 2002

[10] C. Szyperski, with D. Gruntz and S. Murer, "Component Software: Beyond Object-Oriented Programming", Second Edition, Addison-Wesley Pub Co, 1 November 2002

[11] B. Meyer, C. Mingins, H. Schmidt: *Trusted Components for the Software Industry*. IEEE Computer 5/1998, pp. 104-105

[12] A. Mos, J. Murphy, "Performance Management in Component-Oriented Systems using a Model Driven Architecture Approach", The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Lausanne, Switzerland, September 2002

[13] C. Liu, D. J. Richardson, "RAIC: Architecting Dependable Systems through Redundancy and Just-In-Time Testing", ICSE, Workshop on Architecting Dependable Systems (WADS), Orlando, Florida, 2002

[14] M. Rakic, N. Medvidovic, "Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach", Symposium on Software Reusability: putting software reuse in context, Toronto, Ontario, Canada, 2001

[15] B. Littlewood et al., "Modeling software design diversity: a review", ACM Press, New York NY, USA, 2001, pp 177-208

[16] B.Randell and J.Xu, "The Evolution of the Recovery Block Concept", Software Fault Tolerance, JohnWiley&SonsLtd, 1995

[17] G. Kiczales, "Beyond the Black Box: Open Implementation", IEEE Software, January 1996

[18] S. Cheng et al., "Using Architectural Style as a Basis for Self-repair", Proc. Working IEEE/IFIP Conference on Software Architecture, Montreal, August, 2002

[19] P. Oriezy et al., "An Architecture-Based Approach to Self-Adaptive Software", IEEE Intelligent Systems, May/June 1999, p. 54-62