Using Real Options to Select Stable Middleware-Induced Software Architectures

Rami Bahsoon, Wolfgang Emmerich, and Jonathan Macke
Dept. of Computer Science, University College London
Gower Street, WC1E 6BT, London, UK
{r.bahsoon | w.emmerich} @cs.ucl.ac.uk

Abstract

The requirements that force decisions towards building distributed system architectures are usually of *non-functional* nature. Scalability, openness, heterogeneity, and fault-tolerance are examples of such non-functional requirements. The current trend is to build distributed systems with *middleware*, which provide the application developer with primitives for managing the complexity of distribution, system resources, and for realizing many of the non-functional requirements. As non-functional requirements evolve, the "coupling" between middleware and architecture becomes the focal point for understanding the *stability* of the distributed software system architecture in the face of change. We hypothesise that the choice of a stable distributed software architecture depends on the choice of the underlying middleware and its *flexibility* in responding to future changes in non-functional requirements. We devise an option-based model to value such flexibility and guide the selection. We empirically evaluate the model using a case study that adequately represents a medium-size component-based distributed architecture. We report on how a likely future change in scalability could impact the architectural structure of two versions, each induced with a distinct middleware: one with CORBA and the other with J2EE. Our hypothesis is verified to be true for the given change. We conclude with some observations that could stimulate future research in the area of relating requirements to software architectures.

Keywords. Architectural economics; economics-driven software engineering; evolution of non-functional requirements; middleware; real options theory; relating requirements to software architectures.

1. Introduction

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. A 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 [Finkelstein, 2000]. 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.

The requirements that drive the decision towards building a distributed system architecture are usually of a non-functional and global nature [Emmerich, 2000a]. 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) [Sun Microsystems Inc., 2002] and the Common Object Request Broker Architecture (CORBA) [Object Management Group, 2000]. Middleware simplifies 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 in the architecture of the software system induced. 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 [Di Nitto and Rosenblum, 1999]. 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. 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 [Emmerich, 2000b]. Hence, as the non-functional requirements of the software system evolve, the "coupling" between the middleware and the architecture becomes the focal point for understanding the *stability* of the distributed software system architecture in the face of the change.

In an earlier paper [Emmerich, 2002], we reflected on the architectural stability problem with a particular focus on developing 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 advocated 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 lifetime of the architecture. These ranges of requirements may then inform the selection of distributed components technology, and subsequently the selection of application server products. 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 due to the incapability of the middleware to cope with the change(s), when the non-functional requirements evolve (e.g., increased scalability 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, modifying the client(s), and so forth. Such solutions could be problematic, costly, and unacceptable.

We argue that the choice of the distributed software system architecture has to be guided by the choice of the underlying middleware and its flexibility in responding to future changes in non-functional requirements. This is necessary to facilitate the evolution of the software system, to avoid unnecessary future investments (e.g., maintenance overhead, hardware investments, reverting the choice of the middleware etc.), and to ensure that future resources will be used efficiently as the requirements evolve (e.g., new servers are purchased or cycles are leased, only when necessary). As a motivating example, consider a distributed software architecture that is to be used for providing the back-end services of an organization. This architecture will be built on middleware. Depending on which middleware is chosen, different architectures may be induced [Di Nitto and Rosenblum, 1999]. These architectures will have differences in how well the system is going to cope with changes. For example, a CORBA-based solution might meet the functional requirements of a system in the same way as a distributed component-based solution that is based on a J2EE application server. A notable difference between these two architectures will be that increasing scalability demands might be easily accommodated in the J2EE architecture because J2EE primitives for replication of Enterprise Java Beans can be used, while the CORBA-based architecture may not easily scale. The choice is not straightforward as the J2EE-based infrastructures usually incur significant upfront license costs. Thus, when selecting an architecture, the question arises whether an organization wants to invest into an J2EE application server and its implementation within an organization, or whether it would be better off implementing a CORBA solution. Answering this question without taking into account the *flexibility* that the J2EE solution provides and how *valuable* this flexibility will be in the future might lead to making the wrong choice.

The novel contribution of this article is in two interrelated folds:

A. We devise a real option-based model to value the flexibility of the middleware-induced software architecture in response to changes in non-functional requirements. We describe how options theory can be used to inform the selection of potentially more stable middleware-induced software architectures. We argue that the problem of selecting a particular middleware to induce a given architecture is an *option* problem. An option gives its owner the right without the symmetric obligation to invest in the future ending with an expiration date [Hull, 1997; Cox et al., 1979; Schwartz and Trigeorgis, 2000]. From the evolution perspective, the flexibility of the middleware induced-architecture in coping with changes in non-functional requirements has a value that can *assist* in predicting the stability of software architectures. More specifically, flexibility adds to the architecture values in the form of *real options* that give the right but not a symmetric obligation- to evolve the software system and enhance the opportunities for strategic growth. The added value is strategic in essence and may not be immediate. It may take the form of (i) accumulated savings through coping with the change without "breaking" the architecture, mostly these are changes in non-functional requirements; (ii) extending the range of services while leaving the architecture intact; and (iii) the ability to respond to competitive forces and changing market conditions that may pause higher Quality of Service (QoS) requirements, such as the demands for higher availability, scalability, reliability and so forth. From an early development perspective, given several middleware candidates, the architect has the right without the symmetric ob-

ligation to embark on a selection for inducing an architecture. A "wise" selection could be regarded as an investment to buy flexibility, which could be valued as future *growth options* [Schwartz and Trigeorgis, 2000] on the architecture of the software system. These options differ from one middleware to another.

Our application of real options theory to inform the selection of a "more" stable middleware-induced software architectures is novel. The model, which we devise to value the flexibility of the middleware-induced software architecture, in response to likely changes in non-functional requirements, builds on ArchOptions [Bahsoon and Emmerich, 2003a; Bahsoon, 2003]. Given several middleware candidates, the devised model informs the tradeoff analysis and consequently the selection through a simple calculation.

B. We have empirically simulated the model using a case study that adequately represent a medium-size component-based distributed architecture. We have instantiated two versions of the core architecture; each induced by a different middleware, one with CORBA and the other with J2EE. We report on how a likely future change in scalability, as a representative critical change in non-functional requirements, could impact the architectural structure of the two versions. The results of the case study could be summarized as follows. Our hypothesis that middleware induced software architecture differs in coping with changes is verified to be true for the given change. On the methodology level, the results show that value-based reasoning and real options can provide insights on the stability and investment decisions related to the evolution the software. On the discipline level, the study draws some preliminary lessons and insights that could stimulate future research in the area of relating requirements to software architectures and consequently advance our understanding to the architectural stability problem, when addressed from a non-functional requirements perspective.

The article is further structured as follows. In section 2, we describe how we used options theory to inform the selection of middleware-induced software architectures. In Section 3, we empirically evaluate the model, verify our hypothesis, and draw some observations that could simulate future research in the area of relating requirements to software architectures. In Section 4, we discuss related work. Section 5 concludes.

2. Selecting Stable Middleware-Induced Software Architectures with Real Options

Real options analysis recognizes that the value of the capital investment lies not only in the amount of direct revenues that the investment is expected to generate, but also in the future opportunities that flexibility creates [Erdogmus et al., 2002; Erdogmus and Favaro, 2002]. These include growth, abandonment or exit, delay, and learning options. An option is an asset that provides its owner the right without a symmetric obligation to make an investment decision under given terms for a period of time into the future ending with an expiration date [Hull, 1997; Cox et al., 1979; Schwartz and Trigeorgis, 2000]. If conditions favourable to investing arise, the owner can exercise the option by investing the strike price defined by the option. A call option gives the right to acquire an asset of uncertain future value for the strike price.

ArchOptions [Bahsoon and Emmerich, 2003a; Bahsoon, 2003; Bahsoon and Emmerich, 2004b] values the *growth* options of an architecture relative to some future changes, as a way for understanding the architectural flexibility and its stability implications. A growth option is a real option to expand with strategic importance [Myers, 1987; Schwartz and Trigeorgis, 2000]. Growth options are common in all infrastructure-based (as it is the case with software architectures) or strategic industries with multiple-product generations or applications [Myers, 1987; Schwartz and Trigeorgis, 2000]. In the architectural context, growth options are linked to the flexibility of the architecture to respond to future changes. Since the future changes are generally unanticipated, the value of the growth options lies in the enhanced flexibility of the architecture to cope with uncertainty; otherwise, the change may be too expensive to pursue and opportunities may be lost.

Let us assume that the value of the system is V. As the software evolves, a change in future requirement i_i is assumed to "buy" $x_i\%$ of the "architectural potential" taking the form of embedded flexibility, paying C_{ei}, where C_{ei} corresponds to an estimate of the likely cost to accommodate the change on the given architecture of the software system. This is analogous to a *call option* to buy (x_i%) of the base project, paying C_{ei} as exercise price. The call options financial/real and their corresponding ArchOptions analogy is depicted in table 1 and detailed in [Bahsoon and Emmerich, 2003a].

Table 1. Financial/real options/ArchOptions analogy

Option on stock	Real option on a project	ArchOptions
Stock Price	Value of the expected cash flows	value of the "architectural potential" relative to the change (x_iV)
Exercise Price	Investment cost	Estimate of the likely cost to accommodate the change (C_{ei})
Time-to-expiration	Time until opportunity disappears	Time indicating the decision to implement the change (t)
Volatility	Uncertainty of the project value	"Fluctuation" in the return of value of V over a specified period of time (σ)
Risk-free interest rate	Risk-free interest rate	Interest rate relative to budget and schedule (r)

We view the investment opportunity in the system as a base investment plus call options on the future opportunities, where a future opportunity corresponds to the investment to accommodate some future requirement(s). The payoff of the constructed call option gives an indication of how valuable the flexibility of an architecture is to endure some likely changes in requirements. The value of the architecture, is expressed in *(1)* accounting for V and both the expected value and exercise cost to accommodate future requirements i_i, for $i \leq n$. Valuing the expectation E of expression *(1)* uses the assumptions of [Black and Sholes, 1973] and detailed in [Bahsoon and Emmerich, 2003a]. We assume that the interest rate, r, is zero for the simplicity of exposition.

$$V + \sum_{i=0}^{n} E\,[\max\,(x_i V - C_{ei}, 0)] \qquad (1)$$

It is worth noting that previous applications of ArchOptions include (i) valuing the resulted architectural flexibility and its stability implications due to investing in a refactoring exercise [Bahsoon and Emmerich, 2004a], and (ii) evaluating software architectures for stability to understand the success (failure) of the software system's evolution, in response to likely changes in requirements [Bahsoon and Emmerich, 2004b].

The model has the prospect of valuing the architectural flexibility and its value potentials due to various types of changes in requirements. These could be functional or non-functional. However, the changes in non-functional requirements are more critical and revealing for understanding architectural stability problem. As the middleware realizes much of the non-functionalities, analyzing for architectural stability in the face of changes in non-functional requirements can't be done in isolation of the middleware induced. We tailor the ArchOptions model to value the growth potentials of the middleware-induced software architectures to respond to changes in non-functional requirements.

As we have noted in [Bahsoon and Emmerich, 2003a; Bahsoon 2003; Bahsoon and Emmerich 2004a; Bahsoon and Emmerich 2004b], the search for a potentially stable architecture requires finding an architecture that maximizes the yield in the added value, relative to some future changes in requirements. As we are assuming that the added value is attributed to flexibility, the problem becomes maximizing the yield in the embedded or adapted flexibility in a software architecture relative to these changes. Given the choice of two or more middleware candidates, the selection has to maximize the yield in the embedded flexibility, relatives to likely changes in non-functional requirements.

Choosing a particular middleware to induce the architecture of the software system can be seen as an investment to purchase flexibility in the software architecture-induced. The middleware simplifies the construction of a distributed system architecture by offering higher level programming abstractions that shield application developers from distribution complexities, thereby letting them concentrate on the application instead of implementing the non-functionalities and managing system resources. The choice is influenced by the non-functional requirements from one side and the "architectural potential" of the middleware to respond to future changes in these requirements. In this context, deciding on a particular middleware to induce the software system architecture can be seen as an investment to purchase future growth options that enhance the upside potentials of the structure, paying an upfront cost I_e, which corresponds to the cost of developing the architecture by the given middleware. We extend ArchOptions to value the worthwhile of the investment, given in *(2):*

$$V - I_e + \sum_{i=0}^{n} E\,[\max\,(x_i V - C_{ei}, 0)] \qquad (2)$$

Let us assume that we are given the choice of two middleware M_0 and M_1 to induce the architecture of a particular system. Let us assume that S_0, S_1 are the architectures obtained from inducing M_0 and M_1 respectively. Say, inducing M_1 is an economical choice, if it adds value to S_1 relative to S_0. We attribute the added value to the enhanced flexibility of S_1 over S_0. If we are considering stability as a criteria for understanding the value added on the system, then future changes in non-functional requirements will tell us how valuable S_1 is relative to S_0, as we are performing a tradeoff between the architecture induced by M_0 and M_1. But the added value is uncertain, as the demand and the nature of the future changes are uncertain. Hence, using option theory is a promising approach to inform the selection.

The selection has to be guided by the expected payoff in $(- I_e + \sum_{i=1\ldots n} E\,[\max\,(x_i V - C_{ei}, 0])_{S1}$ relative to that of S_0. That is, if $(- I_e + \sum_{i=1\ldots n} E\,[\max\,(x_i V - C_{ei}, 0)]_{S1} > \sum_{i=1\ldots n} E\,[\max\,(x_i V - C_{ei}, 0)]_{S0})$ for some likely changes, then it is worth investing in M_1, as the investment in M_1 is likely to generate more growth options for S_1 than for S_0. We appeal to the use of future savings in maintenance effort as a way to quantify the value added due to a selection. If we assume that $x_i V_{S1}$ is the expected savings in S_1over S_0 due to selection, it is reasonable to consider that if $(- I_e + \sum_{i=1\ldots n} E\,[\max\,(x_i V - C_{ei}, 0)]_{S1} >= 0)$, then investing in M_1 is said to payoff. An optimal payoff could be when the option value (i.e., $\sum_{i=1\ldots n} E\,[\max\,(x_i V - C_{ei}, 0)]$) approaches the maximum relative to some changes in non-functional requirements, indicating an optimal payoff of the selection, provided that $(- I_e + \sum_{i=1\ldots n} E\,[\max\,(x_i V - C_{ei}, 0)]_{S1} >= 0)$. We use sensitivity analysis to manipulate the model variables and analyze when such a situation is likely to occur.

Let us focus our attention on the payoff of the call options, as they are revealing for the flexibility of the architecture induced in responding to the likely future changes in requirements.

For a likely change in requirement *k*,

Call option in-the-money: if (E [max (x_kV - C_{ek}, 0)])$_{S1}$ >0, then the flexibility of S_1 is likely to payoff, relative to S_0, as the flexibility of the architecture in response to the change is likely to add a value, if the change need to be exercised. This means that inducing the architecture with M_1 has more promise than M_0, as the flexibility of S_1 in responding to the likely change is more valuable for S_1 than for S_0.

Call option out-of-the-money: if (E [max (x_kV - C_{ek}, 0)])$_{S1}$=0, then M_1 is not likely to payoff, relative to M_0, as the flexibility of the architecture in response to the change is not likely to add a value for S_1 if the change need to be exercised. Two interpretations might be possible: (i) the architecture is overly flexible in the sense that its response to the change(s) has not "pulled" the options. This implies that the embedded flexibility (or the resources invested in implementing flexibility- if any) are wasted and unutilized to reveal the options relative to the changes. In other words, the degree of flexibility provided is much more than the flexibility demanded for the change. This case has the prospect in providing an insight on how much do we need to invest in the adapted flexibility relative to the likely future changes, while not sacrificing much of the resources; (ii) the other case is when the architecture is inflexible relative to the change. This is when the cost of accommodating the change on S_1 is much more than the cumulative expected value of the architecture responsiveness to the change.

The options model *(2)* requires the estimation of several parameters. Most important are x_iV, I_e, and C_{ei}.

Estimating C_{ei}, I_e. Estimating cost is a well-established component in software engineering; it is outside the scope of our work. As a result of inducing the architecture with the middleware, it is feasible to use existing metrics to cost estimation (e.g. COCOMO-II [Boehm et al., 1995]). This is due to the fact that a considerable part of the distributed applications implementation is already available, when the architecture is defined, for example, during the Elaboration phase of the Unified Process. Another approach is to build on architectural level dependency analysis (e.g., [Stafford and Wolf, 2001]) research to extract cost estimates of accommodating i_i, guided by some structural criteria.

Capturing and estimating x_iV. The application of Black and Scholes [1973] assumes that the stock option is a function of the stochastic variables underlying stock's price and time. We assume that V moves stochastically bounded to two extreme values: optimistic and pessimistic. This assumption appears to be plausible: (i) it tends to account for all possible values within the bound, yielding to a better approximation when opposed to an ad-hoc type of estimation; (ii) the value of an (evolvable) system changes over time; it tends to change in uncertain way due to changes in requirements.

Black and Scholes is an *arbitrage-based* technique. The technique requires knowledge of the value of the asset in question in span of the market. Software architectures, however, are (non-traded) real assets. Real options may be valued similarly to financial options, though they are not traded [Schwartz and Trigeorgis, 2000]. Real options valuation based on arbitrage-based pricing techniques determines the value of an asset in question in span of the market value using a correlated *twin asset* [Schwartz and Trigeorgis, 2000]. The twin asset is an asset that has the same risks the asset in question will have when the investment has been completed [Schwartz and Trigeorgis, 2000]. In financial options, several proxies are available to predict the value of the financial asset - the most obvious proxy is simply the historical values of the asset. In real options, such proxies rarely exist and the analyst may need to rely on experience and judgment in his/her estimations [Schwartz and Trigeorgis, 2000]. Real options valuation (based on arbitrage) focuses on market value and uses the *rate of return* on the twin asset as an input to the valuation of the asset in question. If the asset value is not *directly observable*, it is reasonable to use estimates of the revenues on the asset to estimate the market value [Schwartz and Trigeorgis, 2000]. For example, some aspects of the architectural responsiveness to the change can be justified in terms of the directly observable cash flows linked to future operational benefits or the market- making it easy to use the rate of return to value the options. However, many others aspects may not be directly observable through cash flows. Yet, their contribution to the added value is crucial. If the analyst(s) relies on experience and judgment in his/her estimation, the estimates tend to be subjective but could make an implicit use of market information. However, back-of-the-envelope calculations, which are based on value estimates (rather than on market value) are yet revealing [Sullivan et al., 2001]. We note that it remains an open challenge to strongly justify precise estimates for real options in software [Sullivan et al., 1999]. As a compromise, estimating x_iV requires a comprehensive solution that is flexible enough to incorporate multiple valuation techniques; some with subjective estimates and others based on market data, when available. The problem of how to guide the valuation and introduce discipline in this setting, we term as the *multiple perspectives valuation problem*. As the added value may be relative to the market and/or the enterprise, the solution may be through a valuation framework that captures the added value - of the "architectural potential" of the change- from different perspectives. The purpose is to reach a comprehensive value of options from the different perspectives. Also, the aim is to promote flexibility through incorporating both subjective estimates (may implicitly use market information) and/or explicit market value (when available). As the architecture is the artefact that facilitates both technical and market reasoning, such an approach seems to be viable. Addressing this problem and its solution is outside the scope of this paper.

Sensitivity analysis. Statistical questions on how the uncertainty of the input parameters propagates to the model output often require sensitivity analysis. The objective is to provide an understanding of how the model response variables respond to changes, as the model's underlying assumptions or its parameters change. For example, the estimated parameters may be subject to uncertainty: the valuation could have overestimated or underestimated the value of the parameters. Further, the estimated value may be liable to further adjustment to reflect the time value. We support the model with sensitivity analysis to increase the confidence in the model predictions and to provide a basis for "what-if" analyses.

First derivative analysis is much used in the investment arena for analyzing the sensitivity of the value of a financial option to changes in the variables. *Delta* and *Vega* provide the investment analyst with a ready means to discover financial option's sensitivity to changes in the estimated value of the underlying asset; and increases and decreases to the volatility of the underlying asset. Table 2 provides a summary of the sensitivity parameters, their financial explanation, mathematical formulation and the corresponding ArchOptions analogy.

Table 2. Sensitivity parameters and ArchOptions

Parameter	Financial Explanation	ArchOptions Analogy	Math-formula
Delta (Δ)	Option price rate of change w.r.t. the underlying asset (%)	Option value rate of change w.r.t. $x_i V$	$\frac{\partial c}{\partial (x_i V)}$
Vega (ν)	Option price rate of change w.r.t. the volatility of the underlying asset (%)	Option price rate of change w.r.t. σ (%)	$\frac{\partial c}{\partial \sigma}$

The *Delta (Δ)* of an option is defined as the rate of change of the option price with respect to the underlying asset. Suppose that the delta of a call option is 0.6. This means that when the underlying asset price changes by a small amount, the option price change by about 60% of that amount. Mathematically, delta is the partial derivative of the call price with respect to the underlying asset price given by $\Delta = \partial C/\partial S$. In practice, volatilities may change over time. This means that the value of the option is liable to change because of the movement in volatility as well as because of changes in the asset price and the passage of time. The *Vega (ν)* of an option is the rate of change of the value of the option with respect to the volatility of the underlying asset. If Vega is high, the option value is very sensitive to small changes in volatility. If Vega is low, volatility changes have relatively little impact on the value of the option.

In the following section, we empirically evaluate the theory, exercise the model, and verify its interpretations. We use a case study that adequately represents a medium-size component-based distributed architecture. We report on how a likely future change in scalability could impact the architectural structure of two versions, each induced with a distinct middleware: one with CORBA and the other with J2EE. We calculate the options on each structure and draw some observations.

3. Case Study

We use Duke's Bank application, an online banking application provided by Sun [Sun Microsystems Inc., http://java.sun.com], 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 distinct middleware: one with CORBA and the other with J2EE. We report on 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. The objective is to study how middleware-induced software architectures may differ in coping with changes in non-functional requirements. 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 middleware. 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 [Emmerich, 2000b]. 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.

Architecturally, the Duke's Bank 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, managing accounts, and managing transactions. The clients access the customer, account, and transaction information maintained in a database. The architecture of the Duke's Bank application is given in Figure 1. 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 CORBA version of the Duke's Bank is a straightforward implementation of the above description. In the J2EE, the application consists of six EJB (Enterprise Java Beans) components that handle operations issued by the users of a hypothetic bank. The six components can be associated with classes of operations that are related to bank accounts, customers and transactions, respectively.

For each of these classes of operations, a pair of session bean and entity bean is provided. Session beans are responsible for the interface towards the users and the entity beans handle the mapping of stateful components to underlying database table. The EJBs that constitute the business components are deployed in a single container within the application server, which is part of the middleware.

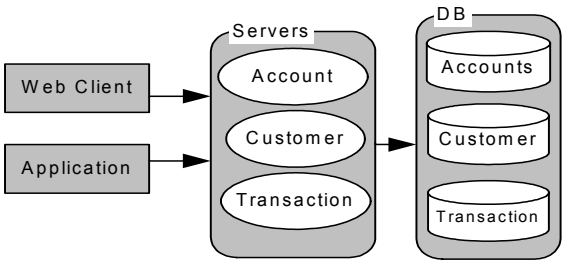


Figure 1. The Architecture of the Duke's Bank

For the J2EE version, we use JBoss application server [http://www.jboss.org], an open source. In one of the experiments, we use WebLogic server [http://www.bea.com/] with an average upfront payable license cost equal to $25000/host. We use JacORB, version 2.0 to implement the CORBA version. JacORB, is a CORBA implementation written in Java; it allows the communication of Java objects. Our choice of JacORB makes the comparison between the two versions feasible and meaningful, as both will be implemented in JAVA.

We assume that the Duke's Bank system needs to scale up to accommodate the growing number of clients. We consider scalability as a goal that needs to be achieved by the architecture of the software system to be induced. We adopt a goal-oriented approach to refining requirements (e.g., [Dardenne et al., 1993; Anton, 1996]). We refine the goal, using guidance on how it could be operationalised 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, such as [Object Management Group, 1999-2000; Sun Microsystems Inc., 2002]; related design and implementation experience, mainly that of [Othman et al., 2001a; Othman et al., 2001b]. 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 [Object Management Group, 1999-2001] 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. Hence, the refinement and its corresponding operationalisation are guided by the solution domain (i.e., the middleware). Refinement of the scalability goal is depicted in Figure 2. Detailing the refinements and the operationalisation of the goal is given in sections 3.1 and 3.2.

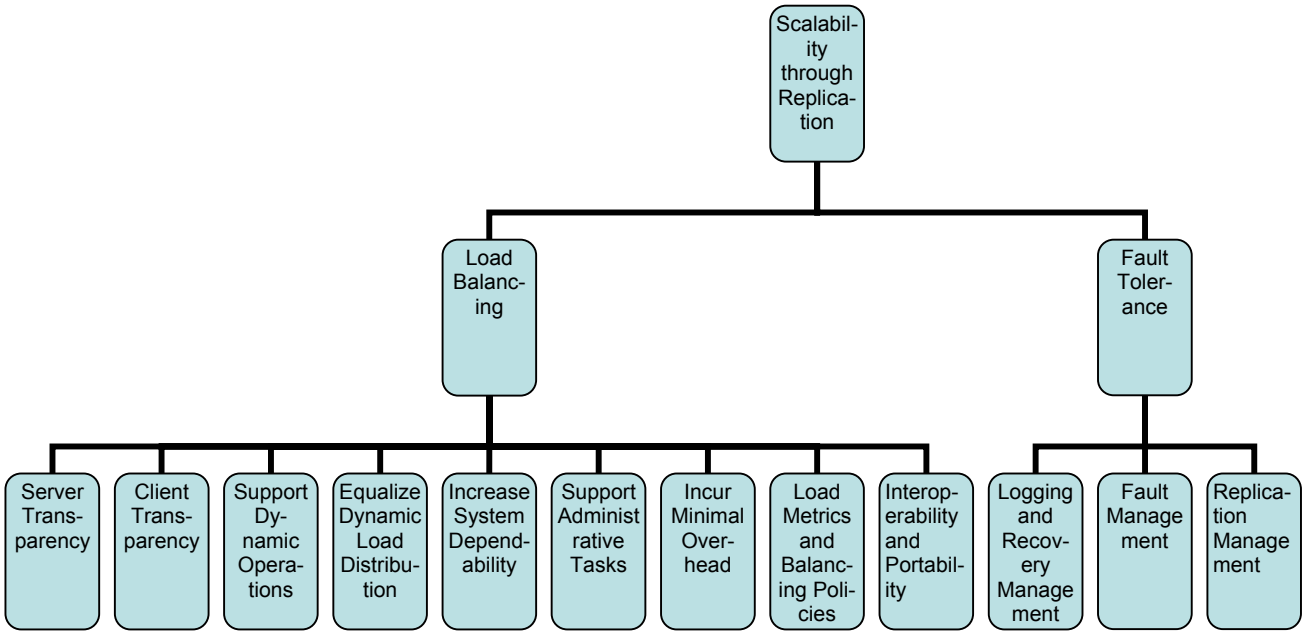


Figure 2. The Goal-oriented (high-level) refinement for achieving scalability through replication

In subsequent sections, we investigate how scalability could be achieved on both versions. We analyze the impact of the change by looking at the structural changes and the source lines of code (SLOC) that need to be modified/added for implementing the change, configuring, and deploying the software system following the change. We apply the model to understand the value added by inducing the architecture by EJB relative to CORBA, if the change needs to be applied. We use future savings in maintenance cost (if any), as a way to quantify the value added. We draw some observations and report on some preliminary conclusions.

3.1 Scaling the CORBA-Induced Architecture

In this subsection, we investigate how scalability could be achieved in the CORBA-induced version through replication mechanisms.

CORBA's object model [Object Management Group, 2000] relies to a large degree on the semantics of *object references*. An object reference uniquely identifies a local or remote object instance- clients can only invoke an operation on an object if they hold a reference to the object. Managing scalability in CORBA, through replication, is not straightforward, for object referencing makes it demanding. If several replicas of a server object are available, providing an object reference to the client is uneasy task. Hence, a CORBA implementation to the management of scalability, through replication, has to incorporate the following: (i) Replication management (i.e., create, remove, manage objects state in case of state retention, etc); (ii) balancing load among replicas (i.e., when a client invokes a request, it needs to get the object reference of the least loaded replica) and (iii) a fault tolerance (i.e., when a server object fails to handle a request, the request has to be forwarded to a replica).

The Object Management Group's CORBA specification defines a fault tolerance support, which provides replication management. The specification also provides the core capabilities needed to support load balancing. Othman et al. [2001] introduces a CORBA load-balancing service, designed on TAO- the ACE (Adaptive Communication Environment) ORB [Schmidt et al., 1998]. The TAO-ORB is a CORBA-compliant ORB that supports applications with stringent Quality of Service (QoS) requirements. The designed CORBA load-balancing service takes advantages of the request forwarding mechanism the CORBA specification mandates [Object Management Group, 1999]. A CORBA server application can use this mechanism to forward client requests to other servers transparently, portably, and interoperably. The combination of the CORBA fault tolerance support and Othman's CORBA load-balancing service provides a strong example of implementing scalability in CORBA. We use both the Object Management Group's CORBA specification and the TAO's design and implementation of the services as guidelines for understanding the structural impact of the change on the Duke's Bank architecture and the corresponding effort/cost required to scale the system.

Subsection 3.1.1 describes the requirements and the architecture for implementing fault-tolerance in CORBA, based on the OMG specification [Object Management Group, 1999]. Subsection 3.2.2 describes the requirements and the architecture for implementing the load-balancing support in CORBA, based on [Othman et al., 2001a; Othman et al., 2001b]. Subsection 3.2.3 analyzes the structural impact, when the fault-tolerance and the load-balancing services need to be implemented to scale the CORBA-induced Duke's Bank architecture.

3.1.1 Achieving fault tolerance support and replication management

The Fault Tolerant CORBA standard aims to provide robust support for applications that require a high level of reliability, beyond the level provided by single backup server. According to the CORBA specification, fault tolerance depends on entity redundancy (replication of objects), fault detection, and recovery. To render an object fault-tolerant, several replicas of the object are created and managed as an *object group*. While each individual replica of an object has its own object reference, an additional *interoperable object group reference* (IOGR) is introduced for the object group as a whole. It is the object group reference that the replicated server publishes for use by the client objects. The client objects invoke methods on the server object group, and the members of the server object group execute the methods and return their responses to the clients, just like a conventional object. Because of the object group abstraction, the client objects are not aware that the server objects are replicated (*replication transparency*) and are not aware of faults in the server replicas or of recovery from faults (*failure transparency*).

The standard provides support for fault detection, notification, and analysis for the object replicas. The standard also supports a range of fault tolerance strategies, including automatic check pointing; logging and recovery from faults; request retry; redirection to an alternative server; passive (primary/backup) replication and active replication, which provides more rapid recovery from faults. The standard aims for minimal modifications to the application programs, and for transparency to both replication and faults.

3.1.2 The fault tolerance architecture

The requirements for implementing Fault Tolerance in CORBA are detailed in the CORBA fault tolerance specification of the OMG. Figure 3 presents an architectural strategy that realizes these requirements and fully documented in [Object Management Group, 1999]. Other architectural strategies for realizing these requirements are possible.

The basic blocks of the architecture are three: *Replication management; Fault Management; and Logging and Recovery Management.*

Replication Management. The Replication Manager inherits three application program interfaces: the *PropertyManager*, *ObjectGroupManage*r, and the *GenericFactory*. The *PropertyManager* provides operations that set properties for object groups. The *ObjectGroupManager* provides operations that allow an application to exercise control over addition, removal, and locations of members of an object group. It also provides operations for obtaining the current reference and identifier for an object group. The *GenericFactory* issues requests for replicating objects (object groups), creating replicas (members of object groups), and unreplicating objects.

Fault Management. The following components are responsible for managing faults in the proposed fault tolerant architecture. These are *fault detection*, *fault notification*, and *fault analysis*. The Fault detection component detects the presence of a fault in the system and generates a fault report. The fault notification component propagates fault reports to entities that have registered for such notifications. The fault analysis component analyses a (potentially large) number of related fault reports to generate a condensed diagnosed report.

Logging and Recovery Management. The *Logging Mechanism* records the state and actions of a member of an object group in a log. The *Recovery Mechanism* sets the state of a member, either after a fault when a backup member of an object group is promoted to the primary member, or alternatively when a new member is introduced into an object group.

Components of the Fault Tolerance Infrastructure are shown on the top of figure 3. These include *Replication Manager*, *Fault Notifier*, and *Fault Detector*. The bottom of figure 3 shows three hosts: H_1, H_2, and H_3. The client application object C on H_1 invokes a replicated server object with two replicas S_1 on host H_2, and S_2 on host H_3. The figure shows Factory and Fault Detector objects that may be present and specific for a host. The service objects are replicated objects. The host-specific objects, however, are not replicated. The figure also shows the Message Handler and the Logging and Recovery Mechanisms that are present on each host. Logically, a single instance of the Replication Manager and Fault Notifier shall exist in each fault tolerance domain. Physically, however, they are replicated to protect against faults, as any other application object are. The architecture defines minimal modifications to the existing ORBs. These modifications allow non-replicated clients to derive fault tolerance benefits upon invoking replicated server objects.

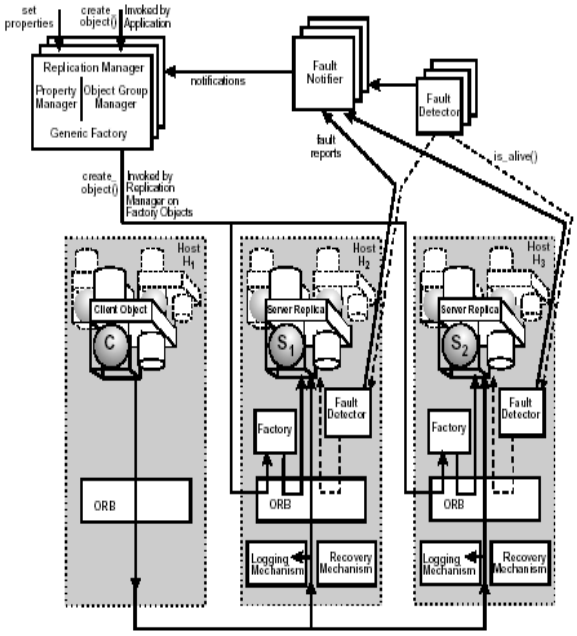


Figure 3. The CORBA fault-tolerance architecture [Object Management Group, 1999].

3.1.3 Achieving load balancing

Load balancing helps improve system scalability by ensuring that client application requests are distributed and processed equitably across a group of servers. Likewise, it helps improve system dependability by adapting dynamically to system configuration changes that arise from hardware or software failures. According to [Othman et al., 2001a], the design of an effective CORBA load balancing service should be based on the following requirements. The interested reader may refer to [Othman et al., 2001a] for further details.

Enable client application transparency. A CORBA load balancing service should be as transparent as possible to clients and servers and should require no changes to clients whose requests it balances.

Enable server application transparency. Implementing a server object's servant (a programming language entity that implements object functionality in a server application) should require no changes to support load balancing. Yet changes to the server application might still be required under certain conditions.

Support dynamic client operation request patterns. The CORBA load balancer, however, shall focus on load balancing techniques that do not require a priori scheduling information, where client operation request patterns are dynamic and the duration of each request might not be known in advance, which is the case of the Duke's Bank.

Maximize scalability and equalize dynamic load distribution. CORBA load balancing service must enhance system scalability by maximizing dynamic resource utiliza-

tion in a group of servers that otherwise would be underutilized.

Increase system dependability. Load balancer should provide mechanisms to handle failures efficiently when detected by administrators or other system components. For example, the load balancer should migrate crashed or failing servers to other servers until the failure is resolved. However, there is still a need for a fault-tolerance support, which we described in previous section based on the [Object Management Group, 1999].

Support administrative tasks. A good CORBA load balancing service should have facilities for dynamic addition/removal/upgrading of new replicas and should adjust to the new load conditions rapidly, without disrupting or suspending service for existing clients.

Incur minimal overhead. A CORBA load balancing service should not introduce undue latency or networking, which may reduce the overall system performance.

Support application-defined load metrics and balancing policies. A CORBA load balancing service should let applications specify the semantics of metrics used to measure load, such as CPU, I/O resources, communication bandwidth, or memory load.

Rely on CORBA interoperability and portability. A CORBA load balancer should not restrict the application developers to single ORB providers.

3.1.4 The load balancing architecture

Othman et al. [2001b] suggest a CORBA adaptive balancing built on TAO to realize the above stated requirements. The TAO's load balancing solution is entirely based on standard features in CORBA, without requiring severe extensions to the ORB or its communication protocols. The suggested load balancing solution is based on the patterns [Schmidt et. al., 2000] of the CORBA component model (CCM) [BEA Systems, 1999] for minimizing the changes on the application layer. In particular, the following patterns are utilized to achieve the above stated transparency requirements: these are the Portable Interceptors pattern, Component Configuration pattern, Component Configurator pattern, and the Asynchronous Completion Token pattern [Schmidt et. al., 2000].

Figure 4 illustrates the components in TAO's load balancing service. The design supports adaptive load balancing and on-demand request forwarding [Othman et al. 2001b] and outlined below:

The *Replica Locator* identifies which of the replicas will be assigned a request. The *Replica Locator* component forwards the requests to the *Load Analyzer* component. The *Load Analyzer* component analyses the requests; it select the replica to be assigned the request. The *Replica Locator* obtains a reference to a replica from the load analyzer and then forwards the request to that replica. The *Replica Locator* binds clients to the identified replicas. The *Load Analyzer* also allows explicit selection of a load balancing strategy at runtime, while maintaining a simple and flexible design. The replica locator is portably implemented using servant locators implementing the interceptor pattern [Schmidt et. al., 2000], abiding to standard CORBA portable object adapter mechanisms [Henning and Vinoski, 1999]. The *Load Balancer* component is a mediator that integrates all the components. It provides an interface for load balancing without exposing clients to the intricate interactions between the components it integrates. The *Load Monitor* component monitors loads on a given replica, reports replica load to a *Load Balancer*, and informs replicas when they should accept requests versus forward them back to the load balancer. Each object that TAO's load balancing service manages communicates with it through a unique proxy. The load balancer uses the *replica proxies* components to distinguish different replicas to workaround CORBA's so-called "weak" notion of object identity [Object Management Group, 1999], where two references to the same object might have different values.

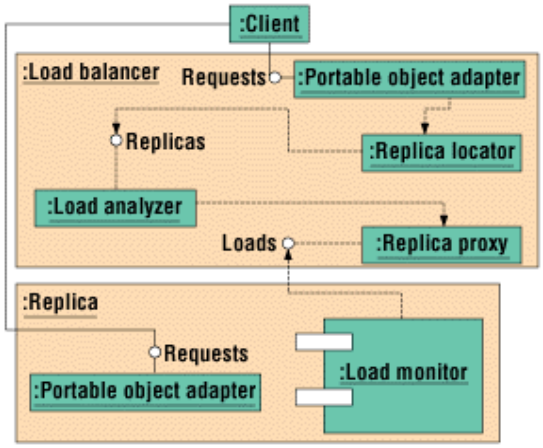


Figure 4. TAO Load Balancing [Othman et al., 2001b]

3.1.5 Change impact analysis

The combination of the CORBA fault tolerance support and Othman's CORBA load-balancing service provides a strong example on how scalability could be achieved in the CORBA-induced architectures of the Duke's Bank. In this section, we analyze the impact of the change by looking at the structural changes and the source lines of code (SLOC) that need to be modified/added for implementing the change, configuring, and deploying the software system. We use the design and the implementation of both services (i.e., fault tolerance and load balancing) on TAO as a guide to estimate the design impact and the effort required to realize the scalability requirements in our given architecture. The TAO design of these services is based on the CORBA specification. We note that the TAO's implementation of both services is in C++. We list all the JAVA classes and files necessary to build the equivalent JAVA implementation of both services.

Considering the CORBA-induced architecture of the Duke's Bank, supporting scalability through replication does not leave the middleware infrastructure and the application layer intact. Though the use of both CORBA specification and design patterns, has simplified the task of realizing the requirements for achieving fault tolerance and load balancing, implementation and integration overhead have not been abandoned. In particular, the fault tolerance and load balancing services need to be implemented. The implementation needs to be integrated into the used middleware. The server application needs to be updated, so that it will be able to support object group, described in section 3.1.1 and section 3.1.2. The client itself has to undergo slight changes.

In particular, to support load balancing, the middleware and the application need to be modified. The modifications include the implementation of the Load Balancing Service and integrating the service into the existing middleware infrastructure. The server-side application, the main CORBA services (mainly, the naming service and the transaction Service), and the client-side needs to be updated. In particular, the binding mechanism needs to be modified to support the introduction of the object groups. The server application, which initially binds instances of server implementation to the naming service, has to be changed. Instead, the client's requests need to be bound to the replica the load balancer selects. Hence, this requires modifications to the standard CORBA services through introducing ad hoc proprietary protocols and interface that abides to the OMG standards. In an environment where several hosts are used to store the server objects, different object groups need to be created. The server application needs to be modified to populate servant instances. Four interfaces need to be implemented, describe in the IDL. These are Strategy, LoadAlert, LoadMonitor and LoadManager. ORB interceptors and initializers have to be implemented.

A List of classes and files necessary to implement the fault tolerant service into the Duke's Bank architecture is depicted in table 4. Table 5 reports on the effort necessary to develop and integrate the load balancing service into the middleware. Table 3 provides an aggregated summary of the over SLOC that need to be implemented.

On the client side, the client application needs to be modified to look up the load balancer instead of the naming service to get a replica object reference. The load balancer will be then able to send an object reference by using the CORBA ForwardRequest exception that the client can catch. Thirty lines of codes are estimated to update the client. To configure, all the instances of JacORB over the different hosts have to be shutdown, which be unappreciated for such a type of application. To compile and package the developed services, an Ant script has to be updated for each service. This introduces additional 200 lines of code. The main Ant script, which executes all the other Ant scripts, has to be updated introducing and additional six lines of codes/host. The properties file (i.e., jacorb.properties) has to be updated on each host requiring seven SLOC/host. These updates concern the ORBInitRef property and the interceptors ORBInitializer. All the JacORB instances then need to be restarted.

Table 3. Aggregate results

Task	SLOC
Fault Tolerant implementation	5117
Load Balancing implementation	3943
Server-side application (Server objects Implementation and Server application- on each host)	170
Client-side application	30
Configuration on each host	Stop/restart, 200 SLOC+ 13/host

Table 4. Implementing the fault tolerance service

File Name	File Type	SLOC	Description
CosFaultTolerance	IDL	242	Interface description of remote methods
PropertyManagerImpl	Java	273	Implementation of the PropertyManager interface
ObjectGroupManagerImpl	Java	672	Implementation of the ObjectGroupManager interface
GenericFactoryImpl	Java	523	Implementation of the GenericFactory interface
ReplicationManagerImpl	Java	865	Implementation of the ReplicationManager interface
FaultNotifier	Java	611	Implementation of the FaultNotifier interface
ClientPolicy	Java	155	Implementations of the RequestDurationPolicy interface
ServerPolicy	Java	61	Implementation of the HeartbeatEnabledPolicy
FTPolicy	Java	207	Implementation of the HeartbeatPolicy interface
FaultDetector	Java	149	Class defining the component illustrated above
DefaultFaultAnalyzer	Java	113	The default fault analyzer
ReplicationManagerFaultAnalyzer	Java	865	Replication Manager's fault analyzer
FaultConsumer	Java	200	Connect to the fault notifier
PropertyValidator	Java	29	Class providing static methods to validate properties
MemberInfo	Java	50	Structure that contains all member-specific information
PropertyUtils	Java	53	Provides some methods used to manipulate properties
Operators	Java	23	Class providing static methods related to operators
ReplicationManagerServer	Java	13	Class running the Replication Manager server
FaultNotifierServer	Java	13	Class running the Fault Notifier server
Total		5117	

Table 5. Implementing the load balancing service

File Name	File Type	SLOC	Description
CosLoadBalancing	IDL	90	Interface description of remote methods
LoadAlertImpl	Java	26	Implementation of LoadAlert interface.
LoadCPUMonitorImpl	Java	138	LoadMonitor implementation that monitors the overall CPU load on a given host
LoadManagerImpl	Java	919	Implementation of LoadManager interface
LeastLoaded	Java	405	Implementations of Strategy interface
LoadAverage	Java	305	Implementations of Strategy interface
LoadMinimum	Java	389	Implementations of Strategy interface
RoundRobin	Java	121	Implementations of Strategy interface
Random	Java	128	Implementations of Strategy interface
MemberLocator	Java	59	Class which defines the component described above
LoadAlertHandler	Java	40	This class handles all asynchronously received replies from all registered LoadAlert objects. It only exists to receive asynchronously sent exceptions
LoadAlertInfo	Java	30	Structure that contains all LoadAlert-specific information
LoadAlertMap	Java	60	Maps a LoadAlertInfo with a location
LoadListMap	Java	60	Maps a LoadList with a location
LoadMap	Java	60	Maps a load with a location
MonitorMap	Java	60	Maps a LoadMonitor with a location
PullHandler	Java	58	Event handler used when the "pull" monitoring style configured
PushHandler	Java	39	Event handler used when the "push" monitoring style is configured
LB_ServerRequestInterceptor	Java	109	Responsible for redirecting the requests back to the manager
LB_ORBInitializer	Java	72	*Creates and registers with the ORB the* LB_IORInterceptor and LB_ServerRequestInterceptor
LB_ClientRequestInterceptor	Java	62	Handles transparent object group member registration with the LoadManager, and registration of the LoadAlert object necessary for load shedding
LB_ClientORBInitializer	Java	33	Creates and registers with the ORB the LB_ClientRequestInterceptor
LoadManagerServer	Java	214	Class running the load balancer server
LoadMonitorServer	Java	315	Class running the load monitor server
Total		3943	

3.2 Scaling the J2EE-Induced Architecture

In subsequent sections, we investigate how scalability could be achieved in the J2EE–induced version through replication mechanisms. We analyze the impact of the scalability change on the J2EE-induced architecture of the Duke's Bank.

3.2.1 Scalability in J2EE through Replication

Figure 5 depicts a common J2EE [Sun Microsystems Inc., 2002] cluster architecture. Clustering enables a group of (typically loosely coupled) servers to operate logically as a single server. The advantages of clustering include the elimination of a single point of failure; the high service availability if multiple servers in the cluster can handle the same service; and load balancing by diverting requests to the least loaded server hosting the same service. We use JBoss 3.0[http://www.jboss.org/], an open source J2EE application server. JBoss clustering aims at improving scalability and high availability using replication techniques. JBoss relies on *Jgroups* [http://www.jgroups.org/] for the clustering of its naming registry face- Java Naming and Directory Inter (JNDI)-and its EJB container. JGroups is an open source group communication middleware fully written in Java. JGroups provides the following main features: group creation and deletion, where group members can be spread across LANs or WANs; joining and leaving of groups; membership detection and notification including joined/left/crashed members; detection and removal of crashed members; sending and receiving of member-to-group messages (point-to-multipoint); and Sending and receiving of member-to-member messages (point-to-point).

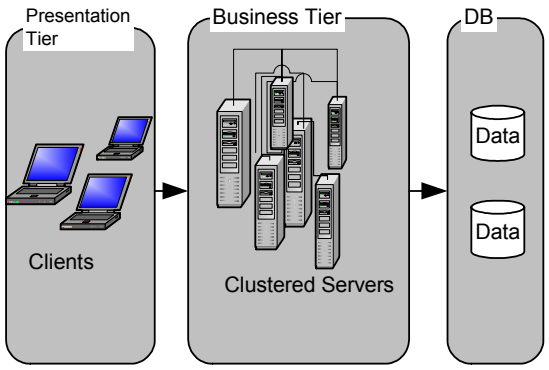
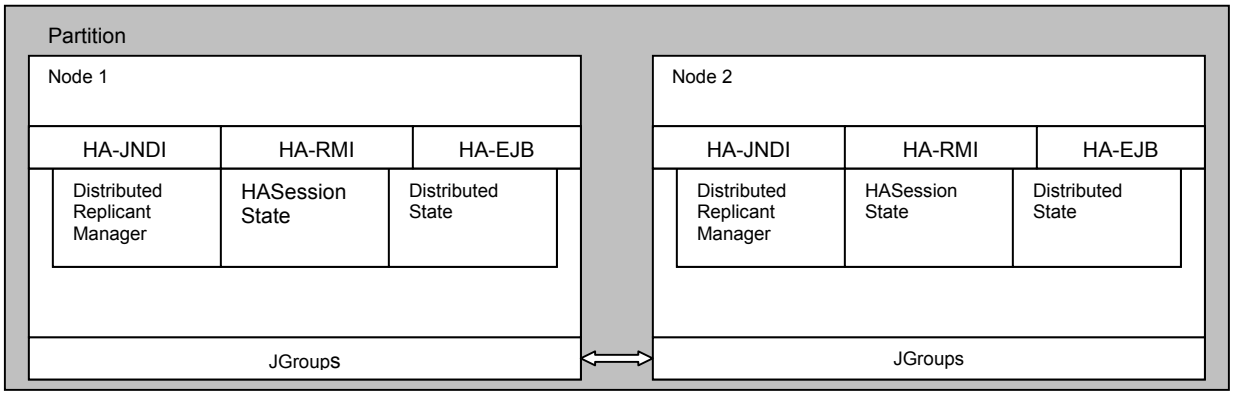


Figure 5. Example of J2EE cluster architecture

JBoss uses a layered architecture to manage clustering. The architecture relies on JGroups for clustering, which is abstracted. Figure 6 describes the architecture using two nodes. The term partition is used to refer to a cluster. A node can be part of several partitions.

Figure 6. Clustering Architecture



The HAPartition (i.e., High Availability Partition) abstracts the communication framework; it provides access to a set of communication primitives. Services need to register with the HAPartition to use the HAPartition services. The *Distributed Replicant Manager* manages the replicas by providing methods to add or remove replicas from a partition. The *HASessionState* is used to manage the state of Stateful Session Beans. The state of all Stateful Session Beans are replicated and synchronised across the cluster each time the state of a bean changes. The *Distributed State* stores settings or parameters that should be used by the containers in the cluster. Clients can use either the local JNDI service or the HA-JNDI service to look up objects. If the local JNDI service is used, the local JNDI namespace is used to locate objects. HA-JNDI delegates the lookup to the local JNDI, if it fails to find the object within global the cluster-wide context. EJB homes are bound to the local JNDI of the server on which the particular EJB is deployed.

HA-RMI provides load-balancing and fail-over facilities for RMI servers. HA-EJB allows selecting the load-

balancing policy to apply (e.g., Round Robin, First Available), when deciding on a replica that will respond to the client request. The load-balancing policy is not adaptive.
JBoss provides clustering for the two main types of EJB: Entity Bean and Session Bean (Stateful and Stateless). Clustering for Message-Driven Bean is not provided yet. Also, JBoss comes with a farming feature. Farming manages cluster hot-deployment. Hot-deploying an application (EAR, WAR or JAR application) on a machine causes the application to be hot deployed on all instances within the cluster.

3.2.2 Change impact analysis

An observable advantage of scaling the software architecture induced by J2EE, using JBoss, is that no development effort is required to realize the scalability requirements through replication, as when compared to the CORBA version. The clustered environment, which mainly includes the HA-JNDI, the HA-EJB for Entity Bean and Stateful Session Bean, and the farming do provide the primitives for scaling the software system. That is, no development effort is required to provide a clustering environment. However, configuring and deploying the application in the clustered environment are still required. Yet, the server need not have to be shutdown for configuration. JBoss manages dynamically the replicas, which means that nodes can be added at run-time. The modifications made on the configuration files are automatically taken into account by JBoss. This is certainly appreciated in environments constrained by high availability, as it is daunting to stop the server for maintenance. Whereas in the CORBA implementation, all the instances of JacORB which are running on the different hosts has to be shutdown for updating JacORB.

In brief, configuration includes the following: configuring clusters, HA-JNDI, HA-EJB, and farming. By default, one partition exists. When adding a partition, the cluster needs to be configured. This simply requires updating the cluster service file (i.e., cluster-service.xml). 11 lines of code are necessary to map a partition with a HA-JNDI service. The property file (*jndi.properties*) on the client-side has to be updated to enable the client to auto-discover the HA-JNDI servers. One line of code is necessary to update this file.

To cluster the EJBs, a special XML tag (clustered) has to be added to the Jboss.xml. To specify the partition(s) to be used, the (cluster-config) tag needs to be added to the same file. More, the load-balancing mechanism may need to be updated in the JBoss deployment descriptor. All of these changes involve 10 lines/bean. For stateful session beans, the cluster service file, cluster-service.xml, need to be updated to add a partition to the HASessionState service, involving 7 SLOC. Therefore, we need 39 SLOC to enable farming for all our partitions. The file farm-service.xml file, by default, enables the farming for one partition. To enable the farming for all the partitions, farm-service.xml file need to be updated; a link will need to be added between the FarmMemberService and a partition. For the Duke's Bank architecture, we use four partitions: two for the Account beans (Entity and Session) and two for the Transactions beans (Entity and Session). Thirty two SLOC need to be added for configuring a partition. This results in 128 SLOC. Other 33 lines of code are necessary to map a partition with the HA-JNDI service. Because four kinds of beans exist in the system, configuring the HA-EJB requires 40 lines to update the JBoss deployment descriptor of the beans. Thirteen SLOC are required. We note that Farming is not enabled by default, requiring the developer's intervention.

Table 6. Scalability in the J2EE version

Changes to make	4 partitions Source Lines of code (SLOC)
Install Jboss	1
Configuring clusters	96
Configuring HA-JNDI	34
Configuring HA-EJB	47
Configuring farming	39
Total for one host	217

3.3 Options Analysis and Discussion

In this section, we empirically evaluate the theory, the model, and demonstrate its applicability.

In summary, to scale the architecture of the Duke's Bank, the requirements depicted in Figure 2 need to be achieved. We have estimated their structural impact on both the CORBA and the J2EE versions. We have estimated the SLOC to be added for implementing the change on both versions, as depicted in tables 3-5. An observable advantage of scaling the software architecture induced by EJB is that no development effort is required to realize the scalability requirements through replication, as when compared to the CORBA version. Our hypothesis that middleware induced software architecture differs in coping with changes is verified to be true for the given change in scalability. Obviously, 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. However, a question of interest is how valuable this embedded flexibility is? We use the model developed in section 2 to answer this question. The objective is to quantify the flexibility value as a way for understanding the added value upon inducing the architecture with J2EE relative to CORBA. We seek an under-

standing of the added value on future savings in maintenance including ease of deployment and configuration upon accommodating the likely change in scalability.

The results could be summarized as follows. On the methodology level, we verify the claim that flexibility of the middleware-induced software architectures creates values in the form of real options. These options differ with the middleware-induced. Our claim that these options are revealing to the stability of the software architecture is verified to be true for the given change in scalability. The results show that value-based reasoning and real options can provide insights into architectural stability and investment decisions related to the evolution the software system. The options analysis confirms the validity of our claim that middleware induced software architectures differs in coping with changes in non-functional requirements. We draw some preliminary lessons and insights that could simulate future research in the area of relating requirements to software architectures and consequently advance our understanding to the architectural stability problem, when addressed from the evolution of the non-functional requirement perspective.

The Application of the ArchOptions Model. For this example, we focus our attention on the payoff of the call options (i.e., $\sum_{i=1\ldots n} E\,[\max(x_i V - C_{ei}, 0)]_{S1}$ relative to $\sum_{i=1\ldots n} E\,[\max(x_i V - C_{ei}, 0)]_{S0}$), as they are revealing for the flexibility of the architecture-induced in responding to the likely future changes. We construct a call option for the future scalability goal, where the change is analogues to buying an "architectural potential", paying an exercise price. The exercise price corresponds to the likely price to accommodate the change. The application of the model is thus done on the goal level, versus their corresponding requirement-refinements. Following the discussion of section 2, CORBA and J2EE correspond to M_0 and M_1 respectively. We refer to the architecture of the Duke's Bank as S_0 when induced by M_0 and S_1 when induced M_1. When necessary, we use \$6000 for man-month to cast the effort into cost. We show how we have estimated the parameters.

Table 7. Scaling the system using replication (1 Host): development, configuration, and deployment costs

		CORBA (JacORB)			EJB (JBOSS)		
		Optimistic	Most Likely	Pessimistic	Optimistic	Most Likely	Pessimistic
Development	Effort	24.1	30.2	37.7	0	0	0
	Cost, C_{ei}	96481	120602	150753	0	0	0
	SLOC	9240			0		
Configuration	Effort	0.4	0.5	0.6	0.4	0.5	0
	Cost, C_{ei}	1527	1909	2386	1558	1948	2435
	SLOC	213			217		
Deployment	Effort	0	0	0	0	0	0
	Cost, C_{ei}	0	0	0	0	0	0
	SLOC	0			0		

Estimating (C_{ei}). The exercise price corresponds to the cost of implementing scalability on each structure, given by C_{ei} for requirement i. As the replicas may need to be run on different hosts, we devise a general model for calculating C_e as a function of the number of hosts, given by:

$$C_e = \sum_{h=1\ldots k} (C_{dev}, C_{config}, C_{deploy}, C_{licesh}, C_{hardw})_k, \quad (3)$$

where, h corresponds to the number of hosts. C_{dev}, C_{config}, *and* C_{deploy}, respectively corresponds to the cost of development(if any), configuration, and deployment for the replica on host h. C_{licesh} and C_{hardw} respectively correspond to licenses and hardware costs, if any. All costs are given in (\$). We provides three values: optimistic, likely, and pessimistic for each parameter. All are calculated using COCOMO II [Boehm et al., 1995]. Upon varying the number of hosts, we only report on pessimistic values for this study, as they are revealing. We also ignore any associated hardware costs for the simplicity of exposition.

Capturing and estimating ($x_i V$). To value the "architectural potential" of S_1 relative to S_0 given by ($x_i V_{S1/S0}$), we take a structural approach to valuation. We use the expected savings (if-any) in development, configuration, and deployment efforts, when the scalability change needs to be accommodated on S_1 relative to S_0, and respectively denoted as $\Delta_{S1/S0} C_{dev}$, $\Delta_{S1/S0} C_{config}$, $\Delta_{S1/S0} C_{deploy}$. Relative savings in licenses and hardware may also be considered and respectively denoted by ΔC_{licesh}, ΔC_{hardw}. Below is a model for calculating $x_i V_{S1/S0}$, for change in requirement i.

$$x_i V_{S1/S0} = \sum_{h=1\ldots k} (\Delta_{S1/S0} C_{dev}, \Delta_{S1/S0} C_{config}, \Delta_{S1/S0} C_{deploy}, \Delta_{S1/S0} C_{licesh}, \Delta_{S1/S0} C_{hardw})_k \quad (4)$$

Similar description applies for ($x_i V_{S0/S1}$). The savings (if any), however, are uncertain and differ with the number of hosts, as the replicas may need to be run on different hosts.

Such uncertainty makes it even more appealing to use of "options thinking".

Calculating the volatility (σ). The volatility of the stock price (σ) is a statistical measure of the stock price fluctuation over a specific period of time; it is a measure of how uncertain we are about the future of the stock price movements. Volatility stands for the "fluctuation" in the value of the estimated x_iV. Intuitively, it "aggregates" the "potential" values of the structure in response to the change(s). We take the percentage of the standard deviation of the three x_iVs estimates-the optimistic, likely, and pessimistic values to calculate σ.

Exercise time (t) and free risk interest rate(r). As a simulation assumption, we set the exercise time to one year, assuming that the Duke's Bank need to accommodate the change in one year time. We set the free risk interest rate to zero (i.e., assuming that the value of money today is the same as that in one year's time).

Results and Observations. Below, we report on sample results and observations upon the application of the model (refer to observations 1& 2). We verify our hypothesis that the choice of a stable distributed software architecture has to be guided by the choice of the underlying middleware and its *flexibility* in responding to future changes in non-functional requirements (refer to observations 4 & 5).

Observation 1. Flexibility creates options: S_1 is more flexible than S_0 (due to the embedded primitives in J2EE); S_1 has created more options when compared to S_0.

Let us consider the scenario where we consider one host. For this scenario, we assume that the license cost (C_{licesh}) is zero for M_1 (e.g., the usage of JBoss an open source). Table 7 reports on the effort (man-month) and cost in ($); it provides three values: optimistic, likely, and pessimistic for each parameter. The $x_iV_{S1/S0}$ correspond to the difference- as reported in Table 8a. The overall expected savings of inducing the structure with S_1 relative to S_0 are in the range of $96450(pessimistic) to $150704 (optimistic). As far as the development effort is concerned, expected savings are in the range of $96481(pessimistic) to $ 150753 (optimistic) for realizing the scalability requirements. As far as configuration effort is concerned, S_1 has not reported any expected savings relative to S_0. However, these figures are insignificant. As far as the effort of deployment is concerned, both are comparable when it comes to SLOC. However, as a limitation on this dimension, interested reader may refer to section 3.4. We note that these figures are based on COCOMO II: the number of man-months is different from the time that will take for completing a project, termed as the development schedule. For example, a project could be estimated to require 50 man-months of effort but have a schedule of 11 months. Accordingly, the cost and relative savings, maybe adjusted relative to the schedule. We have relaxed this, as the aim of the exercise is to simulate the applicability of the model. The x_iVs will be used to quantify the added value, taking the form of options, due to the embedded flexibility on S_1 relative to S_0.

Table 8a shows that S_1 is in the money in response to the changes in scalability, when compared to S_0. Table 8a shows that S_1 is in the money relative to the development, configuration, and the deployment. The results of table 8a read that inducing the architecture with M_1 is likely to enhance the option value by an excess of $96450 (pessimistic) to $150704 (optimistic) over S_0, if the change in scalability need to be exercised in one year time. Thus, the results show that S_1 induced by M1 is likely to add more value in the form of options in response to the change, when compared to S_0. It is worth pointing out that though S_1 is flexible relative to the scalability change, it might not necessarily mean that it might be flexible with respect to other changes. Obviously, JBoss 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. This has lead to a notable savings in maintenance cost. Calculating the options of S_0 relative to S_1, we can see that S_0 is said to be out of the money for this change. The CORBA version has not added value, relative to J2EE, as the cost of implementing the change was relatively significant to "pull" the options, as reported in table 8b. The very low value of Vega means that possible changes in volatility have relatively little impact on the value of the options. The high value of Delta in Tables 8a and Table 8b roughly means that changes in *Xiv* could have high impact on the on the calculated options.

Observation 2. How worthwhile is the embedded flexibility in S_1 when induced in M_1, relative to S_0 when induced with M_0?

For this experiment, we consider the case where we use WebLogic server [http://www.bea.com/] as M_1 with an average upfront payable license cost C_{licesh}= $25000/h. As an upfront license fee is incurred, increasing the number of hosts may carry unnecessary expenditures that could be avoided, if we use M_0 instead. However, M_0 does also incur costs upon scaling the software system through the development of both the load balancing and the fault tolerance services. Such a cost, however, maybe "diluted" as the number of hosts increases. The cost is said to be distributed across the hosts and incurred once, as the developed services can be reused across other hosts. For this experiment, we assume that developing the fault tolerance and load services are upfront investments to buy growth options on the structure. An additional configuration and deployment cost materializes per host and sum up to the exercise price, Ce as in equation (3), when an additional host is needed to scale the software. $x_iV_{S0/S1}$ is calculated based on equation (4). We calculate the options of S_0 relative to S_1. We adjust the options by subtracting the upfront expenditure of de-

veloping both services on M_0, as reported in Table 8c. The adjusted options reveal situations in which S_0 is likely to add value relative to S_1, when the upfront cost is considered. These results may provide us with insights on the cost effectiveness of implementing fault tolerance and load balancing support to scale the software system relative to S_1, where a licensing cost is incurred per host. Therefore, a question of interest is: when is it cost effective to use M_0 instead of M_1? In other words, when the flexibility of M_1 cease to create value relative to M_0. We assume that for any k hosts, S_0 and S_1 are said to support U_kS_0 and U_kS_1 concurrent users, respectively; where U_kS_0 could be different or equal to U_kS_1. For the non-adjusted options results of table 8c, the results of read that inducing the architecture with M_0 is likely to enhance the option value of S_0 relative to S_1 (pessimistic) for the case of n hosts for n>0, under the condition that U_nS_0 >>=U_nS_1 and under the assumption that the upfront cost of developing fault tolerance and load balancing is relaxed. However, if we benchmark these options values against the cost of developing the load balancing and fault tolerance services (i.e., the upfront cost), we can see that payoff following developing these services is far from breaking even for less than 7 hosts, as depicted in figure 7.

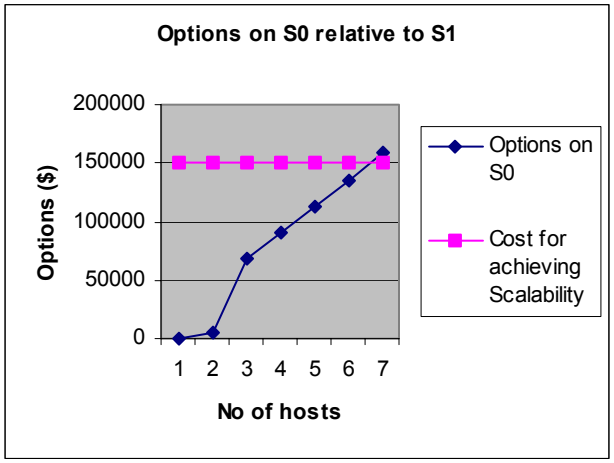


Figure 7. Options on S_0 relative S_1 prior to adjustment

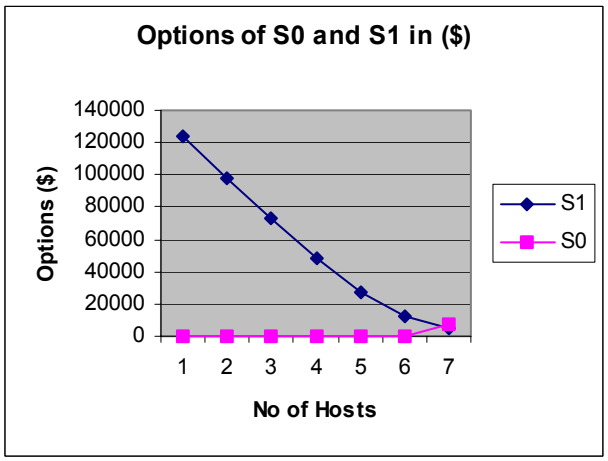


Figure 8. Options on S_0 and S_1 upon varying the No of hosts

Hence, once we adjust the options to take care of the upfront cost of investing to implement the both services, the adjusted options for S_0 relative to S_1 reports values in the money for the case of seven or more hosts, as shown in table 8c and sketched in figure 8. For the case of seven or more hosts, the M_0 appears to be a better choice under the condition that U_nS_0 >>=U_nS_1. These is due to the fact the expenditures in M_1 licenses increases with the number of hosts, henceforth, the savings in adopting M_1 cease to exist. For less than 7 hosts, M_1 has better potentials and appears to be more cost-effective under the condition that U_nS_1 >>=U_nS_0. For 7 or more hosts, M_0 appears to be of better potentials under the conditions U_nS_0 >>=U_nS_1, as depicted in figure 8. The use of this case to exercise the ArchOptions model has the prospect in providing an insight on how much do we need to invest in the adapted flexibility relative to the likely future changes, while not sacrificing much of the resources.

Table 8a. The options in ($) on the architecture induced by S_1relative to S_0 for one host, with S_1license cost (C_{licesh})=zero

		Ce	Xv	σ	T	Options	Delta	Vega
Overall	Opti-mistic	1158	96450	22.7	1	94892	1	9.1149E-71
	Likely	1948	120563			118615	1	1.1628E-70
	Pes-simis-tic	2435	150704			148269	1	1.4533E-70
Devel-opment	Opt	0	96481	22.7	1	96481	1	0
	Likely	0	120602			120602	1	0
	Pes.	0	150753			150753	1	0
Con-figura-tion	Opt	1558	-31	22.7	1	0	0	0
	Likely	1948	-39			0	0	0
	Pes.	2435	-49			0	0	0
Deploy-ment	Opt	0	0	22.7	1	0	0	0
	Likely	0	0			0	0	0
	Pes.	0	0			0	0	0

Table 8b. The options in ($) on the architecture induced by S_0relative to S_1 for one host, with (C_{licesh})=zero

		Ce	Xv	σ	T	Options	Delta	Vega
Overall	Opti-mistic	96450	31	22.7	1	0	0	0
	Likely	120563	39			0	0	0
	Pes.	150704	49			0	0	0

Table 8c. Options in ($) on S_0 relative to S_1 with (C_{licesh})= $25000 and σ=*22.7 and pessimistic* Ce

	Ce	Xv	Options	Adjusted Options	Conc. Users
1	2386	25049	2343	0	$U1S_0$ vs $U1S_1$
2	4772	50049	4772	0	$U2S_0$ vs $U2S_1$
3	7158	75049	67891	0	$U3S_0$ vs $U3S_1$
4	9544	100049	90505	0	$U4S_0$ vs $U4S_1$
5	11930	125049	113119	0	$U5S_0$ vs $U5S_1$
6	14316	150049	135733	0	$U6S_0$ vs $U6S_1$
7	16702	175049	158347	7643	$U7S_0$ vs $U7S_1$

Observation 3. Selecting a more stable architecture

The change impact analysis has shown that the architectural structure of S_1 is left intact when the scalability change needs to be accommodated. However, the structure of S_0 has undergone some changes, mostly on the architectural infrastructure level to accommodate the scalability requirements. From a value-based perspective, the search for a potentially stable architecture requires finding an architecture that maximizes the yield in the added value, relative to some future changes in requirements. As we are assuming that the added value is attributed to flexibility, the problem becomes selecting an architecture that maximize the yield in the embedded or adapted flexibility in a software architecture relative to these changes. Even, if we accept the fact that modifying the architecture or the infrastructure is the only solution towards accommodating the change, valuation the impact of the change becomes necessary to see how far we are expending to "re-maintain" or "re-achieve" architectural stability relative to the change. Note that the economic interplay between evolving requirements, the flexibility of the architecture to accommodate the change, the structural impact, and the corresponding cost/value implications is the key towards selecting a "more" stable architectures that tends to add value as the requirements evolve. Though it might be appealing to the intuition that the "intactness" of the structure is the definitive criteria for selecting a "more" stable architectures, the practice reveals a different trend; it nails down to the potential added value upon exercising the change.

If you consider the case of S_0 and S_1 in response to the change in scalability for one host (table 8a), the flexibility has yielded a better payoff for S_1 than for S_0, while leaving S_1 intact. This implies that inducing the Duke's Bank software architecture with M_1 is likely to be more stable relative to the future change in scalability, than when induced with M_0. However, the situation and the analysis have differed upon varying the number of hosts and upon factoring a license costs for S_1. Though S_0 has undergone some structural changes to accommodate the change, the case has shown that it is still acceptable to modify the architecture and to realize added value under the conditions that $U_nS_0 >>= U_nS_1$ for 7 or more hosts (Table 8c, Figure 8). Hence, what matters is the added value upon either embarking on a "more" flexible architecture, or investing to enhance flexibility which is the case for implementing load balancing and fault tolerance on S_0. For the case of WebLogic, Though M_1 is in principle more flexible (the case of), the flexibility comes with a price, where the flexibility turned to be a liability rather than a value for 7 or more hosts, as when compared with the JacORB, under the condition that $U_nS_0 >>= U_nS_1$. The case verifies our claims that the value of flexibility can guide towards the selection of architectures that tend to add more value, as the requirements evolve. These architectures have the potential of being potentially stable.

The options analysis has complemented the structural analysis to quantify the impact of the change on the software architecture. The intuition is that complementing the structural impact analysis with a value-based back-of-the-envelope calculation, the combination provides the architect/analyst with a useful tool for understanding extent to which the software system tend to be flexible relative to a likely change in requirements, a cost/value indictors of the impact of the change on the structure, the likely success (failure) of the software system evolution, and consequently the potential stability of the software architecture relative to the change.

Observation 4. Understanding Architectural Stability has to be done in connection with the solution domain

Our hypothesis that middleware induced-software architectures differ in coping with changes is verified to be true for the given change. Based on the pervious observations, we can see that the stability of S_1 and S_0 appears to be dependent on the flexibility of the middleware in accommodating the likely changes in the scalability requirements. For the category of distributed software systems that are built on top of middleware, the results of the case study affirm the belief that investigating the stability of the distributed software architecture could be fruitless, if done in isolation of the middleware, where the middleware constraints and dominate much of the solution that relate to the non-functionalities, managing system resources, and their ability to smoothly evolve over the life time of the software system. Hence, the development and the analysis for architectural stability and evolution shall 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: a considerable part of the distributed system implementation could be available, when the architecture is defined, for example, 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, with a given architectures S_1 and S_0 was respectively guided by the clustering primitives provided by M_1 and the core capabilities provided by M_0 to support load balancing and fault tolerance. Interestingly, Di Nitto and Rosenblum [1999] 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". Medvidovic, Dashofy and Taylor [2003] state the idea of coupling the modeling power of software architectures with the implementation support provided by middleware. They noted, "architectures and

middleware address similar problems, that is large-scale component-based development, but at different stages of the development life cycle." In more abstract terms, Rapanotti, Hall, Jackson, and Nuseibeh [2004] 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" [Rapanotti et al., 2004].

The "coupling" between the middleware and the architecture becomes of higher interest in case of developing and analyzing software systems for 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 understanding the stability of the software architectures from a pragmatic perspective.

Observation 5. Understanding Architectural Stability: Intertwined with changes in non-functional requirements, style, and the middleware

Following the definition of Di Nitto and Rosenblum[1999], 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 S_1 and S_0 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 only due to the architectural style, but also due to the primitives that are built-in 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 is not revealing for the stability of the software architecture 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, Sullivan et al. [1997] 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. Sullivan et al. [1997] support 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 S_1, for example, is that no development effort required to realize the scalability requirements through replication, as when compared to that of S_0, knowing that in principle the style of S_1 and S_0 exhibit similar capabilities.

Engineering for stability and evolution, 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., [Dardenne et al., 1993]), 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. 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. This, we believe, is a pragmatic need towards engineering requirements and developing "evolvable" software architectures that tend to be stable as the non-functional requirements evolve.

3.4 Critical assessment of the results

We have used change in scalability, a representative critical change in non-functional requirements, to apply the model and steer the study. We have appealed to the use of structural criteria, combined with value-based analysis, to inform the tradeoff and select a "more" stable architecture. Though the reported observations reveal a trend that agrees with the intuition, research, and the state-of-practice, confirming the validity of the observations are 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. We note that the primary aim of this case study is to exercise the model. Under no considerations should the results be regarded as a definite distinction of the merit of one technology over the other, but yet still revealing on the scalability dimension. The reason is due to the fact that we have only used "fla-

vors" of CORBA and J2EE, respectively through JacORB; and JBoss and WebLogic.

The options analysis is based on the likely effort for scaling the software system. However, we note that the flexibility value is underestimated for S_1 relative to S_0, as we have relaxed considering: (i) the added value due to the provision of hot deployment; JBoss provides hot-deployment and clustered hot-deployment to deploy application without needing to restart the server. Obviously, the quantification of such a value is both business and domain dependent; it can be quantified by estimating the losses avoided from stopping the service; (ii) the added value due to the ease of future maintainability and reduced complexity in the J2EE version, when compared to the CORBA one. For example, in J2EE all the configurations related to the server objects are made in the deployment descriptor; therefore, it results in a better code maintainability in contrary to CORBA where the server object configuration is made in the code. However, relaxing the consideration of the above does not affect the validity of our conclusions, as the results are already in favour of the J2EE-induced version.

Experts may question our use of [Black and Scholes, 1973] to options valuation, as the satisfaction of the spanning condition may be doubtful. We argue that our use for the design and the corresponding implementation of scalability on TAO as guidelines bear a resemblance to the concept of a "twin asset", for we are reusing a past development experience to inform the valuation. We also argue that valuation based on man-month does implicitly hold market-based data and is still done in relation with the market. Alternatively, we could have cast the options model to use different options valuation (e.g., [Cox et al., 1979]). However, the application of [Black and Scholes, 1973] offers a closed and an easy-to-compute solution, for it assumes that x_iV is lognormaly distributed, not requiring x_iV to be probability-adjusted for rise and drop in value, as when compared to [Cox et al., 1979]. Following the argument of [Sullivan et al., 2001], such models need not be perfect: what is essential is that they capture the most important terms; their assumptions and operation must be known and understood so that the analyst can evaluate their predictions.

4. Related Work

In this section, we provide a quick overview of closely related research on: (i) architectural stability research; (ii) the use of real options in software design and engineering; (iii) related research on architectural evaluation, and (iv) ongoing research on the "coupling" of software architecture and middleware.

Architectural stability in perspective. Ongoing research on the relation between requirements and software architectures has considered the architectural stability problem as an open research challenge and *difficult* to handle [van Lamsweerde, 2000; Finkelstein, 2000; Nuseibeh, 2001; Jazayeri, 2002; Emmerich, 2002]. In particular, Finkelstein motivated research in architectural stability; he described the problem in [Finkelstein, 2000]. Nuseibeh proposed the "Twin Peaks" model, a partial and simplified version of the spiral model [Nuseibeh, 2001]. The cornerstone of this model is that a system's requirements and its architecture are developed concurrently; that is, they are "inevitably intertwined" and their development are interleaved. Nuseibeh advocated the use of various kinds of patterns – requirements, architectures, and designs- to achieve the model objectives. As far as architectural stability is concerned, Nuseibeh had only exposed a tip of the "iceberg" (as referred by Nuseibeh): development processes that embody characteristics of the Twin Peaks are the first steps towards developing architectures that are stable in the face of inevitable changes in requirements. Nuseibeh 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? Our work addresses some of these questions.

Not far from the motivation of bridging the gaps between requirements and software architectures, van Lamsweerde noted that the goal-oriented approach to requirements engineering may support building and evolving software architectures guaranteed to meet both its functional and non-functional requirements [van Lamsweerde, 2000]. van Lamsweerde acknowledge that:

> "…. The conflict between requirements volatility and architectural stability is a difficult one to handle" [van Lamsweerde, 2000].

Jazayeri has looked at the architectural stability problem from a software evolution perspective [Jazayeri, 2002]. Jazayeri motivated the use of retrospective approaches for evaluating software architectures for stability and evolution. Retrospective evaluation looks at successive releases of the software system to analyze how smoothly the evolution took place. The analysis relies on comparing properties from one release of the software to the next. The intuition is to see if the system's architectural decisions remained intact throughout the evolution of the system, that is, through successive releases of the software. Jazayeri's approach uses simple metrics such as software size metrics, coupling metrics, and color visualization to summarize the evolution pattern of the software system across its successive releases. The evaluation assumes that the system already exists and has evolved making this approach not preventive and unsuitable for early evaluation (unless the evolution pattern is used to predict for the stability of the next release). In the absence of dedicated tools, the evaluation appears to be expensive and unpractical, for it requires in-

formation to be kept for each release of the software. Yet, such data is not commonly maintained, analyzed, or exploited, as noted by Jazayeri. Moreover, the problem of architectural stability and the architecture "resilience" to evolution is strategic in essence and not purely technical. Jazayeri has addressed the problem from a purely technical perspective. Instead, we aim at to assist in proactively engineering stable architectures. We believe that the economic interplay between evolving requirements and architectural stability needs to be addressed.

The use of real options in software engineering. Economics approaches to software design appeal to the concept of static Net Present Value (NPV) as a mechanism for estimating value [Boehm and Sullivan, 2000]. These techniques, however, are not readily suitable for strategic reasoning of software development as they fail to factor flexibility [Boehm and Sullivan, 2000; Erdogmus et al., 1999]. The use of strategic flexibility to value software design decisions has been explored in, for example, [Erdogmus and Vandergraff, 1999; Erdogmus and Favaro, 2002; Erdogmus 2000; Sullivan; 1996; Sullivan et al., 1999; Sullivan 2001] and real options theory has been adopted to value the strategic flexibility: Baldwin and Clark [2001] studied the flexibility created by modularity in design of components (of computer systems) connected through standard interfaces. Sullivan et al. [1996; 1999; 2001] pioneered the use of real options in software engineering. Sullivan et al. [1996; 1999] suggested that real options analysis can provide insights concerning modularity, phased projects structures, delaying of decisions and other dynamic software design strategies. Sullivan et al. [1999] formalized that option-based analysis, focusing in particular on the flexibility to delay decisions making. An interesting approach that has inspired the early stages of our work is that of Sullivan et al. [2001]. Sullivan et al. [2001] extended Baldwin and Clark's theory [2001] that is developed to account for the influence of modularity on the evolution of the computer industry. Sullivan et al. [2001] use the model developed in [Baldwin and Clark, 2001] to treat the "evolovability" of software design using the value of strategic flexibility. Specifically, they argued that the structure and value of modularity in software design creates value in the form of real options. A module creates an option to invest in a search for a superior replacement and to replace the currently selected module with the best alternative discovered, or to keep the current one if it is still the best choice. The value of such an option is the value that could be realized by the optimal experiment-and-replace policy. Knowing this value can help a designer to reason about both investment in modularity and how much to spend searching for alternatives. Erdogums [1999] describes how strategic flexibility in software development, involving COTS components, can be valued using real options. An interesting use of real options theory is that of [Erdogums and Favaro, 2002]. Erdogmus and Favaro uses real options to value the inherent flexibility in the Extreme Programming (XP), where they have considered XP as a lightweight process that is well positioned to respond to change and future opportunities; hence, creating more value than a heavy-duty process that tends to freeze development decisions.

Architectural evaluation. Interested reader may refer to [Bahsoon and Emmerich, 2003b] in which we provide a comprehensive survey on architectural evaluation methods. In short, we have distinguished between two classes of software architecture evaluation methods: (i) general-purpose methods that evaluate software architectures for qualities that need to be met by the system (e.g. performance, security, and modifiability) and (ii) an emerging class of methods that explicate evaluation for stability and evolution. Apart from our work, the only evaluation method under the latter category is the work of [Jazayeri, 2002] and sufficiently detailed in the above subsection.

Existing methods to architectural evaluation have ignored any economic considerations, with CBAM [Asundi and Kazman, 2001] being the notable exception. The evaluation decisions using these methods tend to be driven by ways that are not connected to, and usually not optimal for value creation. Factors such as flexibility, time to market, cost and risk reduction often have higher impacts on value creation [Boehm and Sullivan, 2000]. Hence, flexibility is in the essence. In our work, we link flexibility to value, as a way to make the value of stability tangible.

Relating CBAM to our work, the following distinctions can be made: with the motivation to analyse the cost and benefits of architectural strategies, where an architecture strategy is subset of changes gathered from stakeholders, CBAM does not address stability. Further, CBAM does not tend to capture the long-term and the strategic value of the specified strategy. ArchOptions, in contrast, views stability as a strategic architectural quality that adds to the architecture values in the form of *growth options*. When CBAM complements ATAM [Kazman et al., 1998] to reason about qualities related to change such as modifiability, CBAM does not supply rigorous predictive basis for valuing such impact. Plausible improvements of the existing CBAM include the adoption of real options theory to reason about the value of postponing investment decisions. CBAM uses real options theory to calculate the value of option to defer the investment into an architectural strategy. The delay is based on cost and benefit information. In the context of the real options theory, CBAM tends to reason about the *option to delay* the investment in a specific strategy until more information becomes available as other strategies are met. ArchOptions, in contrast, uses real options to value the flexibility provided by the architecture to expand in the face of evolutionary requirements; henceforth, referred to as the options to expand or growth options.

On the "coupling" of software architectures and middleware. There is only very little work on the "coupling" of middleware and software architectures. Notable

exceptions include [Jazayeri, 1995; Gall et al., 1997; Sullivan et al., 1997; Oreizy et al., 1998; Di Nitto and Rosenblum, 1999; Metha et al., 2000; Medvidovic et al., 2003; Denaro et al., 2004].

Jazayeri [1995] explores the relationship between software architectures and component technologies. Gall et al. [1997] have looked at an existing component framework, the C ++ standard library, and identified the architectural style induced. Sullivan et al. [1997] 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. Sullivan et al. [1997] support this claim by demonstrating that a style, that in principle seems to be easily implementable using the COM middleware, is actually incompatible with it. Oreizy et al. [1998] discuss the importance of complementing *component interoperability models* with explicit architectural models. Di Nitto and Rosenblum [1999] devised the term *middleware-induced architectural styles*. Middleware-induced architectural styles uses Architecture Definition Languages (ADLs) to describe the assumptions and constraints that middleware infrastructures impose on the architecture of system. They have evaluated ADLs for their suitability in defining middleware-induced architectural styles. Metha et al. [2000] propose a classification framework of software connectors. They describe types of services provided by connectors for enabling and facilitating component interactions. They aim at building implementation topologies (e.g., bridging of middleware) that preserve the properties of the original architecture, under the motivation of coupling architectures and middleware. Medvidovic et al. [2003] state the idea of "coupling" the modelling power of software architectures with the implementation support provided by middleware. They have noticed that "architectures and middleware address similar problems, that is large-scale component-based development, but at different stages of the development life cycle." They have investigated the possibility of defining systematic mappings between architectures and middleware. Recently, Denaro et al. [2004] measures performance attributes of an architecture based on the early available implementation support provided by the middleware.

In summary, recent research effort on the relation between software architectures and middleware has been motivated by pragmatic needs. The effort has revolved on issues such as investigating the compliancy of architectural styles with middleware; capabilities that the middleware and the architecture can bring when "coupled" to understand quality attributes of the system such as performance; mapping between middleware and software architectures; and semantics and syntactical issues related to the mapping process.

As it has been noted in several occasions [Emmerich 2000b; Emmerich 2002], research on software architectures has over-emphasized functionality and not sufficiently addressed how global properties and non-functional requirements are achieved in an architecture, where these requirements cannot be attributed to individual components or connectors. Though we believe 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. As far as the architectural stability problem is concerned, no effort has been devoted for understanding the evolution of non-functional requirements in relation to both the architecture and the middleware, when coupled. Our use of architectural flexibility and its value as metric to inform the decision of selecting a "more" stable middleware-induced architecture is novel and only a step toward such an understanding using a *value-based* [EDSR 1-6] reasoning.

5. Summary and Future Work

We have hypothesized that the choice of a stable distributed software architecture has to be guided by the choice of the middleware-induced and its *flexibility* in responding to future changes in non-functional requirements. We have devised an option-based model to value such flexibility and guide the selection. We have empirically evaluated the model using a case that adequately represent a medium-size component-based distributed architecture. We have used change in scalability, a representative critical change in non-functional requirements, to apply the model and steer the study. We have reported on how a likely future change in scalability could impact the architectural structure of two versions, each induced with a distinct middleware, CORBA and J2EE. We have appealed to the use of replication, as an architectural mechanism to scale the software system. We have estimated the structural impact of implementing this mechanism on both the CORBA and the J2EE versions. We have estimated the expected relative savings in maintenance including development, deployment, and configuration efforts. We have applied the ArchOptions model. Our hypothesis that middleware induced software architecture differs in coping with changes is verified to be true for the given change in scalability. We have reported on some observations that could stimulate future research in the area of relating requirements to software architectures. 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 aim to investigate as part of our ongoing research agenda. The contribution demonstrates that using *value-based* reasoning, we can analyze for architectural stability and support the development (evolution) of software systems that need to adapt to the inevitable evolving requirements.

6. References

[1] Anton, A.: Goal-based Requirements Analysis. In: Proc. 2nd IEEE Int. Conf. Requirements Engineering. April (1996)

[2] Asundi, J., Kazman, R.: A Foundation for the Economic Analysis of Software Architectures. In: Proceedings of the Third Workshop on Economics-Driven Software Engineering Research (2001)

[3] 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 (2004b)

[4] Bahsoon, R., Emmerich, W.: Applying ArchOptions to Value the Payoff of Refactoring. In: Proceedings of the Sixth ICSE Workshop on Economics-Driven Software Engineering Research (2004a)

[5] 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 (2003a)

[6] Bahsoon, R., Emmerich, W.: Evaluating Software Architectures: Development, Stability, and Evolution. In: Proceedings of IEEE/ACS Computer Systems and Applications, IEEE CS Press (2003b)

[7] Bahsoon, R.: Evaluating Software Architectures for Stability: A Real Options Approach. In: Proceedings of the Doctoral Symposium of the 25th International Conference on Software Engineering (2003)

[8] Baldwin, C. Y., Clark, K. B.: Design Rules - The Power of Modularity. MIT Press (2001)

[9] Black, F., Scholes, M.: The Pricing of Options and Corporate Liabilities. Journal of Political Economy (1973)

[10] Boehm, B., Clark, B., Horowitz, E., Madachy,R., Shelby, R., Westland, C.: The COCOMO 2.0 Software Cost Estimation Model. In: International Society of Parametric Analysts (1995)

[11] Boehm, B., Sullivan, K. J.: Software Economics: A Roadmap. In: Finkelstein, A. (ed.): The Future of Software Engineering (2000)

[12] Cox, J., Ross, S., Rubinstein, M.: Option Pricing: A Simplified Approach. Journal of Financial Economics. Vol.7 (3). (1979) 229-264

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

[14] Denaro, G., Polini A., Emmerich W.: Performance Testing of Distributed Component Architectures. In: S. Beydeda and V. Gruhn (eds), Building Quality into COTS Components - Testing and Debugging. Springer (2004)

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

[16] Sun Microsystems Inc.: Duke's bank application, http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html.

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

[18] Emmerich, W.: Engineering Distributed Objects. John Wiley & Sons, Chichester, UK (2000a)

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

[20] Erdogmus, H. and Vandergraaf. J: Quantitative approaches for assessing the value of COTS-centric development. In: Proc. Sixth International Symposium on Software Metrics (METRICS' 99), November 4-6, Boca Raton, FL (1999)

[21] Erdogmus, H., Boehm, B., Harriosn, W., Reifer, D. J., and Sullivan, K. J.: Software Engineering Economics: Background, Current Practices, and Future Directions. In: Proceeding of 24th International Conference on Software Engineering, Orlando, FL. (2002)

[22] Erdogmus, H., Favaro, J: Keep Your Options Open: Extreme Programming and Economics of Flexibility, In: XP Perspective, Addison Wesley (2002)

[23] Erdogmus, H.: Value of Commercial Software Development under Technology Risk. The Financier, vol. 7. (2000)

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

[25] Gall, H., Jazayeri, M., Klösch, R., Trausmuth, G.: The Architectural Style of Component Programming. COMPSAC (1997)

[26] Gamma E. et al.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Longman, Reading, Mass. (1995)

[27] Henning, M. and Vinoski, S: Advanced CORBA Programming With C++, Addison-Wesley Longman, Reading, Mass (1999)

[28] Hull, J. C.: Options, Futures, and Other Derivative Security. Third edition, Prentice-Hall (1997)

[29] Jazayeri, M.: Component Programming - a Fresh Look at Software Components, In: ESEC, 457-478 (1995)

[30] Jazayeri, M.: On Architectural Stability and Evolution. Lecture Notes in Computer Science, Springer Verlag, Berlin (2002)

[31] JGroups Website, http://www.jgroups.org.

[32] Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T., and Carrière, S.J.: The Architecture Tradeoff Analysis Method. In: Proceedings of ICECCS, Monterey, CA. (1998)

[33] Labourey, S. and Burke B.: JBoss clustering documentation, JBoss Group LLC (2003)

[34] Medvidovic N, Dashofy E, Taylor R: On the Role of Middleware in Architecture-based Software Development. International Journal of Software Engineering and Knowledge Engineering 13(4) (2003)

[35] Mehta, N., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of the 22nd Interna-

tional Conference on Software Engineering, ACM Press (2000)

[36] Myers, S. C.: Finance Theory and Financial Strategy. Corporate Finance Journal. Vol. 5(1) (1987) 6-13

[37] Nuseibeh, B.: Weaving the Software Development Process Between Requirements and Architectures. In: Proceedings of STRAW 01 the First International Workshop From Software Requirements to Architectures, Toronto, Canada (2001)

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

[39] Object Management Group: The Common Object Request Broker: Architecture and Specification, 2.3 ed., Framingham, Mass. (1999b)

[40] Object Management Group: Fault Tolerant CORBA Specification, OMG document orbos/99-12-08 ed., OMG, Needham, Mass. (1999a)

[41] Oreizy, P., Medvidovic, N., Taylor, R. and D. Rosenblum, D.: Software Architecture and Component Technologies: Bridging the Gap. In Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA, January (1998)

[42] Othman, O., O'Ryan, C., Schmidt, D.C.: Designing an Adaptive CORBA Load Balancing Service Using TAO. IEEE Distributed Systems Online 2(4) (2001b)

[43] Othman, O., O'Ryan, C., Schmidt, D.C.: Strategies for CORBA Middleware-Based Load Balancing. IEEE Distributed Systems Online 2(3) (2001a)

[44] EDSER 1-6. Proceedings of the Workshops on Economics-Driven Software Engineering Research: In conjunction with the 21st through 26th International Conference on Software Engineering (1999 - 2004)

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

[46] Schmidt D.C. et.: Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2, John Wiley & Sons, New York (2000).

[47] Schmidt, D.C., Levine, D.L., and Mungee, S.: The Design and Performance of Real-Time Object Request Brokers, Computer Communication, vol (21)(4), pp. 294-324 (Apr. 1998)

[48] Schwartz, S., Trigeorgis, L.: Real options and Investment Under Uncertainty: Classical Readings and Recent Contributions. MIT Press Cambridge, Massachusetts (2000)

[49] Stafford, J. A., Wolf, A. W.: Architecture-Level Dependence Analysis for Software System. International Journal of Software Engineering and Knowledge Engineering. Vol. 11(4) (2001) 431-453

[50] Sullivan, K. J., Griswold, W., Cai, Y., Hallen, B.: The Structure and Value of Modularity in Software Design. In: Proceedings of ESEC/FSE-9, Vienna, Austria (2001) 99-108

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

[52] Sullivan, K. J.: Chalasani, P., Jha, S., Sazawal, V.: Software Design as an Investment Activity: A Real Options Perspective. In: Real Options and Business Strategy: Applications to Decision-Making. Trigeorgis L.(ed.) Risk Books (1999)

[53] Sullivan, K. J.: Software Design: The Options Approach. In: 2nd International Software Architecture Workshop, Joint Proceedings of the SIGSOFT '96 Workshops. San Francisco, CA (1996) 15–18

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

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