Towards Architecture-based Autonomic Software Performance Engineering

Xu Zhang*, Chung-Horng Lung*
Greg Franks*

*Carleton University, Ottawa, Canada
{zhangxu, chlung, greg.franks}@sec.carleton.ca

Abstract. Autonomic systems can be self-adaptive and have the potential to achieve high performance through run-time configuration changes. This paper describes an architecture-centric self-adaptive approach and presents a simple application in a distributed system where it can be advantageous to switch architectures based on the workload being presented to the system. The self-adaptive framework is built on top of a generative system which comprises three software architectural alternatives, namely Single Thread (ST), Half-Sync/Half-Async (HS/HA) and Leaders-Followers (LFs). A software performance analysis tool called the Layered Queuing Network Solver (LQNS) is integrated into the framework to support the architecture selection process. A comparison of the performance of the three different software architecture alternatives is also presented. The results from this analysis are used to support the construction of a performance knowledge base and analysis policies for the self-adaptive system.

1 Introduction

The complexity of computer systems is increasing at a fast speed and the number of computing devices in use is growing dramatically (Parashar and Hariri, 2007). As a result, IT personnel have to face the burden of supporting tasks such as configuration, maintenance and system performance evaluation (Enterprise Managment Associates, 2006). Further, manual control of a distributed computing system or a Web system is prone to errors, time-consuming, and expensive. The goal of autonomic computing, initiated by IBM (Parashar and Hariri, 2007), is to define rules for a system for controlling its behavior so that the system regulates its actions to automatically configure, heal, protect, and optimize itself (Kephart and Chess, 2003). Many research projects related to autonomic computing have been started (Muller et al., 2006), but there is still a lack of research in the area of evaluating performance of software architectures supporting architecture-based self-adaption at runtime.

Software architectures have significant impact on the performance of a system, however, determining an optimal architecture early in the life cycle of a project is a challenging issue, For instance, from the concurrency management perspective, two efficient architectural alternatives have been proposed: Half-Sync/Half-Async (HS/HA) and Leader/Followers (LFs) (Schmidt et al., 2000). The question that an architect often has to address is that of finding an architecture

which is sufficiently efficient. Software Performance Engineering (SPE) (Smith and Williams, 2001), which relies on performance modeling, has been recognized as an effective approach for addressing this issue.

The performance of a system is also often dependent on the operating systems and the hardware. Various factors, including workloads, system resources, contentions, complex interactions between the application and the kernel, and increasing hardware complexity, e.g., threading and parallelism, may affect the performance. Therefore, if an autonomic system can reconfigure the architecture of a running system, integrating a software performance analysis tool or a mechanism into the autonomic control system is necessary.

This paper proposes a self-adaptive approach at the architecture level and integrates a performance analysis tool, the Layered Queuing Network Solver (LQNS) described in Franks et al. (2009), into the self-adaptive system. In practice, numerous scenarios related to workloads, configurations, and resource usages need to be conducted for performance evaluation of a product, which is often hindered by limited time and resources. Performance modeling complements the self-adaptive framework by facilitating further scalability analysis or higher workload evaluation. The operational generative system produces realistic performance data which are collected and used for the simulation, and/or to build a performance knowledge base (Woodside et al., 2007). The performance knowledge base can support self-adaptation at runtime.

The major contributions of this paper are first to conduct performance modeling and comparison for three different software architectural patterns by building Layer Queuing Network (LQN) models (Franks et al., 2009). Secondly, the approach introduces the concept of integrating the performance models into a self-adaptive framework at the architecture level. The system has the ability to achieve higher performance than a static system based on environment parameters, such as arrival rate, throughput, and waiting time, and architecture specific features.

This paper is organized as follows: Section 2 briefly describes related background on autonomic computing, a generative system, and the LQN model. Section 3 illustrates the comparison of two different software architectural alternatives by building the LQN models. Section 4 demonstrates the self-adaptive framework. Finally, Section 5 presents the conclusions and some future research directions.

2 Background

This section briefly provides the background information on autonomic computing and outlines IBM's architectural blueprint for building a self-adaptive system. Then a generative system in distributed applications and LQN modeling are also introduced.

2.1 Overview of the Autonomic Computing

A significant challenge of modern distributed computing systems is that they become tremendous complex. Manual control of a large distributed computing system is prone to errors, time-consuming, expensive and they tend to increase very quickly as the size of the system grows. The goal of the "Autonomic Computing Initiative", started by IBM, is to define rules for a system for controlling its behavior so that the system itself regulates its actions, much

like the autonomic nervous system of an animal regulates actions, such as breathing, without conscious effort (Parashar and Hariri, 2007).

The following four functional areas for autonomic computing have been defined by IBM (IBM Autonomic Computing Architecture Team, 2006):

1. Self-Configuration: automatic configuration of components;
2. Self-Healing: automatic discovery and correction of faults;
3. Self-Optimization: automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;
4. Self-Protection: pro-active identification and protection from arbitrary attacks.

This paper will focus on the third item: self-optimization or self-adaptive for higher performance. IBM's blueprint for autonomic computing defines autonomic managers that are used to manage software or hardware resources. The autonomic manager is a component that implements the control loop including four main functions that share knowledge (IBM Autonomic Computing Architecture Team, 2006). They are

1. *monitor*: collect the details the autonomic manager needs from the system;
2. *analyze*: analyze those details to determine if something needs changing;
3. *plan*: create a plan or sequence of actions that specifies the necessary changes;
4. *execute*: perform those actions.

When these functions can be automated, an intelligent control loop is formed.

2.2 A Generative System

Our self-adaptive framework is built on top of an existing component-based distributed and concurrent system. The system is an architecture-centric generative framework (Lung et al., 2006), shown in Figure 1, that can be adopted for Web systems, client-server, or peer-to-peer applications. The generative framework can be used to help the architect rapidly develop a prototype and subsequently evaluate the software architecture effectively. The framework is composed of three architecture alternatives, namely HS/HA, LFs and single thread (ST), that are built with robust software components based on recognized patterns or existing solutions. The framework can then be used to instantiate specific types of software architecture, as selected by the architect. Such a framework facilitates rapid prototype development for comparison among different alternatives based on actual execution data measured from each alternative. In other words, the framework can generate quantitative and concrete operational information, or collect more accurate or realistic data such as workload, processing speed, response time, and packet loss (Lung et al., 2006). Each architectural alternative is made up of generic reusable components (represented by the letters A, B, C, D, and E in Figure 1) and specific features (represented by the letters X and Y). The designer simply needs to focus and build the application on top of the framework.

2.3 Software Patterns

Software patterns describe established proven solutions to recurring problems. Patterns have the potential to improve software quality and/or productivity. There are different kinds of

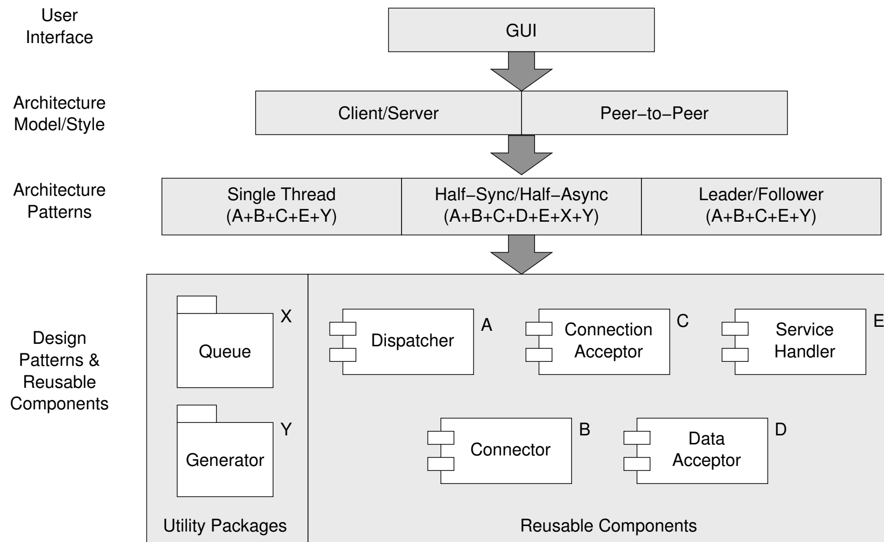


FIG. 1 – *Structure and Components of the Generative Framework (Lung et al., 2006).*

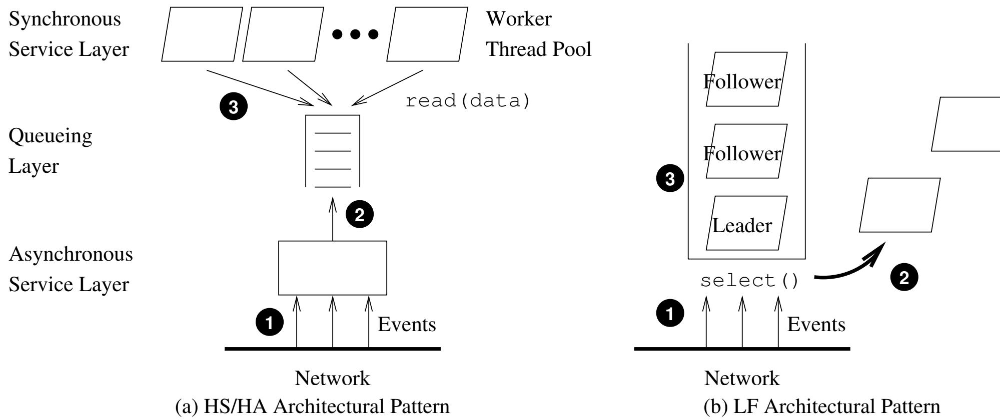
patterns: architectural patterns, design patterns, coding patterns (idioms), etc. This paper deals with two specific architectural patterns in concurrency: HS/HA and LFs.

The HS/HA pattern is an architectural solution that decouples asynchronous and synchronous service processing in concurrent systems in order to simplify programming without unduly reducing performance (Schmidt et al., 2000). The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing. Figure 2a shows three layers in this pattern. The *asynchronous layer* interacts with external requests (1). Once the requests are received, they are stored in a queue in queuing layer (2). These requests are then removed and processed by the *synchronous layer* (3). As a result, the synchronous layer does not need to deal with external requests. Layers are independent and can perform operations concurrently. Therefore, this pattern advocated for "performance-sensitive concurrent applications" (Schmidt et al., 2000).

The LFs pattern, shown in Figure 2b, allows one thread called the *leader* to wait for a incoming network request (1). All other idle threads, called *followers*, queue up waiting their turn to become the leader. When the current leader thread receives a request (2), it promotes a follower thread to become the new leader (3) and then processes the request. At this point, the former leader and the new leader thread can execute concurrently (Schmidt et al., 2000).

2.4 Layered Queueing Networks (LQN)

The Layered Queuing Network (LQN) model is a canonical form for extended queuing networks with a layered structure. The layered structure arises from servers at one level making requests to servers at lower levels as a consequence of a request from a higher level. LQN was developed for modeling software systems, but it applies to any extended queuing network with



(a) HS/HA Architectural Pattern (b) LF Architectural Pattern

FIG. 2 – *Architectural Patterns from Schmidt et al. (2000).*

multiple resource possession, in which multiple resources are held in a nested fashion (Woodside et al., 1995).

The performance model used in this paper for the *ST* architectural pattern, shown in Figure 3, is used to show the key elements of a layered queueing network. Requests originate from layer 1, the "open arrival source" shown here, with a rate of $\lambda = 0.9$ and propagate down through tasks by way of entries. Entries accept requests, consume time on processors, and make requests to other entries. Service time demands to a task's processor are shown using square brackets and request rates are shown using parenthesis. Figure 3 also shows results from solving the model, for example, the service time for an entry, which includes the time blocked at lower level servers, the queueing delay for requests, the utilization of tasks and processors (labeled as μ in the figure), and the throughput at tasks (labeled at λ in the figure). The complete model is described in Franks et al. (2009).

3 Performance Models for the Architectural Patterns

This section presents layered queueing network performance models of the HS/HA and LFs architectural patterns. There are three goals for building the performance models: First, there is lack of rigorous comparison for these two architectural alternatives. Second, the performance models can be used to support performance planning and improvement and scalability analysis in the future. Third, each architectural pattern or alternative has its advantages and disadvantages. Performance modeling can help identify those areas to support decision making for self-adaptation.

For the performance analysis of the architectural patterns, a simple model is used which consists of application communicating with a database. To "improve performance", and to introduce a software bottleneck through blocking (Franks et al., 2006), the application and the database run on separate processors. The service time for the system was split evenly between the application and the database so that the utilization at the processors was balanced. Input to the system is modeled as an open stream, where the workload intensity is specified by an

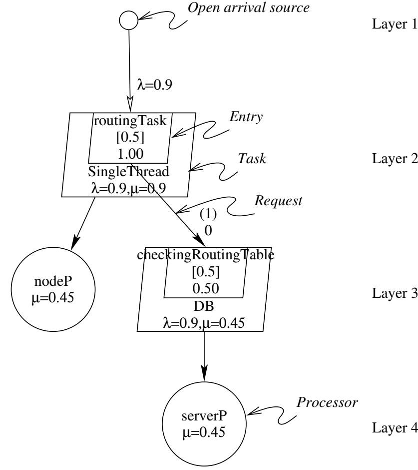


FIG. 3 – *Graphical Notations for Key LQN Elements.*

arrival rate that is independent of the system state (Menasce et al., 2004). To find the capacity of the system, its performance model is solved with increasingly large arrival rates. When a resource such as a task or processor saturates, i.e. its utilization approaches one, the system can no longer accept increased traffic. The performance model for this system using the ST architecture pattern is shown in Figure 3. Note that the bottleneck is the `SingleThread` task, and not either of the two processors.

3.1 HS/HA LQN Model

The HS/HA architecture queues asynchronous requests from the external environment into a common buffer which is read by a set of synchronous worker threads. The LQN for this architecture for the application described earlier is shown in Figure 4. In this model, the asynchronous service layer and the common queue are modeled using the task labeled `Buffer`. It is assumed that these buffers are retained until all processing is completed, so the `Buffer` task is modeled as a multi-server where the number of copies of the task, shown using braces, represents the size of the queue. The synchronous portion of the architecture (the worker threads in Figure 2a) is modeled using the multiserver task labeled `SynchThread`. Each instance of this task makes one blocking call to the database server labeled `DB`. It is assumed that there is insignificant overhead for buffer processing, so the service time of entry `store` is small.

Figure 4 also shows the results from solving the performance model using an arrival rate of $\lambda = 1.65$. Using the approach described in Franks et al. (2006), the bottleneck for this architecture is the `SyncThread` task, as its utilization is $4.94/5 = 0.99$.

3.2 LFs LQN Model

The LF architectural pattern employs a queue of threads which take turns servicing a `select()` call. The LQN for this architecture is shown in Figure 5. For this model, the *leader*

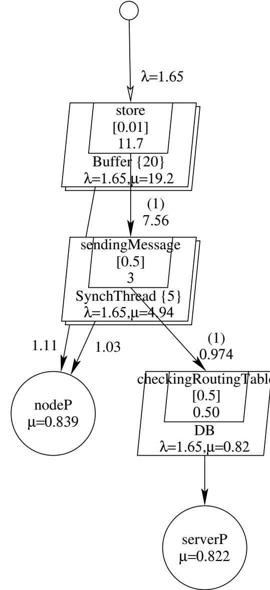


FIG. 4 – *Half Sync/Half Async LQN Model.*

thread is modeled using two tasks: `LeaderThread` and the first phase of the two-phase multi-server, `FollowerThread`. The *second phase* (Franks et al., 2009) of `FollowerThread` performs the actual work of the *follower* thread shown in Figure 2b. The service time for entry *gettingMessage* corresponds to the overhead of switching the `select()` from task to task in the LF architecture. The *phase change* of `FollowerThread` models event (2) in Figure 2b.

The results for solving this model using an arrival rate of $\lambda = 1.5$ are also shown on Figure 5. For this particular architecture and configuration, the bottleneck is caused by the processing by the `LeaderThread` task. While this task is almost fully utilized, its processor is not.

3.3 Performance Comparison for Three Architectural Alternatives Using LQN Models

Two performance metrics are of interest for this study. First, which architecture has the highest capacity, and second, which architecture has the lowest residence time for incoming events. Figure 6 plots the utilization of the bottleneck task for the three architectures. The ST architecture has the lowest capacity and the HS/HA the highest. Based on this metric alone, the autonomic system should always choose the HS/HA architecture.

The second performance metric of interest is the residence time for incoming events. For the HS/HA models, the events are stored in input buffers which are held for the duration of processing by the application and database. For the LF architecture, events are stored in the follower tasks so the leader thread is free to accept a subsequent event sooner. Figure 7 plots this result of the three models shown earlier over a range of arrival rates. For arrival rates less

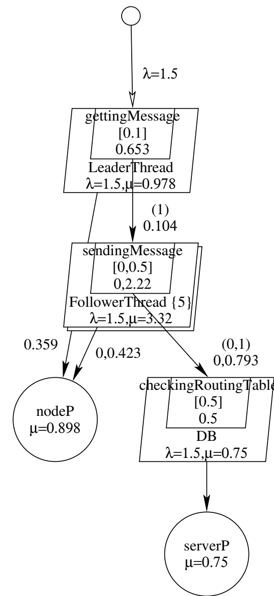


FIG. 5 – *Leader Followers LQN Model.*

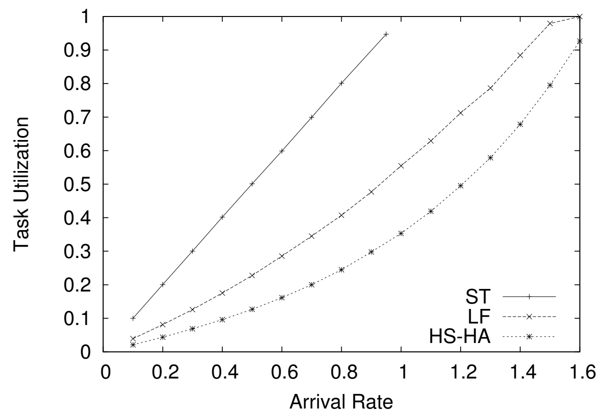


FIG. 6 – *Utilization Comparison of the Bottleneck Layers.*

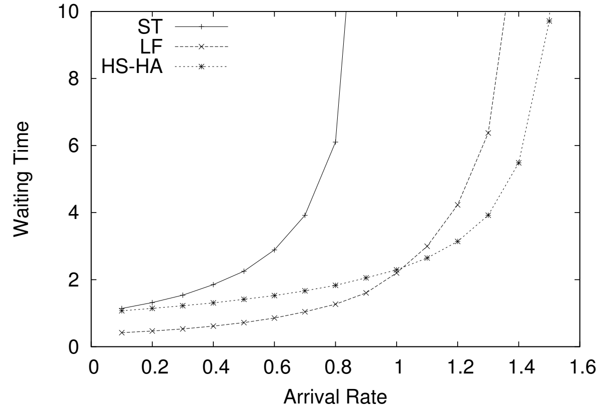


FIG. 7 – *Residence Time Comparison of Three Architectural Alternatives.*

than roughly 1.05, the LF architecture has a shorter residence time because the `select()` call can occur sooner. However, when the arrival rate is greater than 1.05, the overhead caused by the additional complexity of the LF pattern causes its residence time to exceed that of HS/HA.

Clearly, these results are only applicable for the parameters specified in this paper and by choosing different values for service times, other conclusions can be drawn. However, if the service times for the system are not stationary, and the alternative architectures exhibit the behaviour shown here, then an autonomic system would benefit from incorporating a performance model to allow it to choose the most optimum configuration at run time.

4 Self-adaptive Framework

The approach adopted in our research incorporates operational systems and software performance modeling into autonomic framework by providing analysis services to the autonomic manager. This section describes those components that are specific for self-adaptive support of SPE. Figure 8 presents the framework. The bottom layer is the generative framework and the top layer is the LQN solver. The operational generative system produces realistic performance data which are used for the simulation. In addition, the simulation results are validated against the performance measurements. The information is stored in the performance knowledge base to support quick performance estimation and self-adaptation at runtime (Osogami and Kato, 2007).

4.1 Self-adaptation with Performance Modeling

The minimal autonomic manager consists of the following main components: monitor, analyzer, planner, executive, interface sensor, effectors, and a knowledge base. The self-adaptive framework is built on top of the component-based generative system to gather the performance data and to support adaptation of software architecture via two interfaces. The framework in-

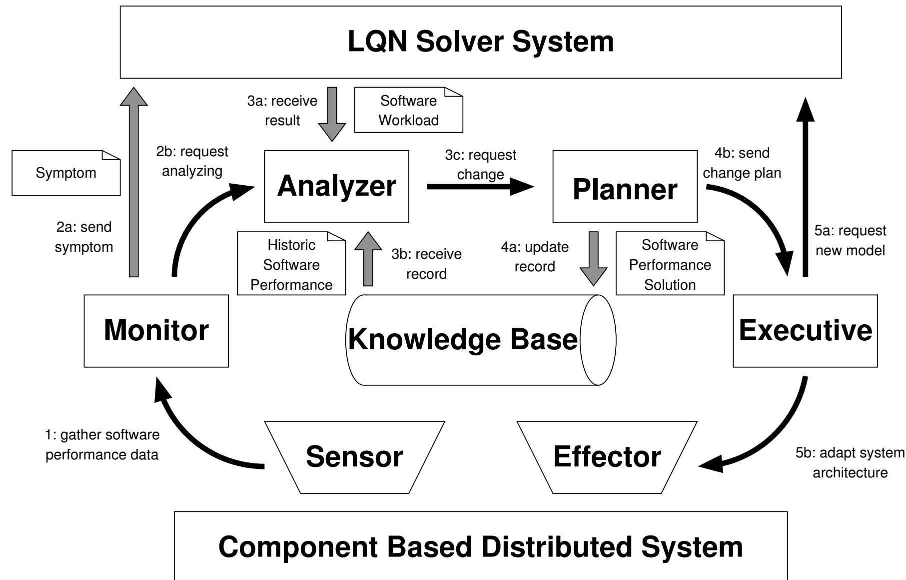


FIG. 8 – *Self-adaptive Framework.*

troduces the LQN solver that interacts with autonomic manager to dynamically analyze the software performance. The steps corresponding to the work flow shown in Figure 8 are:

1. The monitor component collects the software performance data from the generative system via the sensor interface. These data include information about those parameters of LQN performance models such as system throughput, the capacity of each software layer, buffer size, and the number of threads.
2. After the monitor component filters and aggregates these data, the information is passed to the LQN solver which generates the analysis result (2a). The monitor component also informs the analyzer component to do the performance analysis (2b).
3. The analyzer component receives outputs from the LQN solver (3a) and checks the related performance analysis result from the knowledge base that stores the historic system performance and simulated system performance (3b). It then generates the change requests and passes them to the planner component (3c).
4. The planner component creates a change plan based on the change requests and updates the knowledge base with new system architecture plan and predicted system performance (4a). It then sends the change plan to the executive component (4b).
5. Based on the change plan, the executive component informs the LQN solver to load the performance model corresponding to the new system architecture (5a). It also carries out the change procedure to the generative system via the effector interface (5b).

The performance analysis strategy requires the LQNs result and performance data from the knowledge base including modeling and historic results. From the control loop, the LQN results will be generated, and the performance analysis results will be stored in the knowledge

base at run time. To increase the accuracy of the analysis, a large amount of performance data should be collected and/or generated and stored in the knowledge base.

4.2 Design and Implementation of the Control Mechanism

The design and implementation of the control mechanism is facilitated by the generative framework. The main purpose of the generative framework was a proof-of-concept in raising the abstraction to the architecture level. With that framework, one architecture alternative is chosen and running at a time. To support autonomic computing, a middleware or a manager, `ArchitectureManager`, has been added to the generative framework. In the new design, one architecture alternative is at the running state, the others are on standby. The manager can dynamically switch from one architecture alternative to another based on the control policy. The manager can also disable the architecture switching feature based on user's decision.

The `ArchitectureManager` controls which architectural alternative will be selected. ST, HS/HA and LFs architectural alternatives are implemented by the `STPerformer`, `HS/-HAPerformer` and `LFSPerformer` components, respectively. They are derived from the previous generative framework and have been restructured for the self-adaptive framework.

Each architectural pattern has an associated buffer, i.e., `STInputQueue`, `HS/HAInput-Queue` or `LFInputQueue`. When the `ArchitectureManager` receives requests from clients, it will put requests into the buffer of the currently running architecture alternative, e.g., ST, HS/HA or LFs. During run time, one of the alternatives is active, the others are in the standby state. Once the `ArchitectureManager` receives a switching command from the Executive, it will activate the standby architectural alternative and put new requests into its corresponding buffer. The buffers are used to ensure that the requests stored in the buffer that have not been processed during architecture switching will not be discarded. The previously running architecture will be changed to the standby mode after it finishes processing all the requests stored in its own buffer.

By using the control mechanism, architecture switching delay is negligible and it does not affect the performance and robustness of the whole system. Because each architecture alternative has its own buffer, so the request lost at the sockets is avoided during the switching. We have manually tested the architecture switching between HS/HA and LFs every 10 seconds and there is no packet drops. Further, the architecture alternatives are decoupled from the generative system, so new architecture alternatives can be added into the system.

Currently, a simple control policy has been implemented for proof-of-concept. The policy is primarily threshold based as a result of the pre-configured queue length based on measurements or LQN performance analysis. With the threshold-based control policy, architecture switching takes place when a threshold-crossing occurs. Other control policies can be integrated into the framework. We are experimenting different policies, e.g., exponential averaging and fuzzy logic, to estimate near-future workloads and conduct self-adaptation.

5 Conclusions

This paper presented the idea of architecture-centric autonomic systems. We proposed a self-adaptive system that supports switching from one software architecture to another previ-

ously built architecture based on performance monitoring and modeling. The idea is facilitated with a generative framework.

Two well-known architectural patterns in this area have been documented, but no rigorous performance comparison has been conducted. To build a self-adaptive system, performance analysis is a key step. We have conducted experiments using operational systems and built LQN models to compare the performance of two common patterns to better understand the tradeoffs. Models have been constructed which show that different patterns have better performance under different traffic loads. This phenomena strengthens the argument for incorporating performance analysis into an autonomic control system.

Currently, a simple control policy has been implemented for a proof-of-concept autonomic system. The policy is primarily threshold-based. From our experiments with the self-adaptive framework, oscillations due to frequent switching do not seem to happen.

The approach also has the potential to improve reliability. It could support switch over in the event of a failure of one architectural alternative. When a failure occurs and is detected, the manager can switch to another standby system which is based on a different architectural design. If the active and standby systems are identical, as used in most cases in practice, the same failure may occur again (White et al., 2009). Therefore, the proposed approach may improve reliability, because different architectural alternative adopts different design policy.

References

Enterprise Managment Associates (2006). Practical autonomic computing: Roadmap to self-managing technology. Techical Report.

Franks, G., T. Al-Omari, M. Woodside, O. Das, and S. Derisavi (2009). Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering 35*(2), 148–161.

Franks, G., D. Petriu, M. Woodside, J. Xu, and P. Tregunno (2006). Layered bottlenecks and their mitigation. In *Proceedings of the Third International Conference on the Quantative Evaluation of Systems (QEST)*, Riverside, CA, USA.

IBM Autonomic Computing Architecture Team (2006). An architectural blueprint for autonomic computing. Technical report, IBM.

Kephart, J. O. and D. M. Chess (2003). The vision of autonomic computing. *Computer 36*(1), 41–52.

Lung, C.-H., K. Selvarajah, B. Balasubramaniam, P. Elankeswaran, , and U. Gopalasundaram (2006). Architecture-centric software generation: An experimental study on distributed systems. In *Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems (GPCE4QoS)*, Portland, OR.

Menasce, D. A., L. W. Dowdy, and V. A. Almeida (2004). *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall.

Muller, H. A., M. H. Klein, W. G. Wood, and W. O'Brien (2006). Autonomic computing. Technical Report CMU/SEI-2006-TN-006, Carnagie Mellon University, Pittsburgh, PA.

Osogami, T. and S. Kato (2007). Optimizing system configurations quickly by guessing at the performance. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, San Diego, CA USA, pp. 145–156.

Parashar, M. and S. Hariri (2007). *Autonomic Computing: Concepts, Infrastructure, and Applications*. CRC Press.

Schmidt, D. C., M. Stal, H. Rohnert, and F. Buschmann (2000). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, Volume 2. Wiley.

Smith, C. U. and L. G. Williams (2001). *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley.

White, J., H. Strowd, and D. C. Schmidt (2009). Creating self-healing service compositions with feature modeling and microrebooting. *International Journal of Business Process Integration and Management (IJBPIM) 4*(1), 35–26.

Woodside, C. M., J. E. Neilson, D. C. Petriu, and S. Majumdar (1995). The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers 44*(8), 20–34.

Woodside, M., G. Franks, and D. C. Petriu (2007). The future of software performance engineering. In L. C. Briand and A. L. Wolf (Eds.), *Future of Software Engineering FOSE 07*, Minneapolis, MN, USA, pp. 171–187.

Résumé

Les systèmes autonomes peuvent s'adapter à leur environnement et peuvent ainsi atteindre de haute niveaux de performance par le biais de reconfigurations en temps réel. Cet article présente une approche de conception d'architecture auto-adaptive ainsi qu'un exemple de son utilisation dans un cas réel simple. Cette approche est basée sur un système génératif qui comprend trois principales architectures logiciels (Single Thread, Half-Sync/Half-Async and Leaders-Followers), ainsi qu'un outil d'analyse de performance pour la sélection de l'architecture la plus adéquate en fonction de la charge soumise au système. Nous comparons la performance des trois architectures et les résultats nous permettent d'améliorer la compréhension de leur performance ainsi que de proposer des politiques de sélection d'architecture pour le système autonomes.