Software Design Pattern Analysis for Micro-services Architecture using Queuing Networks

Hanzhong Zheng, Justin Kramer, Shi-Kuo Chang
Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA
{victorzhz@cs.pitt.edu, jpk91@pitt.edu, schang@pitt.edu}

Abstract—Software design patterns are used to identify simple ways of realizing relationships among software entities or components for solving a commonly occurring problem. Design patterns allow the final software system to support different realized, non-functional requirements. In this paper, we are interested in three popular design patterns in Micro-services architecture: Fan (distributed), Chain, and Balanced, and study the influence of different system parameters to system performance. The simulation mimics system behaviors under specified design requirements for assisting software developers to select appropriate design pattern in software development life cycle (SDLC). In order to enable multi-pattern code generation, we extended our previous research on an automated modularity enforcement framework [1] from design pattern analysis to pattern evaluation.

Keywords – Micro-services, Design Pattern Analysis, Software Architecture Evaluation, Queuing Network Modelling

I. Introduction

Architectural patterns and design patterns are usually employed in the software development life cycle (SDLC). Software design patterns are "general and reusable solutions to a commonly occurring problem in software design within the context of software system design". It helps the developers to communicate software architectural knowledge, bypass traps and pitfalls during the development process [2]. Usually, design pattern can only provide the templates or descriptions to the developers about how to solve problems in the process of designing an application or system, but cannot be directly transformed into code. To ensure the continuous delivery of trustworthy and high-quality software systems while reducing the burdens on programmers, design patterns become critical in the software development process. The current approach on employing design patterns has been focusing on object-oriented software design with emphasis on the relationship and interactions between classes or objects [2] [3] [4]. There have been a lack of emphasis on the design pattern for service-oriented architecture, especially for Micro-services. Micro-services consider an application to be a collection of loosely coupled, interconnected modular services, where individual services communication through REST APIs, and lightweight messages.

In this paper, we put focus on three popular design patterns in micro-service architecture. A software system is usually divided into several modules during the design phrase. To explicitly enforce the modularity in design patterns becomes very important in software system design. We extend previous work (automated modularity enforcement framework) for those design patterns in Micro-service architecture. The formal definitions of three top design patterns are [6]:

Chain: A "pipeline" layout of all components in the execution process. Clients establish one-to-one relationship with servers.

Fan (Distributed): All clients establish many-to-one relationship with a central database server. All the information will be stored into the central server.

Balanced: also known as Shared Data Pattern. Each client establishes one-to-one relationship with its database server. Multiple servers share their data.

We have conducted a comprehensive simulations using Queueing Network Modelling based tool, named Java Modelling Tool (JMT). The simulation results indicate that each design pattern has its own advantages for building appropriate software system products for satisfying proposed design requirements. The contributions of this paper are as followings:

1. We developed the automated design pattern analysis in Micro-service architecture from system pattern design to pattern evaluation.
2. We build a mathematical estimation model through parameterizing the expected cost for software system development
3. We conducted comprehensive simulation experiments and analysis for better understanding of each design pattern's properties.

DOI Reference Number: 10.18293/SEKE2021-180

II. Related Work

Micro-services Architecture. Micro-service architecture is a type of service-oriented architectural type, in which an application is constructed as a set of loosely coupled small services. The current practice of micro-service architecture mostly concentrates on the business enterprises such as Netflix, Amazon, e-commerce, etc. The next generation of micro-services requires ad-hoc tools for the creation of design patterns [5]. Sahiti Kappagantula introduces the several design patterns in Microservices architecture for providing reusable solutions to overcome common problems and improve application performance [6]. However, it is lack of the instructive and qualitative analysis among different patterns. In our work, we aimed to provide comprehensive analysis for different design patterns based on the performance quality attributes under a specific application scenario.

Queueing Network Modelling. Software architecture evaluation ensures an appropriate architecture is chosen for building complex software-intensive systems to any organization. Hence, architecture evaluation helps developers to ensure all stakeholders' requirements have been satisfied. Commonly, software architecture evaluation can be classified into experience-based, simulation-based, mathematical modeling, and scenario-based [7]. The simulation-based evaluation approach usually is combined with mathematical modeling for estimating a more accurate system performance. However, it is not easy for evaluating system performance during the design process. Queueing Network Modelling represents the computer system as a network of queues, and analytically evaluates the system performance [8]. It can simulate a group of service centers, which can make use of our three different patterns. Macro Bertoli et al. presented Java Modelling Tools (JMT) suite for evaluating system performance using queueing models [8]. JTM integrates a graphic user interface and other methodologies such as discrete event simulation, bottleneck identification in multiclass environment, etc. We simulated the performance results of our three different design patterns using JMT with the same initial parameters setting.

III. Mathematical Modelling Optimization

The software cost estimation not only can minimize the total cost of software development cost, but it also ensures the final product can satisfy the requirements, which generally refer to the quality attributes such as performance, functionalities, etc. Many estimation models have been developed and widely used. In general, there are two major categories of existing models: algorithmic and non-algorithmic.

Algorithmic cost modelling uses mathematical expressions to predict the development costs based on the estimations of system size, complexity, and other process and product factors. Finding the most appropriate expression can estimate software development costs, which are important for analyzing the performance of our three different structural patterns with keeping a relatively low development cost. The general form of an algorithmic cost estimation can be expressed as:

$$E[effort] = A * S^B * M \tag{1}$$

where A is constant factor that depends on the type of final software product, S is the code size of the software or functionalities of certain components, B is the exponential factor that usually lies in range of [1, 1.5], indicating the fact that costs do not linearly increase with project size, and M is a constant multiplier for combing process such as dependability requirements. Our automatic code generation tool is built based on Micro-service applications. The service reliability is critical for service-oriented system. We added a reliability modeling term to the cost estimation expression as following:

$$E[effort] = A * S^B * M + \sum_{i=1}^{n} (1 - \frac{f_i}{t_i}) * c_i + \epsilon \tag{2}$$

where the term $1 - \frac{f_i}{t_i}$ is the simplified probability model of estimating service reliability from, f_i is the number of services executions without exceptions occurrence; t_i is the total number of service invocations. c_i is the anticipated cost of executing service. During the simulation, we assume that the probability of software system failure as a stochastic process. In most logarithmic cost models, the code size (S) is usually difficult to estimate when the specifications are not available. Since factors B and M are usually subjective, we are mainly interested in their relations to the cost estimate model during the optimization process. To study the relation between M and $E[effort]$ we take the partial derivative with respect to M:

$$\frac{\partial E[effort]}{\partial M} = A * S^B \tag{3}$$

Similarly, we take the partial derivative with respect to B:

$$\frac{\partial E[effort]}{\partial B} = A * M * S^B * ln(S) \tag{4}$$

Therefore, in the simulation experiment section, we study the changing of M or B variables, while keeping other variables to be constant to determine their effects to the cost and system performance.

IV. Experiment Tool

we extend our previous work by adding extra functionalities and input parameters to support the three patterns. This allows software developers select their preferred architecture after testing the performance of each design pattern. Our key objective was to add a layer of experimentation and design to our automatic code generation. We added a cleaning functionality to AutoGenerator, which allows users to experiment with combinations of design patterns and service creation without the risk of damaging their templated product. Therefore, our 'cleaning' function provides users with flexibility in their designs and improve software development process. After we established our 'cleaning' functionality, we started on the creation of each micro-service architecture design pattern. The first stage of production was analysis of each design pattern. The results of our design pattern analysis led to formulated architecture diagrams. In Fig. 1, we demonstrate the balanced design pattern for our micro-service architecture in medical application. In the architecture diagram, a balanced design pattern is followed by combining the fan and chain design patterns. In the figure 1, the doctor and patient services act as examples of the fan design pattern. The doctor and patient services connect to our centralized registration service through HTTP communication via their localized micro-service servers. Within the fan design pattern, each micro-service will contain its own database, server to communicate with the central registration server, and controller layer. Also, the prescription service acts as an example of the chain design pattern. In the chain design pattern, our AutoGenerator establishes the formation of a new micro-service consisting of a service layer and a database. The chain design pattern demonstrates the pipeline execution pattern, so prescription service acts as an addition to the functionality of the fan structure's patient service. In this example, the patient service will query the prescription service directly through a public interface to acquire information about the prescriptions of a specific patient. IServices are assigned a unique probability to appear as micro-services within the fan or chain design patterns. The element of probability in the balanced design pattern allows for permutations of samplings to appear as the result of the auto generation. Thus, basic analysis of each design pattern allowed for translation into a micro-service architecture for our AutoGenerator.

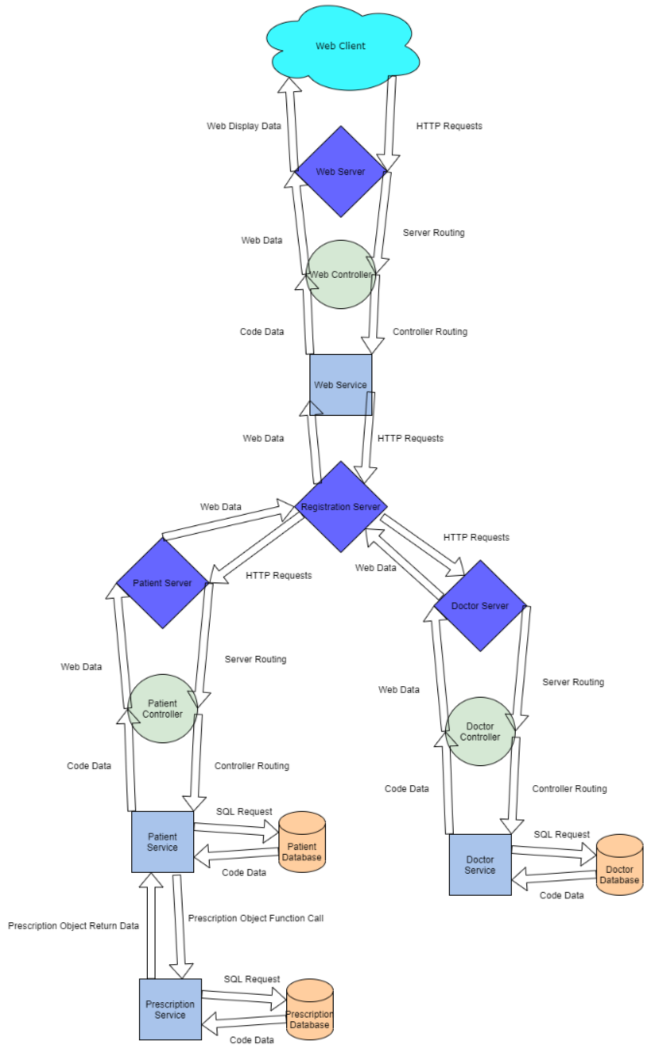

Fig. 1: A balanced micro-service architecture utilizing both chain and fan pattern design.

Our process begins with IC cards, which define the service interactions witnessed in the architecture being designed [1]. Once the IC cards are defined, the ICMS can output an XML specification as shown in Fig. 2. The XML specification specifies the structure of different software components and initial system parameters, which all are used in the system simulations.

```
<?xml version="1.0" encoding="UTF-8"?>
<icCardList
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <icCardEntry icEntryId="2788" icEntryName="ex3">
```

```
<icCard icId="10265" icName="drug_drugs_service" icDescription="Serves information about the drug" icIntPattern="quietstate" icMyTask="Serve information about the drug " icTimeCriticalCondition="&lt; 30 minutes and Begin_Table T_DRUG(number, name, address) End_Table" icNumberCurrent="1" icNumberTotal="1">
    …
  </icCardEntry>
  <configOptions>
       <arrivalRate value="Normal"/>
       <queueingDiscipline value="FCFS"/>
       <routing value="RoundRobin"/>
       <serviceTimeSeconds value="50"/>
       <serviceDemandSeconds value="30"/>
       <designPattern value="fan"/>
       <userPreference value = "Chain">
    </configOptions>
</icCardList>
```

Fig 2. The XML specification for the architecture structure.

Based upon the XML specification of the IC cards, the AutoGenerator can then create the output modules. Fig. 3 demonstrates a portion of the code to generate these modules, which allows for several key variations. The first variation is the parameter that designates the design pattern to utilize. The input parameter may be selected as “balanced”, “fan”, or “chain” depending on the use case. Furthermore, each design pattern follows a set of generation rules which establishes modularity and provides a basis to fill in implementation details for the architecture. For the chain pattern, our AutoGenerator builds a simplified micro-service layer composed of the necessary modules to connect a Micro-service to its database and to another Micro-service. The necessary modules are comprised of a data repository interface and a method-layer interface to interact with the service. For our balanced pattern, we follow the directed creation of a fan or chain implementation of the auto generation based upon probability. The same skeleton of our AutoGenerator now contains flexibility in its design pattern and the ability to enable hypothesis testing through cleaning function.

```
for i, service in enumerate(services):
    if tables[i]:
        separated_table = tables[i].split('(')
        adjusted_columns =
adjust_columns(separated_table[1])
        TABLE_NAME = separated_table[0]
        TABLE_COLUMN_1 = adjusted_columns[0]
        TABLE_COLUMN_2 = adjusted_columns[1]
        TABLE_COLUMN_3 = adjusted_columns[2]
    NEW_SERVICE = service
    NEW_SERVICE_PLURAL = services_plural[i]
    if DESIGN_PATTERN == 'balanced':
        choices = ['chain', 'fan']
        choice = random.choice(choices)
        if choice == 'chain':
            run_generation_chain(SRC_PATH,
NEW_SERVICE_PLURAL)
        else:
            run_generation_fan(SRC_PATH,
NEW_SERVICE_PLURAL)
    elif DESIGN_PATTERN == 'chain':
        run_generation_chain(SRC_PATH,
NEW_SERVICE_PLURAL)
    elif DESIGN_PATTERN == 'fan':
        run_generation_fan(SRC_PATH,
NEW_SERVICE_PLURAL)
    else:
        print("Please enter a valid design pattern!")
```

Fig. 3 Code to generate each of the architecture structures.

V. Queuing Network Modelling Analysis

We decide to use Queuing Network simulation for evaluating the different patterns’ performance. The layout designs of the patterns are illustrated in Fig. 4 using the JMT simulation software. All the simulation layouts reflect the definition of different design patterns in the beginning of the paper. JMT contributes to perform system evaluation studies in the following two ways: 1. Statistically analysis such as confidence interval analysis, variance estimation, etc. 2. A friendly user interface for the description of system and parameters analysis. The main parameters related to the interested variables: system size (*B*) and multiplier of system combining process (*M*). Since we assume that the micro-service design patterns are targeted to medical application, so that *M* can be reflected the requirements of patients’ service demand time, and *B* is the doctor service time. The *S* is the application that handles the number of patients in the system evaluation (Arrival rates): Exponential, Normal, Uniform, etc. In this paper, we use the normal distribution for better modelling the number of patients in real case. Service Demand: the average amount of time (workload) that each user/patient required for the doctor’s service (on user/patient side). Queueing disciplines: 1. Non-preemptive: First Come Frist Served (FCFS), Last Come

First Served (LCFS), Random (RAND), etc. 2. Preemptive: Server sharing, Discriminatory Server sharing, etc. Routing of the users/patients in the system: the current setting is the Round Robin, which simulates that there is a waiting room for the patient to visit doctor in our system. Service Time: the maximum amount of time that doctor to diagnose each patient (on doctor side). User Preference: the developers' inclining towards to specific structure patterns. This allows the user to select its own desired patterns.

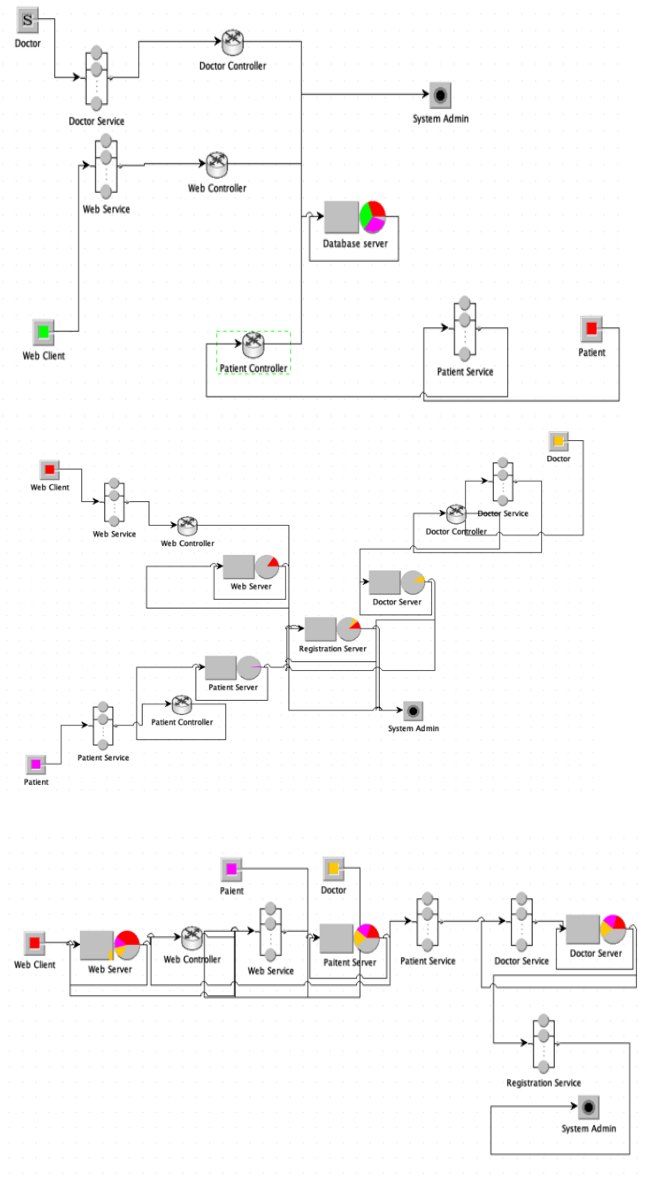

Fig. 4 The top, middle, and button figures are the simulation layouts for distributed, balance, and chain pattern.

We converted the design of patterns into well-structured XML code to input of simulation with specified components layout and system parameters. The simulation settings are: (1) The number of users is continuously increasing; (2) All the servers have the limited amount of disk capacity; (3) Each user submits the jobs according to a normal distribution with parameters different parameters; (4) The evaluation metrics are Throughput (# of jobs /second), Queuing Time (sec/user), Response Time (second/job), System utilization (# of working jobs/second); **(**5) Each simulation lasts until the model converges and we conducted 15 repeat runs making sure the accurate final performance results of each pattern. The experiment results are fall into with relative error < 0.03. Our purpose of the simulation is to find an appropriate structural design pattern, which plays a critical role in software development process. We alternate different parameters for different patterns during the simulation. Figures 5, 6, and 7 show the performance results of the 3 different patterns under the different simulation inputs. The alternations of the desired variables enable to reflect the behaviors of different patterns. The simulation can see the fluctuations of different patterns under the different parameters' setting. This allows the developers to decide appropriate design pattern during the development process.

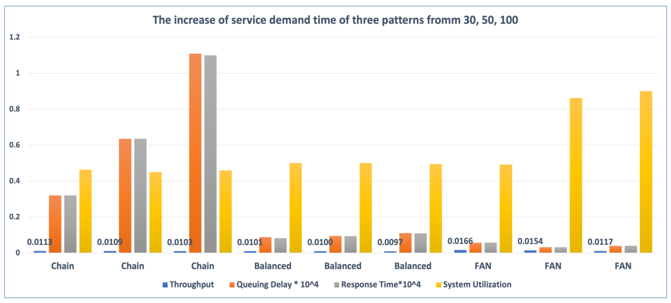

Figure 5: the performance of three patterns with the increase of service demand from 30, 50, 100.

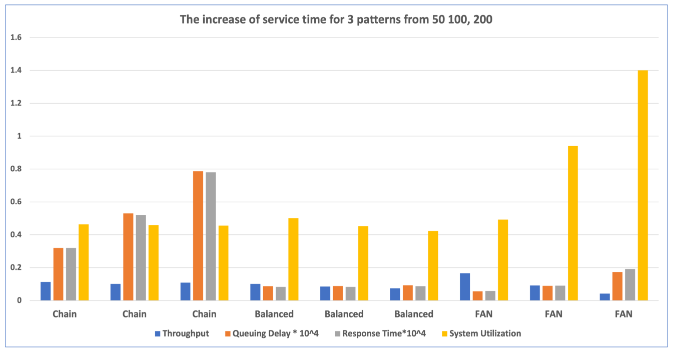

Figure 6: the performance of three patterns with the increase of service time from 50, 100, 200.

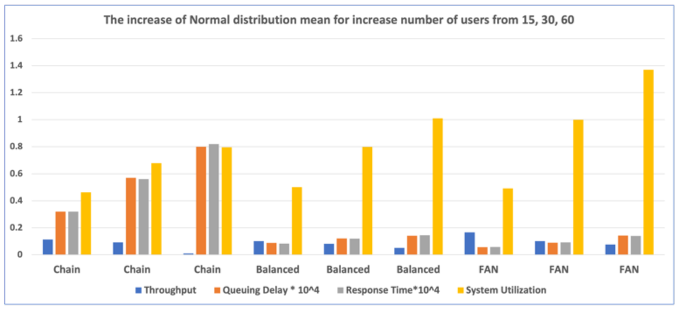

Figure 7: the performance of three patterns with the increase of normal distribution mean (15, 30, 60) for modelling the increase number of users in the system.

We treat each user is managed by a process in the system, which is related to the system requirements in operating and combing user processes. The simulation results show that the throughput of all patterns decreases with the increase of all the parameters. The FAN pattern largely impacted by the service time and the Chain pattern largely influenced by the number of users in the system. The FAN pattern has a higher and higher value in the System utilization, which represents that there are increasing number of working jobs in the system. The increase of the service time surprisingly decreases the system utilization of Balanced pattern, while the rest of two patterns both increase. Our explanation is the routing problem due to the number of data servers and components. Since there are multiple servers in the Balanced pattern, it brings the higher capacity to handle the dramatically increase of users in the system. However, the data replication and synchronization become the main challenge in this design pattern.

As for the FAN (distributed) pattern, it maintains a comparatively reasonable performance under different parameters setting, but it can easily be influenced by the alternations of parameters. The centralized data server avoids problems in other two patterns but requires a more intelligent routing and queuing discipline for handing users in "burst" situation. The choice of different design patterns depends on the design requirements and also takes the user's preference into the consideration. The Chain pattern involves the structure of "pipeline" design. The execution of each process is strictly followed the order, which unavoidably cause the "stalls" inside the execution pipeline. However, certain applications such as online patient diagnosis in medical domain has the preferences on chain pattern.

VI. Conclusion and Future Work

This paper proposes a new software design approach using the Micro-service architecture. The extended automated code generation framework enables code generation under three different design patterns. We compared our design patterns using Queuing Network modelling for performance analysis. The queuing network allows for analytic study on the software system, which is represented as a network of queues with collections of service centers. We compared the performances of the different design patterns under different parameters settings and provided an analytical evaluation for them. Our next goal is to study the influence of parameters (such as the capacity of the servers, data usage volume, user preferences of specific patterns and so on) to the performance of different design patterns.

References

[1] H. Zheng, J. Kramer, and S. Chang, "Auto-Modularity Enforcement Framework Using Micro-Service Architecture", in Journal of Visual Language and Computing, pp. 17-22, 2020.

[2] H. Mu and S. Jiang, "Design patterns in software development,"2011 IEEE 2nd International Conference on Software Engineering and Service Science, pp. 322–325, 2011

[3] P. Kuchana, Software architecture design patterns in Java. Auerbach Publications, 2004.

[4] S. Jiang and H. Mu, "Design patterns in object-oriented analysis and design," in 2011 IEEE 2nd International Conference on Software Engineering and Service Science, pp. 326–329, 2011.

[5] L. Safina, M. Mazzara, F. Montesi, and V. Rivera, "Data-driven workflows for microservices: Genericity in Jolie," in 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA),pp. 430–437, 2016.

[6] Kappagantula, S. (2020, November 25). *Everything You Need To Know About Microservices Design Patterns*. Edureka. https://www.edureka.co/blog/microservices-design-patterns#Database

[7] M. Svahnberg, C. Wohlin, L. Lundberg, and M. Mattsson, "A method for understanding quality attributes in software architecture structures," in Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE '02, New York, NY, USA, p. 819–826, 2002.

[8] M.Bertoli, G.Casale, G.Serazzi. "JMT: performance engineering tools for system modeling," ACM SIGMETRICS Performance Evaluation Review, Volume 36 Issue 4, New York, US, 10-15, 2009.