Variability Analysis for Communications Software

Chung-Horng Lung
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
chlung@sce.carelton.ca

Introduction

Most software systems contain areas where behavior can be configured or tailored based on user objectives. These areas are referred to as variation points. Management of variability becomes more and more important, because it is closely related to software reuse, object-oriented design frameworks, domain analysis, and software product lines. Software variability is the ability of a software system that can be changed, tailored, or configured for specific use in a particular environment. Variability management is recognized as a critical concept in software engineering. Successful management of variability can shorten development time and lead to more flexible and better customizable software products.

Generally, the main reason for software variability management is to support reuse in a product families. Variability management could range from more formal approach based on mathematical models [Lung94], systematic methods like domain analysis, to simple programming support, e.g., inheritance in object-oriented programming languages or the #ifdef compiler directive. This paper, however, studies variation points and software variability from the performance point of view. Specifically, this paper deals with analyzing and building a framework for communications software for routing applications with an aim to support detailed software performance evaluations.

There are many possible alternatives for concurrent and networked software. Schmidt et. al, [Schmidt00] captured and documented a set of design patterns for this area. The book discussed alternatives in details. However, it is often still difficult to make concrete evaluation or objective tradeoff analysis based on patterns from the performance point of view due to the details we need in performance evaluation.

This paper studies various variation points for communications area. The study will be used to build a generative framework. The framework will be studied together with software performance engineering techniques, layered queuing networks (LQNs) [Woodside95], to characterize performance aspects for various approaches. The approach will provide useful guidelines for the users to choose an appropriate model or design to meet their performance requirements.

Problem Description and Approach

In distributed applications, there exist many variations. For example, there are client-server model and peer-to-peer model. For each model, there are further variations depending on specific applications and requirements, typically scalability, performance, and portability. For instance, for a server design, we may adopt a straightforward Reactive design pattern. However, the approach often leads to scalability concern. This can be improved using either Half-Sync/Half-Async or Leader/Follwers pattern. For a design pattern like Half-Sync/Half-Async, there still exist further variants, as discussed in [Schmidt00].

The design patterns document general guidelines and principles for building software systems. However, for some applications, we need deeper understanding and more detailed analysis. A simple example is demonstrated here. Figure 1 illustrates the structure of the Half-Sync/Half-Async pattern. It is easy to identify a simple variation point, which is number of worker threads in the thread pool. The number can be easily configurable. Yet, from the performance perspective, it is difficult to determine the number of threads that will provide the best result. The most commonly adopted approach in industry is measurement, because there are many implementation and platform specific details involved.

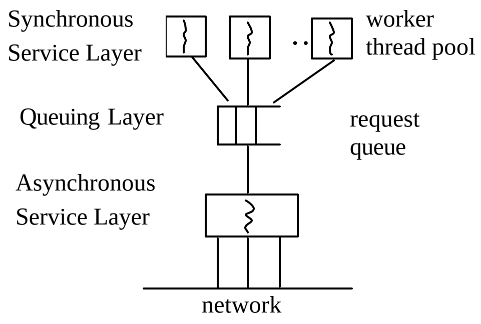

Figure 1. Structure of the Half-Sync/Half-Async Pattern

Another variation point that is more difficult to deal with is the number of request queues. Multiple queues provide more flexibility to support QoS (Quality of Service), but we need a scheduling policy to retrieve data from those queues. Moreover, we need to consider if it is better to have a dedicated thread for each request queue than a tread pool.

An even more difficult tradeoff analysis is to determine an appropriate design pattern or structure. The Leader/Followers pattern can also be used as an alternative for concurrent and networked software. There are advantages and disadvantages for each approach. Schmidt et al, [Schmidt00] discussed those issues. However, there are many questions need to be answered in order to derive an objective tradeoff analysis. On the other hand, it is almost impossible in practice to develop several alternative designs and perform thorough evaluations for each of the alternative due to resource constraints and competitions.

The main idea of this paper is to actually develop some typical alternative designs and conduct thorough performance analysis and characterization for each design. Hands-on experience is critical in building a useful framework. The process will help identify concrete variation points and the results will be useful in predicting performance and building a generative framework to support future system development.

The focus of this project is on network router software. One of the main functions of a router is to route and forward data packets. However, many features or requirements are related to data routing and forwarding. For example, there may be different levels of QoS requirements. Each level may need a separate queue associated with a queuing mechanism. Each level of traffic may also need to be policed differently based on pre-defined policy. Even for the same level of QoS, there exist different approaches.

We did not build a system from scratch; instead, we obtained a router software system from industry. The original design of the software was similar to the Reactive pattern as shown in Figure 2. The software process contains a main thread. When a router receives a packet from the network, the packet is stored in a kernel buffer. The main thread will then read packets from the buffer and process them and put them in a destination queue. There is a dedicated thread for each destination queue to forward the packet to an adjacent router. The select () function is used to demultiplex a set of socket handles.

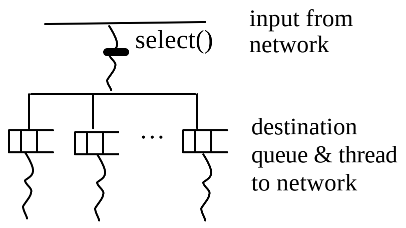

Figure 2. The Structure of a Router Software Process

Figure 3 illustrates our initially modified design based on the Half-Sync/Half-Async pattern. The software process now contains several threads. Multiple threads cannot use the select function concurrently to demultiplex a set of socket handles because the operating system will erroneously notify more than one thread calling the select function when I/O events are pending on the same set of socket handles [Steven98]. Therefore, there is only one thread for this layer to properly read data from the network. The asynchronous layer reads data packets from the network and stores them into an appropriate queue, depending on the data type. There are several worker threads in the synchronous layer. The number of worker thread is configurable. Currently, the number of input queue is static, because there are two types of data packet. The number of input queues, however, can be changed. Moreover, a scheduling algorithm is needed among multiple queues. The scheduling policy is another point of variation.

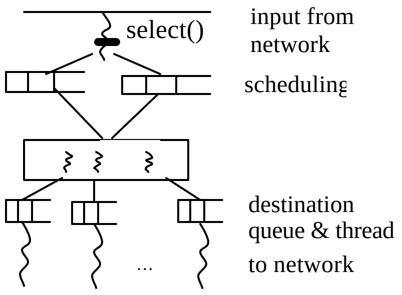

Figure 3. An Alternative Design based on the Half-Sync/Half-Async Pattern

Work in Progress

Currently, we are in the process of building another alternative design based on the Leader/ Followers pattern as diagrammed in Figure 4.

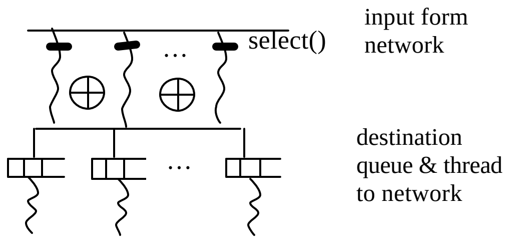

Figure 4. An Alternative Design Based on the Leader/Followers Pattern

In this design, multiple threads coordinate themselves. Only one thread at a time – the leader – waits for an event to occur. Other threads – the followers – can queue up waiting for their turn to become the leader. After the leader detects an event, it promotes one follower to be the leader. It then becomes a processing thread [Schmidt00].

The main reason that we choose to convert the original router system to the Leader/Followers pattern is that the model adopts a different design principle that is closely related to performance. By doing it, we will identify more variation points, which will provide valuable lessons in building the framework. Moreover, this design will help us better understand related performance issues.

We are also considering other alternatives. Figure 5 illustrate some examples.

We are also investigating issues associated with notation and evolution. Evolution is more complex and may be problematic for a generic system that is not well represented and designed.

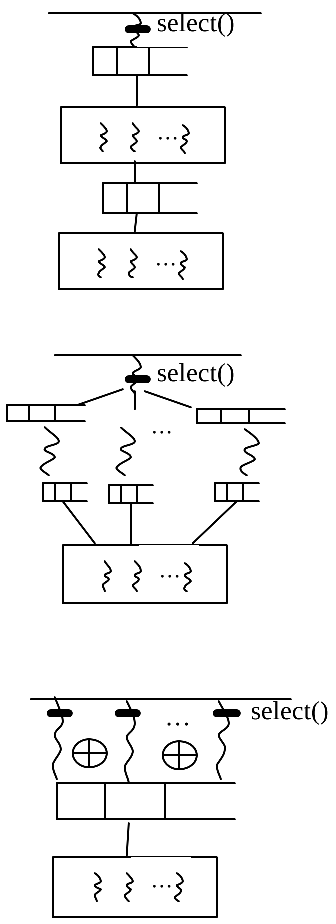

Figure 5. Other Alternatives: an Example

References:

[Lung95] C.-H. Lung, J. Cochran, G. Mackulak, and J. Urban, "Computer Simulation Software Reuse by the Generic/Specific Domain Modeling Approach," *Int'l J. of Software Eng. and Knowledge Eng.*, vol. 4, no. 1, pp. 81-102, 1994.

[Schmidt00] D. Schmidt, M. Stal, H. Rohnet, and F. Buschmann, *Pattern-Oriented Software Architecture, vol. 2,* John Wiley & Sons, 2000.

[Stevens98] W.R.Stevens, *Unix Network Programming, Volume I: Networking APIs: Sockets and XTI, 2nd Edition*, Prentice Hall, 1998.

[Woodside95] C.M. Woodside, "A Three-View Model for Performance Engineering of Concurrent Software", *IEEE Trans. On Software Eng.*, vol. 21, no. 9, pp. 754-767, Sept. 1995.