EXPERIENCE OF COMMUNICATIONS SOFTWARE EVOLUTION AND PERFORMANCE IMPROVEMENT WITH PATTERNS

Chung-Horng Lung, Qiang Zhao, Hui Xu, Heine Mar, Prem Kanagaratnam
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
chlung@sce.carleton.ca

Abstract: Software evolves as requirements or technologies change. Tremendous efforts are often needed to support software evolution as evolution may involve reverse engineering and subsequent restructuring or forward engineering. Design patterns have captured great attentions as they provide rapid transfer of proven solutions. The paper presents an experimental study of applying design patterns to restructuring in communications software. The restructured software not only satisfies the new functional requirements, but also increases the performance. The paper demonstrates the benefit by showing concrete performance results to support the improvement.

Keywords: software evolution, restructuring, networks, design patterns, software performance engineering, quality of service, modeling

1. Introduction

Software architecture has become both a theoretically and practically important topic recently because of the increasing complexity of software systems. Software architectures also play a critical role for change impact analysis [1]. It is common in practice to reconstruct architecture from the existing design or implementation, and modify or restructure the system to meet new changing needs. The process is referred to as reengineering. The need for software reengineering has increased significantly, as heritage software systems have become obsolescent in terms of their architecture, the platforms on which they run, or their suitability and stability to support evolution.

The new changing needs that trigger reengineering can be either functional or non-functional requirements [5]. Examples of non-functional requirements include performance and maintainability. Performance is critical for computer systems and networks, as it affects real-time accuracy and system scalability. Computer systems and networks are crucial in today's technologies, because many applications are dependent on computer systems and networks in the Internet age. Evaluation of computer systems and networks is needed at every stage in the life cycle of the product including design, implementation, marketing, use, upgrade, tuning, and etc.

MPLS (Multiprotocol Label Switching) has been recognized as a fundamentally important technology in networks [3]. MPLS has the potential to bring benefits to IP-based networks, including traffic engineering and quality of service (QoS). cgNet is a traffic controller based on MPLS. It was developed as a prototype software-based router to test the feasibility of real-time traffic engineering based on MPLS, but could also be used as a general-purpose test network. Later, there is a need to improve the performance and to support new QoS features for further research. Hence, the first objective of this paper is to reengineer cgNet to satisfy both the functional and non-functional requirements.

In addition to the traffic engineering, cgNet involves distributed computing and protocols. The areas related to protocols and traffic engineering are relatively new. However, from the software architecture perspective, cgNet shares commonalities with many systems in distributed computing and concurrent systems. Many critical concepts in this area have been captured and documented with design patterns [14].

Design patterns capture recurring structures and dynamics among software participants to facilitate reuse of successful designs. Patterns, generally, codify expert knowledge of design constraints and "best practices". Pattern languages define a vocabulary for talking about software development problems, provide a process for the orderly resolution of these problems, and help to generate and reuse software architectures [2,4,14].

Design patterns can be used to support software development, reuse, and maintenance. However, can design patterns improve or degrade performance? Performance, in general, is not directly dependent on design patterns. In other words, the answer is largely depends on where the performance bottlenecks are and how patterns are actually implemented. However, performance can benefit from patterns is concurrency and locking patterns [13]. These patterns tend to have a broad influence on application performance.

The second objective of this paper is to study those well known design patterns in concurrent and networked domain and apply them to cgNet for restructuring. The

restructured system needs to support QoS requirements and to have better system performance.

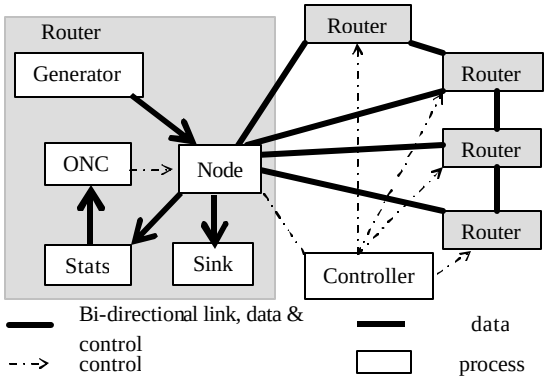
The remaining of the paper is organized as follows. Section 2 provides a brief overview and the original structure of cgNet. Section 3 demonstrates the restructured cgNet based on design patterns. The enhanced system supports additional QoS requirements. Section 4 presents performance evaluation of the restructured system. Section 5 presents lessons learned from reengineering of software with patterns. Finally, Section 6 is the summary of this paper.

2. Overview of cgNet

Figure 1 demonstrates the software structure of cgNet. cgNet is composed of a network of software-based routers, each consisting of the following main components:

- Node: A node process represents a router or a switch. It forwards traffic to another node or to its associated sink. Each node can also distribute flows across multiple paths based on the bandwidth ratio.
- Generator: A generator is a payload traffic source. It generates data packets to various destinations based on pre-configured distribution.
- Sink: A sink is a destination for generated traffic.
- Statistics (Stats) sink: A statistics sink is the destination for those statistic reports that are generated periodically by its associated node process. This information is then made available to the traffic controller to manage traffic.
- Traffic controller, ONC: ONC is used to manage local traffic. It periodically interprets network status from network statistics and formulates the necessary changes required to improve network performance.

Figure 1. Software structure of cgNet



The main routing or switching functionalities are realized in the node process. Figure 2 illustrates the structure of the original node process. The node process contains multiple threads: a main thread, a stats thread, and a thread for each destination. The node process communicates with other nodes. The Reactor design pattern is used to demultiplex multiple incoming sources.

Figure 2. Recovered structure of the Node process

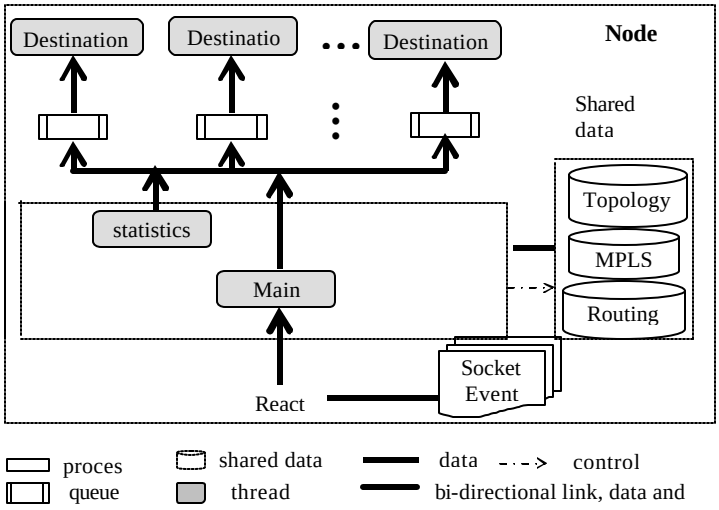
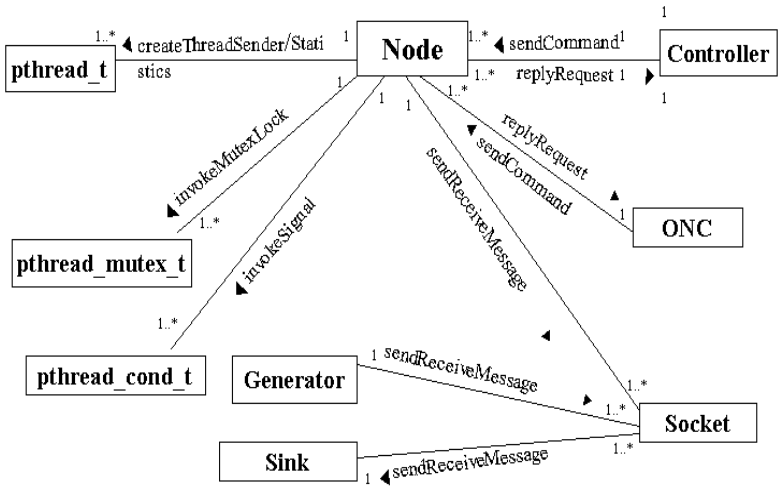


Figure 3 depicts the collaboration diagram. The collaboration diagram demonstrates detailed interactions for the main components.

To improve software performance, we adopt the software performance engineering approach [7,15]. The next step after we identify the software structure is to identify frequent workload scenarios and their execution paths. The most critical scenario from the performance perspective is data forwarding. Hence, we focus on it for performance evaluations.

Figure 3. Partial collaboration diagram of cgNet components



For the original design, the processing of the incoming messages requires several steps. In the mean time, there are still many messages that are waiting in the socket queue to be read. The main thread, hence, is a performance bottleneck. Intutively, if there is a thread that reads messages from the queue and another thread that does the processing concurrently, the performance should be improved. The next section discusses the restructuring effort for performance enhancement of the design.

3. Restructuring with Design Patterns

This section presents the restructuring effort of cgNet based on design patterns. Design patterns have drawn a lot of attentions lately. Concurrent and networked applications have been studied intensively. Recurring structures and dynamics among software components for this area have been captured and documented. Schmidt at al. [14] give an in-depth treatment of how to deal with those systems design issues in a systematic way.

Instead of starting from scratch to find solutions, we reuse some well-known patterns to begin with. We adopt the Half-Sync/Half-Async pattern as the overall architectural structure for the node process. Half-Sync/Half-Async pattern separates asynchronous and synchronous service processing by introducing layers in the structure. Asynchronous programs are generally more efficient, especially in globally distributed computing. Synchronous processing, on the other hand, is usually less complex, since services can be locally constrained to follow a sequence of operations. Queuing layer is used in between these two layers to mediate the communication [14].

The main reason is its resemblance to the original design and its features just described. Figure 4 demonstrates the restructured view of the node process. In the new design, the original main thread is divided into three main layers: asynchronous, queuing, and synchronous, as documented in the pattern. An input thread will read incoming messages into a queue, from which the worker threads can take and process them concurrently. This way, the messages will not need to wait in the socket queue until a message is done processing.

Figure 4. Restructured view of the Node process

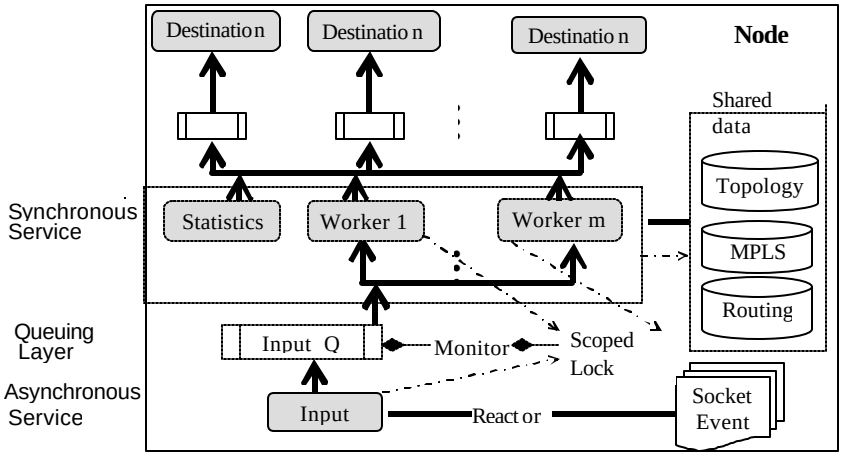
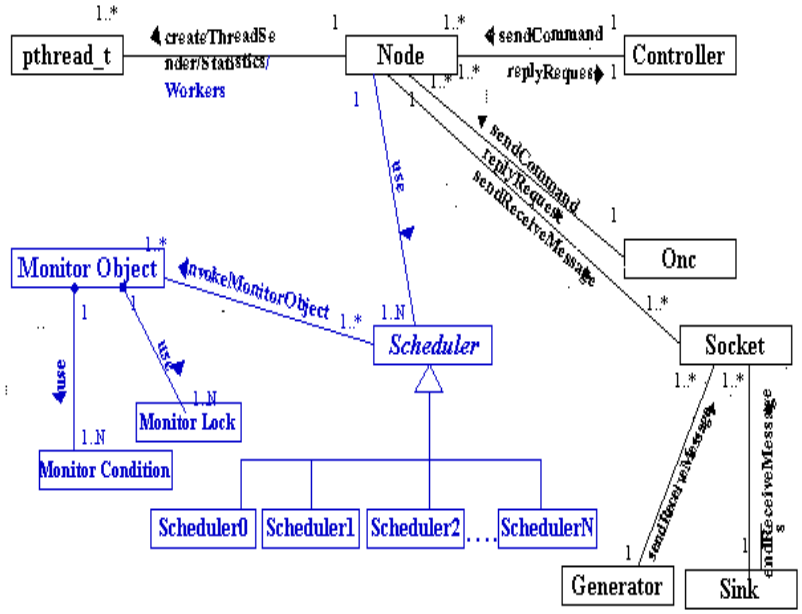


Figure 5 demonstrates the collaboration diagram for the new design to support QoS. New components are added. A Scheduler class is added to support QoS with multiple queues. There could be multiple schedulers with different number of queues. For this project, three schedulers were experimented: Schedulers 0-2. These scheduler classes were inherited from the abstract Scheduler class.

Figure 5. Partial collaboration diagram of cgNet components after restructuring



Each scheduler can have different scheduling algorithm Scheduler 2 is designed with four queues to support QoS with packets of four different priorities: command, gold, silver, and bronze. Since the queue is implemented in each scheduler, Monitor Object is utilized to control the concurrent operations by threads.

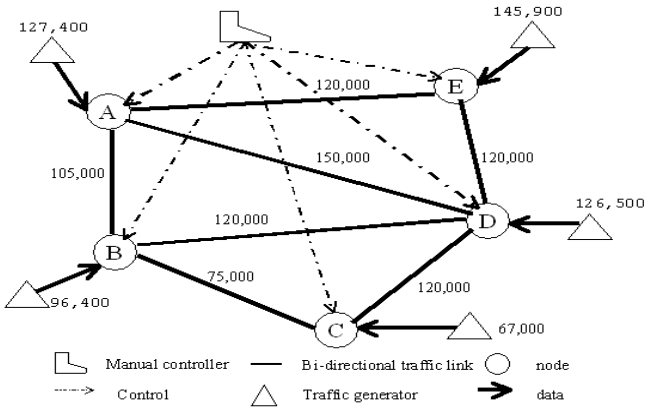
Each scheduling algorithm has its own policy for adding, dropping, and sending packets depending on the priority of incoming packets and traffic workload.

If a high priority packet arrives and the total queue capacity is full, then the scheduler checks the lower priority queue and drops the last packet in that queue. For instance, if a silver packet arrives and the total queue length is full, then Scheduler 2 checks the bronze queue and drops the last packet in that queue. Detailed discussion of the scheduling algorithm is beyond the scope of this paper. However, different scheduling algorithms can be implemented for specific needs.

4. Performance Evaluation

This section presents some evaluation results from the performance and QoS perspectives. The evaluations are conducted on a five-node network, shown in Figure 6, based on actual network data. The numbers represent the base traffic rate in bits per second (bps) or the link bandwidth (bps). The evaluations are conducted on a Pentium (R) IV machine with 1.7 GHz CPU and 256 MB of memory. The operating system is Linux Red Hat 6.0 kernel 2.4.18-3.

Figure 6. A Five-Node Evaluation Network



4.1 Evaluation Plan

cgNet consists of various types of configuration data. To evaluate the performance of the restructured cgNet, we need to consider three main areas: traffic generation rate, test duration, and technology type.

The traffic generation rate is specified as a percentage of the base engineered traffic rate. The test duration is the length of the test run. There are three different scenarios in terms of technology that need to be tested.

- OSPF only. Packets are strictly routed based OSPF protocol. No LSPs will be created.
- MPLS only. Packets are forwarded based on statically created LSPs. There are two diverse LSPs between each source and destination pair.
- Traffic controller (ONC) based on MPLS. LSPs are established just as the previous case. However, ONC may automatically control the LSPs (increase or decrease bandwidth capacity, reroute or delete LSPs) based on the traffic status.

The initial test plan is to go through all the combinations of various values. Each type has its own set of values. The following lists all the potential parameter values for the evaluation:

- Technology type: OSPF, MPLS, ONC
- Rate: 100, 130, 150, 155, 160, 170, 200, 225, 250
- Run Duration: 20 minutes
- Number of worker threads: 1, 2, 4, 6
- Number of queues:2, 3, 4

Therefore we need to run the restructured cgNet for $3 \times 9 \times 1 \times 4 = 108$ times to compare the performance with the original design. And each time the program lasts 20 minutes. We also need to include and configure the number of worker threads and queues for QoS verification. For comparison, we also need to run the original cgNet with the same set of parameter values as listed above (except for worker number parameters).

4.2 Performance Improvement

The data that we are interested in include total number of packets processed, discarded on links and on MPLS paths. For QoS, we need to measure the delay for each packet class. A tremendous amount of effort is needed for verification. The following highlights partial results to demonstrate the performance improvement and support of additional QoS requirements using the design patterns. Detailed results can be found in [11,16]

Table 1 demonstrates the results for OSPF only technology. The point to take from this table is that the number of worker threads does not have much impact on performance. The results from MPLS and ONC, not presented due to page limits, also concur this point. Therefore, we just use the restructured cgNet with one worker thread in our later performance comparison.

Table 1. Evaluation results for OSPF

	1 worker thread		2 worker threads		4 worker threads		6 worker threads	
Base Engineered Rate multiplier	Packets processed	Link Loss	Packets processed	Link Loss	Packets processed	Link Loss	Packets processed	Link Loss
1	1209947	0	1241876	0	1241453	0	1241764	0
1.3	1609670	0	1610087	0	1610126	0	1610131	0
1.5	1855030	0	1855127	0	1854781	0	1854861	0
1.55	1908934	0	1916162	0	1893297	14047	1916658	0
1.6	1978136	0	1947201	22736	1946140	22794	1978152	0
1.7	2100449	0	2100809	0	2100674	0	2091853	0
2	2377809	90126	2393399	72989	2443909	13679	2468678	0
2.25	2774561	0	2773407	0	2774619	0	2708108	38425
2.5	2993370	61806	2967616	100373	2945209	137932	2939605	83954

The original cgNet and the restructured cgNet are compared to evaluate the improvement of performance in this section. We run the original cgNet with exactly the same configurations. Table 2 shows the evaluation results for all three types of technology using one worker thread.

Table 2. Evaluation Results for the Original cgNet

	OSPF		MPLS			ONC		
Base Engineered Rate multiplier	Packets processed	Link Loss	Packets processed	Link Loss	Mpls Loss	Packets processed	Link Loss	Mpls Loss
1	1208861	0	1655413	0	13	1655480	0	26
1.3	1560857	37564	2142797	0	2051	2143763	0	129
1.5	1718044	108042	2429156	0	25079	2472751	0	276
1.55	1757378	125842	2475511	0	40605	2524556	0	741
1.6	1796612	143357	2534343	0	61821	2625639	737	987
1.7	1874747	178745	2599230	0	121021	2707464	27586	2696
2	2090450	293299	2660470	14892	367662	2788632	231517	55271
2.25	2247254	400069	2669410	36928	591021	2795026	364158	160011
2.5	2371229	531188	2669986	60480	818940	2812510	424317	334219

The evaluation results for the restructured cgNet with one worker thread are demonstrated in Table 3.

Table 3. Results for the Restructured cgNet

	OSPF		MPLS			ONC		
Base Engineered Rate multiplier	Packets processed	Link Loss	Packets processed	Link Loss	Mpls Loss	Packets processed	Link Loss	Mpls Loss
1	1209947	0	1655697	0	8	1656043	0	37
1.3	1609670	0	2144027	0	2127	2146929	0	91
1.5	1855030	0	2430515	0	25369	2462772	0	433
1.55	1908934	0	2487619	0	40732	2506863	0	1875
1.6	1978136	0	2534074	0	61647	2599288	0	762
1.7	2100449	0	2589669	0	119527	2771384	6024	2820
2	2377809	90126	2664349	8328	365847	3113868	36727	54209
2.25	2774561	0	2724666	0	591390	3117574	41357	238529
2.5	2993370	61806	2748549	0	809794	3184967	173817	383859

By comparing the data from Tables 2 and 3, we see that the total number of packets processed in the restructured cgNet is slightly larger than the number in the original cgNet. But the differences are not significant. On the other hand, the packet loss ratios are greatly reduced in the restructured cgNet. From Tables 2 and 3, the packet loss ratios are calculated respectively and shown in Tables 4 and 5. It can be seen that the packet loss ratios are significantly reduced for the OSPF and ONC scenarios for the new design using the design patterns for this particular system.

Table 4. Packet loss ratios for the original cgNet

Base Engineered Rate multiplier	OSPF packet loss	MPLS Packet loss	ONC Packet Loss
1	0.0%	0.0%	0.0%
1.3	2.4%	0.1%	0.0%
1.5	6.3%	1.0%	0.0%
1.55	7.2%	1.6%	0.0%
1.6	8.0%	2.4%	0.1%
1.7	9.5%	4.7%	1.1%
2	14.0%	14.4%	10.3%
2.25	17.8%	23.5%	18.8%
2.5	22.4%	32.9%	27.0%

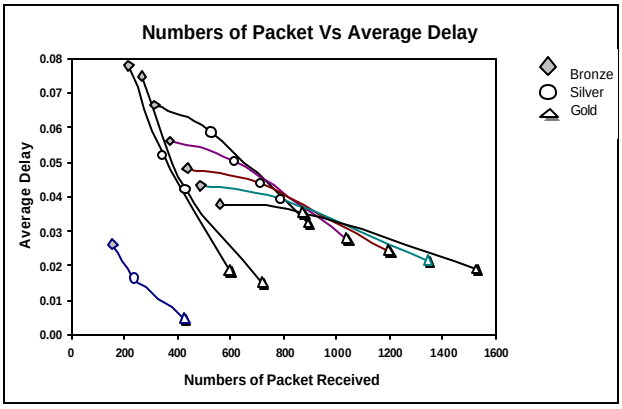
Table 5. Packet loss ratios for the restructured cgNet

Base Engineered Rate multiplier	OSPF packet loss	MPLS Packet loss	ONC Packet Loss
1	0.0%	0.0%	0.0%
1.3	0.0%	0.1%	0.0%
1.5	0.0%	1.0%	0.0%
1.55	0.0%	1.6%	0.1%
1.6	0.0%	2.4%	0.0%
1.7	0.0%	4.6%	0.3%
2	3.8%	14.0%	2.9%
2.25	0.0%	21.7%	9.0%
2.5	2.1%	29.5%	17.5%

4.3 Addition of QoS Requirements

From the QoS point of view, Figure 9 illustrates the number of packets received versus the average delay. There are eight different lines in Figure 9. Each line represents a set of statistics. The leftmost line of the graph represents the first set of statistical data, which was taken in 15 minutes after the emulation. On this line, gold series packets can be spotted at the bottom of the line. While middle empty circle represents the silver and the top diamond represents the bronze series packets. The rest of the lines follow the same sequence. From Figure 9, it can be seen that the numbers of gold packets are received relatively higher than other two, at the same time it has lower delay than others. On the other hand, bronze packets are the fewest in numbers, but they have the highest delay. So, Figure 9 clearly shows how average delay varies in QoS.

Figure 9. QoS results: number of received packets vs. average delay



5. Lessons Learned

This section presents several lessons that we have learned from this study. They are listed and discussed as follows.

- *Design patterns.* As reported by practitioners in patterns community, design patterns effectively capture design rational and provide a standardized design vocabulary. For a well studied problem, design patterns are more effective. This point is generally accepted. For this study, design patterns help us reuse existing proven technique and facilitate knowledge extraction and transfer [9,12]. Furthermore, in this particular case, patterns also help in the design recovery process. We assume that the target system, although not written based on design patterns, contains some similar concepts, since it belongs to the same problem domain and the domain is stable, and the designers are experienced in this area.

We first chose design patterns in some specific areas and then reviewed the code based on the concept of those patterns. For this study, we started with patterns for concurrent and networked objects. That way, we had more concrete goals derived from specific patterns to look for and the person performing the analysis did not need to be very experienced in patterns. It is not difficult to

identify some areas in the target system that shares commonalities with patterns because the target is also in the area of communications. More detailed discussion can be found in [10]. The description of the patterns also helps us better understand the original design.

The effort to select an appropriate pattern for restructuring for this project is insignificant mainly due to two factors. The first one is that the area is specific and well documented. Secondly, the original design shares some similarities with the Half-Sync/Half-Async pattern. However, this may not be true for other cases.

Strictly speaking, design patterns do not have direct impact on performance, as performance is dependent on specific implementation. However, performance often can benefit from patterns is concurrency and locking patterns [13]. These patterns tend to have a broad influence on application performance. For this particular study, design patterns not only satisfy the new functional requirement, but also improve the performance.

- *Software architecture*. Software architecture inevitably evolves. For this project, considerable amount of effort was spent on reverse engineering and subsequent restructuring. Verification of the system was also extremely time consuming. To mitigate the evolution effort, architecture should be built to accommodate anticipated changes or evaluated for sensitivity due to changes [8]. For example, had the architecture considered the QoS requirements to incorporate multiple queues, the verification effort would not need to be duplicated.

- *Software performance engineering*. If a software performance bottleneck is identified, increasing the number of threads for the bottleneck, generally, will result in better system performance. It is easy to configure the number of threads or queues if the original design had put it into the consideration. However, if the system is originally designed with only a single thread or a single queue, it may be difficult or time consuming to restructure the system into a multi-threaded application. The restructured system also requires tremendous amount of effort in testing and evaluation. This study illustrates the importance of conducting software performance engineering [15] at the early stage to mitigate effort for performance enhancement later in the life cycle phases.

- *Tools and aspect-oriented programming*. As stated earlier, a tremendous amount of effort was spent on issues related to concurrency and synchronization. Synchronization is tedious, but is well understood. Tools can be developed to support translating a single threaded program to multi-threaded. The concept of aspect-oriented programming [6] can be applied to this area.

6. Summary and Future Directions

This paper presented an experimental study on software restructuring with design patterns. With patterns, we modified the system to meet new functional requirements as well as non-functional performance attribute. Software evolution is common. Software evolution with proven design patterns will likely improve the quality of the system. For specific areas, such as concurrency and synchronization, patterns often can result in better performance.

The translation of a single-threaded program to a multi-threaded program is tedious. Many parts involved are mechanical. Tools can be developed to support this process to mitigate human errors. We are working on tools to support this process.

References:

[1] R. S. Arnold and S. A. Bohner, *Software Change Impact Analysis*, IEEE Computer Society Press 1996.

[2] Buschmann, F. and Meunier, R. *Patterns of Software Architecture: A System of Patterns*. Addison-Wesley, Reading, MA., 1995.

[3] B. Davie and Y. Rekhter, *MPLS Technology and Applications*, Morgan Kaufmann Publishers, 2000.

[4] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software Architecture.* Addison Wesley, Reading, MA, 1995.

[5] R. Kazman, G. Abowd, L. Bass, P. Clements, "Scenario-Based Analysis of Software Architecture", *IEEE Software*, November 1996, pp. 47-55.

[6] G. Kiczales et al., *Aspect-Oriented Programming*, *Proc. of European Conf on Object-Oriented Programming*, 1997.

[7] C.-H. Lung, et al., "Performance-Oriented Software Architecture Analysis", *Proc. of the Int'l Workshop on Software Performance Eng. (WOSP)*, 1998, pp. 191-196.

[8] C.-H. Lung and K. Kalaichelvan, "A Quantitative Approach to Software Architecture Sensitivity Analysis", *Int'l J. of Sw. Eng and Knowledge Eng*, 10 (1), 2000, pp. 97-114.

[9] C.-H. Lung, G. Maculak, and J. Urban, "Software Reuse and Knowledge Transfer through Analogy and Design Patterns", *Proc. of Int'l Conf. on Software Eng. Research and Practice (SERP)*, June, 2002, pp. 618-624.

[10] C.-H. Lung, "Agile Software Architecture Recovery through Existing Solutions and Design Patterns", *Proc. of 6th IASTED Int'l Conf. on Software Engineering and Applications (SEA)*, Nov. 2002, pp. 539-545.

[11] H. Mar, H. Xu, and P. Kanagaratnam, *Quality of Service for MPLS Software*, 4th Year Project Report, 2003, Dept. of Systems and Computer Eng, Carleton Univ., Ottawa, Canada.

[12] D. May and P. Taylor, "Knowledge Management with Patterns", *Com. of the ACM*, 46 (7), 2003, pp. 94-99.

[13] P.E. McKenney, "Selecting Locking Primitives for Parallel Programming", *Com. of the ACM*, 39 (10), 1996, pp. 75-82.

[14] D. Schmidt, et al., *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000.

[15] C. U. Smith, *Performance Engineering of Software System*, Reading, MA, Addison-Wesley, 1990.

[16] Q. Zhao, *Pattern-Oriented Software Reengineering of a Network Traffic Engineering System*, Master Project Report, 2003, School of Computer Science, Carleton Univ., Ottawa, Canada.

Acknowledgements:

We are grateful for Nortel Networks for granting us permission to use cgNet for research and education.