# Pattern-Oriented Reengineering of a Network System

**Chung-Horng LUNG**
**Department of Systems and Computer Engineering, Carleton University**
**Ottawa, Ontario K1S 5B6, Canada**

**and**

**Qiang ZHAO**
**Department of Systems and Computer Engineering, Carleton University**
**Ottawa, Ontario K1S 5B6, Canada**

## ABSTRACT

Reengineering is to reorganize and modify existing systems to enhance them or to make them more maintainable. Reengineering is usually necessary as systems evolve due to changes in requirements, technologies, and/or personnel. Design patterns capture recurring structures and dynamics among software participants to facilitate reuse of successful designs. Design patterns are common and well studied in network systems. In this project, we reengineer part of a network system with some design patterns to support future evolution and performance improvement. We start with reverse engineering effort to understand the system and recover its high level architecture. Then we apply concurrent and networked design patterns to restructure the main sub-system. Those patterns include *Half-Sync/Half-Async, Monitor Object, and Scoped Locking idiom*. The resulting system is more maintainable and has better performance.

**Keywords**: Reverse Engineering, Reengineering, Design Patterns, Networked and Concurrent Software, Refactoring.

## 1. INTRODUCTION

Software architecture has become a major topic in the past several years because of the increasing complexity of software systems. Software architectures also play a crucial role for managing the changes. It is common in practice to reconstruct architecture from the existing design and modify the system to accommodate changes through 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 maintenance and evolution.

Reengineering consists of two main phases: reverse engineering and forward engineering. Reverse engineering is the process of extracting system abstractions and design information out of existing software systems to facilitate program comprehension. Forward engineering, in this context, deals with the subsequent re-design and implementation from the recovered system to meet the evolutionary objectives. Restructuring or refactoring [3] may be needed in this stage to improve the quality, maintainability, or performance of the existing design.

Design patterns are proposed as a way to produce more reusable and adaptable designs. Design patterns capture recurring structures and dynamics among software participants to facilitate reuse of successful designs. They provide a common vocabulary for talking about design solutions among designers. The basic idea behind design patterns is that similar idioms are found repeatedly in software designs and that these patterns should be made explicit, codified, and applied appropriately to similar problems.

cgNet is a network system and application based on MPLS protocols (Multi-Protocol Label Switching) [2] developed at Nortel Networks. The system was designed to realize basic functional requirements under timing constraints for concept demonstration. As a result, some portions of the system were not well designed and there was no documentation. Furthermore, there was a need to enhance the system to support research in QoS (Quality of Service) and other areas at Carleton University.

cgNet was not originally designed based on patterns. Nevertheless, it shared similarities with other network systems, because the original designers were mostly experienced in the network area. The system shares similarities with other network systems. Concurrent and networked design patterns are common in network systems and applications, and are well documented [9].

Hence it is logically reasonable to assume that cgNet shares some similar concepts with the design patterns for concurrent and networked objects. We began with reverse engineering effort with an aim to understand the system. The reverse engineering process starts with studying well-known patterns in network applications together with code review to help better understand the system. By studying the relevant design patterns and reviewing the code, we realized that the core the system could be restructured with some design patterns to facilitate the addition of new QoS features. In addition, we found that some design patterns had the potential to increase system performance.

We adopted the *Half-Sync/Half-Async* design pattern as the overall structure for the main software process. The pattern is used together with the *Monitor Object* pattern and *Scoped Locking* for handling the request queue in the *Half-Sync/Half-Async* pattern. A *multi-read/single-write* mechanism is provided to realize the synchronization among threads. The resulting system has better code and faster system performance.

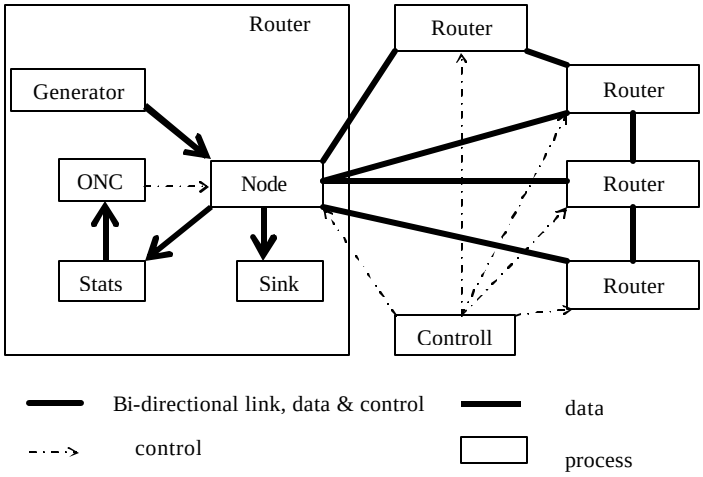
The rest of the paper is organized as follows. Section 2 describes the original cgNet structure through a reverse engineering effort. Section 3 demonstrates the restructured design based on the design patterns. Section 4 discusses the effort spent on the reengineering effort and briefly presents some performance comparisons. Finally, Section 5 summaries the paper with conclusions and provides suggestions for future related works.

## 2. REVERSE ENGINEERING OF CGNET

The cgNet software consisted of about 30,000 lines of code and contained complicated operations, algorithms, and domain knowledge in networks and traffic management. The system was written in C++, but some parts were written in C style. There was no design documentation for the software except the user's guide. From the user's guide and the executable software processes, we have the high-level process view of the design shown in Figure 1.

As shown in Figure 1, cgNet is composed of software routers. Routers are symmetrical and identical except possibly that the neighbors and number of connections to other routers may be different. These router processes can be run on the same machine concurrently or on separate machines.

**Figure 1. System level view of cgNet**




A router consists of a traffic generator, a sink, a statistics sink, an ONC (intelligent network controller) and a node. A generator process randomly generates data packets that will be forwarded to other router processes. A sink process consumes data packets received from other router or its own generator. A statistics sink process consumes statistic reports generated periodically by the node process. A manual controller process is the user interface that receives commands from the user and sends the commands to the node processes. An ONC process automatically sends the appropriate commands to the network to formulate the necessary network changes required to improve network status. These commands are based on network status from real-time network statistics. A node process forwards traffic (control and data) towards the destination sink along the MPLS paths or along Layer 3 routes that use the OSPF protocol.

Among all the processes, the node process is the most complicated one and plays a crucial role in the system. A node process consists of about 10,000 lines of code and 90 methods or so. It was written in C++, but mostly in C style. In fact, the key to recover the structure and design of cgNet was to analyze and understand the node process.

The reverse engineering process is an extension of the approach described in [6]. More precisely, the process is an iterative effort of walking through the program and studying the patterns for concurrent and networked applications. Studying and comparing those patterns actually helped us better understand the system, even though the system was not built with patterns in the first place. The main reason probably was because the system was built by a few very experienced software designers. In other words, those designers had seen or applied similar concepts in previous projects.

Another reason that studying patterns helped the process was the problem domain. Identifying potential design patterns that may exist in the structure of an analyzed system is an important complement to improve the comprehension of how determined parts of the system were designed and the relationships with some other components. Although cgNet was not written based on design patterns, it is in the area of communications that is a well-studied domain. Numerous articles on design patterns in networks and telecommunications have been published in the literature [8, 9].

It is, therefore, logically reasonable to assume that cgNet shared similar concepts of design patterns for concurrent and networked objects. The strategy was to study design patterns listed in [9] and review the code based on the concept of those patterns. By studying the patterns and comparing them with cgNet, we also better understood the system.

Figure 2 demonstrates the structure of the node process which consists of multiple threads: a main thread, a statistics thread, and multiple destination threads.

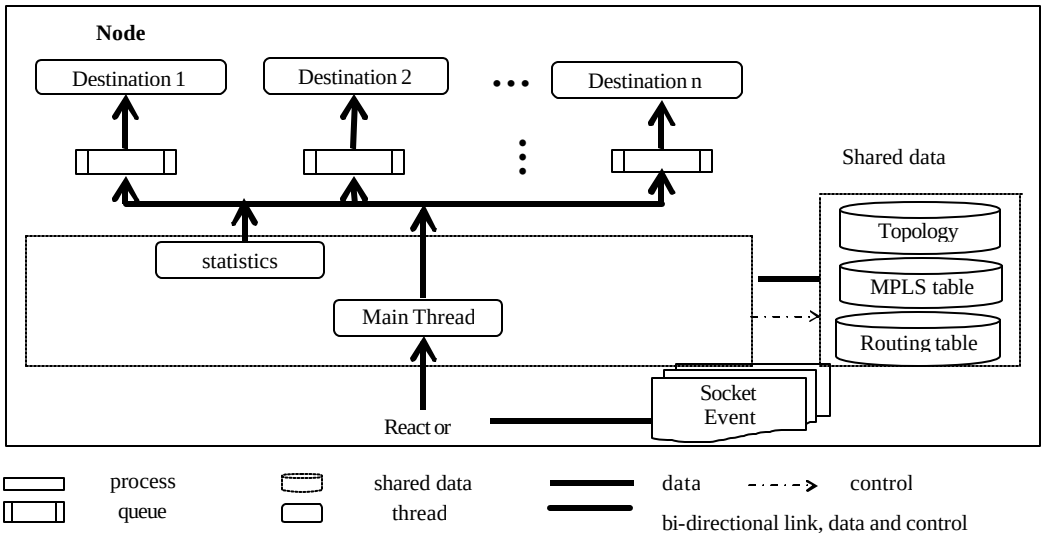
The main thread first initializes the node, connects to its generator, sinks and neighboring nodes, and creates the statistics thread and destination threads. After the initialization stage, the main thread repeatedly reads a packet from one of its sources, processes it, and enqueues it according to the first-come first-served principle.

The statistics thread collects statistics on all sources, destination links and MPLS paths at the end of each statistics interval. Then the thread generates statistics packets and puts them into the appropriate destination thread's queue.

There are a thread and a queue for each destination. A destination could be a data sink, a statistics sink or a neighboring node process. A destination thread removes a packet at a time from the corresponding queue and sends the packet out through the destination link.

The main thread and the statistics thread also access the shared data including network link topology, routing table, and MPLS table synchronously.

**Figure 2. Recovered software architecture of the Node process**




### 3. RESTRUCTURING OF CGNET WITH DESIGN PATTERNS

After the reverse engineering process, the main tasks were identified in the subsequent phase. The original cgNet did not support QoS. So, the first objective was to restructure the system with design patterns to facilitate multiple queues management to classify received packets and process them according to their priorities with an aim to support QoS.

As we conducted the reverse engineering process, we also identified a software performance bottleneck in the node process. The processing tasks for a packet may include route lookup, routing table update, MPLS table lookup, MPLS path setup/reroute/change/delete, traffic policing, topology update or data forwarding. It takes a long time to execute a sequence of tasks just mentioned for each packet before reading the next one.

Generally speaking, performance is not directly related to patterns, as performance depends on where the performance bottlenecks are and how patterns are implemented. However, performance will benefit from patterns is concurrency and locking patterns [8, 9]. These patterns tend to have a very broad affect on application performance. Therefore, the second objective was to increase performance by adopting a concurrent design pattern.

We adopted the *Half-Sync/Half-Async* design pattern as the overall structure for the node process. The pattern is coupled with the *Monitor Object* pattern and the *Scoped Locking* for handling the request queue in the *Half-Sync/Half-Async* pattern. A *multi-read/single-write* mechanism is provided to realize the synchronization among threads. Figure 3 shows the restructured design.

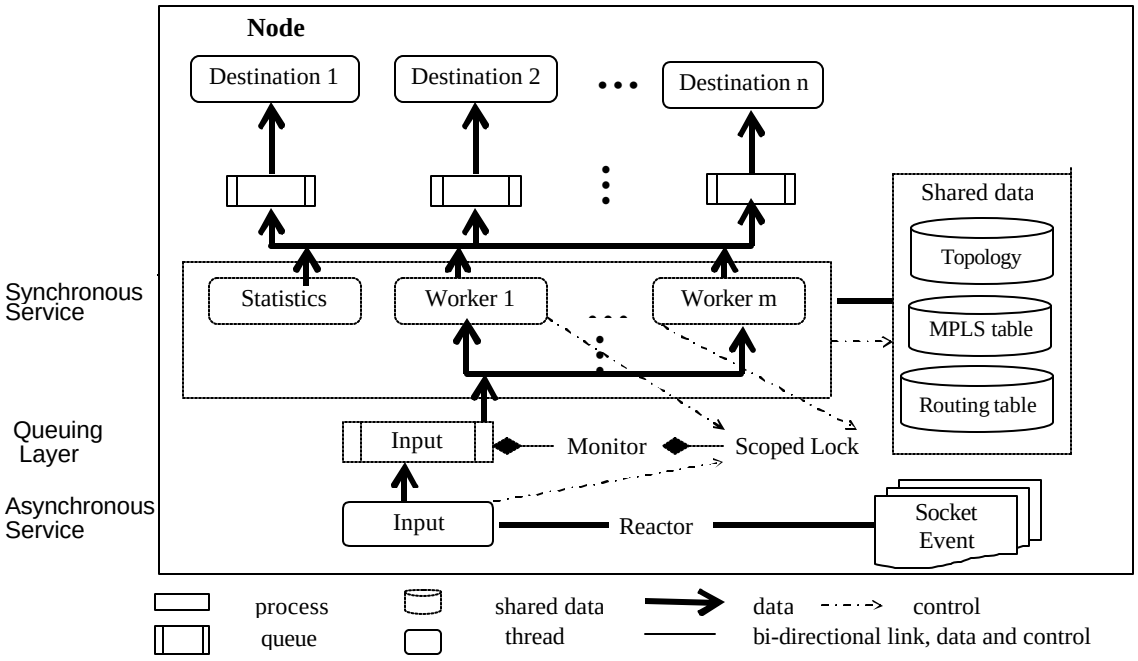
The input thread deals with the asynchronous input and output with peers. The input thread stores the incoming messages into the input queue. The queuing layer has an input queue which actually consists of multiple sub-queues; each having a priority and being used for one type of messages or packets. The property is used to store different types of packets into different sub-queues. QoS can then be realized with this design. Supporting QoS was the main objective of this project.

The input queue was implemented with the Monitor Object pattern in which object synchronization corresponds to method invocations. Synchronized methods (put and get) use their monitor conditions to determine the circumstances under which they should suspend or resume their execution. Scoped Locking idiom was adopted in the implementation to acquire and release locks automatically when control enters and leaves critical sections.

A *multi-read/single-write* mechanism was used to implement the synchronization of worker threads and the statistics thread to protect shared data. Multiple threads are allowed to read the shared data concurrently. But other threads are excluded to access the shared data when one thread is updating the shared data.

Multiple worker threads remove messages in the input buffer and handle them according to message type. Synchronization among the input thread and worker threads are supported with the Monitor Object pattern and the Scoped Locking mechanism. In the new design, the input thread and the worker threads are working concurrently. As opposed to the main thread described in Section 2, the input thread performs much less work in the new design. Most of the tasks are actually delegated to the worker threads. Therefore, the performance bottleneck is removed. Next section will present some performance results for comparison.

**Figure 3. Structure of the new Node process**




#### 4. EVALUATION OF THE REENGINEERING EFFORT

This section presents assessments of the reengineering task. The first phase of this project included reverse engineering of the code and review of design patterns discussed in [9]. This phase played a crucial role to the success of the subsequent restructuring task. As mentioned in section 2, cgNet involves many technical areas, including complicated operations, algorithms, concurrent programming, and domain knowledge in networks protocols, signaling, and traffic management. Roughly, 4.5 man-months were spent in this phase.

The second phase was restructuring of the design with patterns discussed in the previous section. Approximately, this phase took 4 man-months. Design patterns helped shorten the development time for this stage. The third phase was the postmortem analysis for performance based on various traffic parameters Performance characterization of a network system is a complicated task, which is not directly related to this paper.

We did not conduct rigorous change impact analysis [1] or software architecture sensitivity analysis [5] before the project started. Here, we just provide some assessment data after the restructuring effort for reference purpose. The main process that was modified had about ninety methods. Twenty-six of them were modified, all with minor changes; one method was removed; and sixteen new methods were added. Many changes were related to synchronization, which spread over all the places. With the incorporation of those patterns, the sensitivity will be reduced due to similar changes, because those changes will be confined to some patterns only in the new design.

We also conducted performance evaluations based on the software performance engineering approach [4, 10]. Performance has improved substantially for most scenarios with the new design primarily due to the parallelization of the input thread and the worker threads. Table 1 illustrates some performance results for one scenario on Pentium (R) IV with 1.7 GHz CPU and 256 MB of memory and Linux kernel 2.4.18-3. However, the number of worker threads does not have significant differences. This paper emphasizes on software reengineering. Detailed discussion on performance analysis is beyond the scope of this paper.

**Table 1. Comparison of packet loss ratios**

| Base engineered rate multiplier | OSPF packet loss in the original design | OSPF packet loss in the new design with one worker thread |
|---|---|---|
| 1 | 0.0% | 0.0% |
| 1.3 | 2.4% | 0.0% |
| 1.5 | 6.3% | 0.0% |
| 1.55 | 7.2% | 0.0% |
| 1.6 | 8.0% | 0.0% |
| 1.7 | 9.5% | 0.0% |
| 2 | 14.0% | 3.8% |
| 2.25 | 17.8% | 0.0% |
| 2.5 | 22.4% | 2.1% |

## 5. CONCLUSIONS

This paper analyzed and reengineered a network traffic engineering system. We recovered its high level architecture and then restructured part of it using concurrent and networked design patterns. The design patterns adopted in the restructured cgNet included *Half-Sync/Half-Async pattern*, *Monitor Object pattern*, and *Scoped Locking idiom*.

With the pattern-oriented restructuring, we achieved the following benefits:

- Support of QoS. Packets can be inserted into different sub-queues for processing according to their QoS priorities. The numbers of sub-queues and worker threads are also configurable for different requirements.

- Performance improvement. In the original cgNet, packets/commands processing, including MPLS path setup/change/reroute/delete/policing, routing lookup, command executing, happened in a single thread which caused a bottleneck for the performance. Multi-threaded packets/commands processing of the restructured cgNet improves the performance considerably.

- Better code. The restructured code with well-known design patterns has better structure and common vocabularies. This will also support future evolution.

The concept of some other design patterns is also used in the system. For example, Reactor pattern, Accept-Connector pattern, and Active pattern. We are also planning to further refactor the system with some of those design patterns to make the code better.

## 6. REFERENCES


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

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

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, **Refactoring: Improving the Design of Existing Code**, Addison-Wesley, 1999.

[4] C.-H. Lung, A. Jalnurpukar, and A. El-Rayess, "Performance-Oriented Software Architecture Analysis**", Proc. of the International Workshop on Software Performance Engineering (WOSP),** 1998, pp. 191-196.

[5] C.-H. Lung and K. Kalaichelvan, "A Quantitative Approach to Software Architecture Sensitivity Analysis", **International Journal of Software Engineering and Knowledge Engineering**, vol. 10, no. 1, Feb 2000, pp. 97-114.

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

[7] P.E. McKenney, "Selecting locking primitives for parallel programming", **Communications of the ACM**, vol. 39, no. 10, Oct. 1996, pp. 75-82.

[8] L. Rising and D.G. Firesmith (editors), **Design Patterns in Communication Software,** Cambridge University Press, 2001.

[9] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, **Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects**, John Wiley and Sons, 2000.

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


## ACKNOWLEDGEMENTS


We are thankful for *Nortel Networks* for granting us permission to use cgNet for research and education. We are particularly grateful for cgNet and ONC designers: Gord McLennan, Chris Hobbs, Geroge Young, Michel Dallaire, Gwenda Lindhost-Ko, and Anthony Van Alphen.