Architecture-Centric Program Transformation for Distributed Systems

Chung-Horng Lung, Jianning Liu, Xiaoli Ling, Dan Jiang

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6, Canada
chlung@sce.carleton.ca

Abstract

Software evolution is inevitable due to changes in requirements and technology. Software quality and productivity are critical to software evolution. Architecture-centric evolution has the advantage of improving quality attributes; however, it does not directly address the issue of the time needed for evolution. This paper proposes an architecture-centric program transformation approach to support both quality and productivity concerns. The approach emphasizes the analysis of similarities and differences among architecture alternatives, which facilities the tool development for architecture evolution or evaluation. A tool was constructed to automatically transform a program to another form based on user's selection. The tool currently supports the transformation of one traditional design to the other two commonly used architecture patterns in distributed and concurrent systems.

1. Introduction

Distributed software systems and applications are common these days in the Internet age and they are getting more and more complex because of the development of new requirements and features. As a result, the evolution of existing distributed software systems also becomes difficult, time-consuming, and error prone. In addition, the evolution of software systems usually occurs at the code level, which in general is not as effective as that at the architecture level, because software architecture captures the main components and their relationships, and the design rationale of the system. Substantial properties of a system, such as the quality attributes, are best described and tackled at the architecture level. Therefore, software evolution at the architecture level or architecture-centric evolution will better facilitate the synchronization of new requirements: either user or technology requirements, design, and implementation.

However, enterprises also face another challenge–time-to-market. Software quality attributes–such as performance, security, reliability, availability, and modifiability have been advocated as drivers for architecture-centric design [Barbacci03]. But it often does not directly or explicitly address the issue of time-to-market. In fact, time-to-market may play a more important role in practice in the initial design or the evolution stage than those quality attributes to enterprises, especially in highly competitive areas. From the evolution perspective, the architect may adopt a new software architecture for non-functional requirements such as those qualities. Therefore, it is critically important to support software developers who attempt to transform an existing system to a new one with a different architecture and/or new technologies to reduce the time and improve other qualities.

As stated earlier, distributed systems have become common these days. However, the design of many distributed systems has not taken advantage of some new technologies in this area. For example, two original systems under study in telecommunications and wireless applications were developed using the single-thread (ST) approach. Multi-thread design alternatives, such as Half-Sync/Half-Async (HS/HA) and Leader/Followers (LFs) design patterns [Schmidt00], typically can increase the performance. In fact, one of the systems under study was converted from the ST to HS/HA and the other one was transformed to LFs. Many design patterns in distributed applications have been captured and documented [Schmidt00]. Some of the patterns are lower-level artifacts suitable for detailed design; some, on other hand, depict the high-level structure or architecture of a system. Two examples of patterns that fall into the later category are HA/HA and LFs.

Manual transformation of a legacy system to a new one using the advanced technology may need a lot of time or specialized knowledge even though those patterns have been well-documented. Lung, et al [Lung04a, Lung04b] reported an empirical study of transformation from ST to HS/HA for higher

performance and quality-of-service requirements, which required extensive knowledge and lots of efforts. Therefore, the objective of this paper is to develop a tool to facilitate the transformation of distributed programs using well-documented design patterns. The focus of this paper at this point is on two commonly adopted concurrency patterns: HS/HA and LFs.

The program transformation tool was developed based on analyses of existing systems, including a generative framework and reusable components designed using relevant design patterns. The tool was written in Python to automatically transform an existing system that is designed using the traditional ST approach in Java to another Java program that uses either of the two advanced concurrency design patterns: HS/HA and LFs. Selection of the design patterns is made by the user. The existing system can be transformed with our software tool much more easily compared to manual transformation. Time spent on transformation can be greatly reduced and software quality can also be assured since the tool has been well tested and verified. In addition, a ST system can be transformed to HS/HA and LFs using the tool to support architecture tradeoff [Kazman98] or sensitivity analysis [Lung00].

The rest of this paper is organized as follows: Section 2 describes some related work. Section 3 discusses the program transformation approach. Section 4 presents a program transformation tool and illustrates a case study using the tool. Finally, section 5 is the summary.

2. Related Work

Design patterns have been well accepted in the software engineering community because of their practical and theoretical importance. Design patterns could be generic to many different domains or specific to a particular domain. For example, many design patterns described in [Gamma95] can be applied to multiple domains, but patterns documented in [Schmidt00] are specific to distributed and concurrent systems. This paper focuses on the patterns discussed in [Schmidt00].

Schmidt, et al [Schmidt00] presented several types of design patterns in networked and concurrent systems. They include service access and configuration patterns, event handling patterns, synchronization patterns and concurrency patterns. Service access and configuration patterns mainly deal with effective application programming interfaces (APIs) to access and configure services and components in stand-alone or networked systems. Detailed discussions of those patterns can be found in [Schmidt00].

Event handling patterns are used to describe how to initiate, receive, demultiplex, dispatch, and process events in networked systems. There are four patterns in this category: reactor, acceptor-connector, proactor, and asynchronous completion token. The first three patterns were used in our study. Reusable components were built based on these three patterns, which are useful for program transformation.

Concurrency patterns represent the architecture alternatives of typical communications software. Three basic architectural alternatives are selected in our paper from the concurrency management perspective. They are the traditional ST approach using the Reactor pattern, HS/HA, and LFs. ST was selected, primarily due to its simplicity and, more importantly, it was the style originally used in the existing software under study. Many legacy systems were also developed using the ST approach. The other two alternatives are included mainly due to their acceptance in this field [Schmidt00]. Both HS/HA and LFs have the potential to improve system performance.

Lung [Lung03] conducted a preliminary variability analysis for various architectural alternatives in communications software. Figure 1 displays three of these common alternatives at an abstract level. The thread in the ST approach will handle events via the *select()* function and process the incoming message. However, this approach often leads to scalability concerns. This problem can be improved using either the HS/HA or the LFs pattern as the overall architecture.

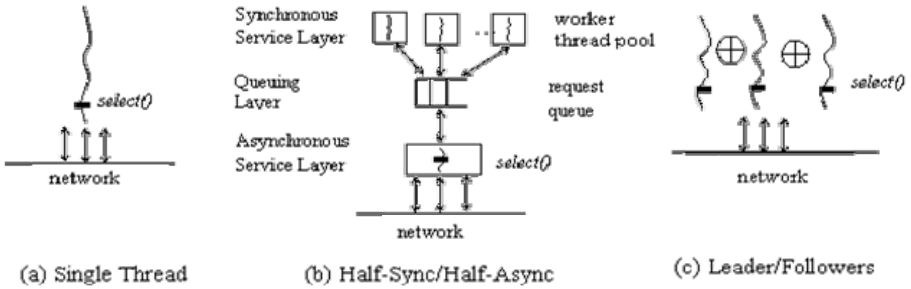

Figure 1. Basic Networked and Concurrent Architectural Patterns

In LFs, multiple threads function similarly to that in the ST example and synchronization of those threads is provided. However, only one thread at a time—the leader—waits for a network event to occur. Other threads—the followers—can queue up, waiting for their turn to become the leader. Once the leader detects an event, it promotes one of the followers to be the leader. It then becomes a service-processing thread. HS/HA, on the other hand, divides the system into three layers, as shown in Figure 1(b). The asynchronous layer reads messages from the network and stores them in the queue. Multiple worker threads read messages from the queue and process them.

3. Architecture-Centric Program Transformation

As illustrated in Section 2, those architecture alternatives share similarities, but they also have differences. Each alternative has its pros and cons, even though HS/HA and LFs in general could have better performance. Nonetheless, software architecture may involve complicated operations or behaviours. For a real case study, we reengineered a network system from ST to LFs for performance improvement. However, the system based on the LFs pattern did not actually perform better than the ST approach as expected [Alhussaini04]. Using a tool to facilitate architecture-centric evolution not only can save time, but also can foster architecture evaluation with working systems before the actual transformation.

The following describes the approach that we have adopted to support architecture-centric program transformation. The approach is built on top of our previous research of an architecture-centric generative framework [Lung06]. The development of the framework consists of the following steps:

1) Define the scope and conduct a variability analysis at the architecture level.
2) Conduct design recovery of existing robust software systems.
3) Reengineer existing systems using patterns.
4) Conduct evaluation of architecture alternatives.
5) Construct reusable components and the framework.

Section 2 already highlighted the first step. Step 2 advocated that it is important to reuse existing working systems rather than building from scratch. We have studied multiple systems, including a network prototype system developed by the industry. The design recovery process provided valuable information and was very helpful for the subsequent reengineering effort. We have manually reengineered ST systems [Alhussaini04, Lung04a, Lung04b] using various design patterns: Reactor, Acceptor-Connector, HS/HA, and LFs.

Thorough software architecture evaluation for ST, HS/HA, and LFs with emphasis on performance was then conducted [Alhussaini04, Lee04, Lung04b, Wu03]. The process helped us better understand the quality attributes of different alternatives. Lastly, a framework consisting of reusable components–Dispatcher, Connection Accepter, Connector, Service Handler, HS/HA, and LFs–was developed. The framework allows the user to select and instantiate a system using a particular alternative–ST, HS/HA, or LFs.

The paper extends the previous work by building a tool to support automatic program transformation based on the user's selection. The approach consists of the following main tasks:

1) Conduct analysis to identify similarities and differences among the alternative patterns at the architecture/design level based on the selected design patterns and existing systems.
2) Conduct similar analysis at the component level.
3) Conduct similar analysis at the code level.
4) Conduct manual transform from ST to HS/HA or LFs, and verify the results.
5) Develop a scheme and build a tool to transform the ST program to either HS/HA or LFs.

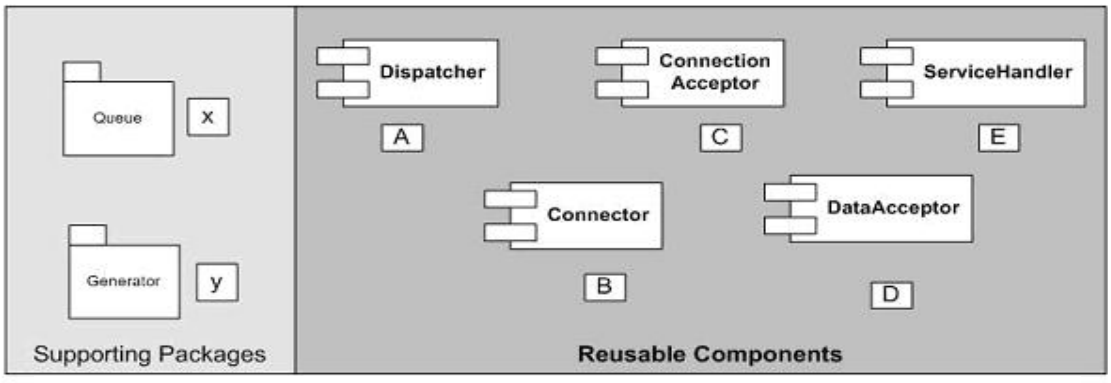

Figure 2. Reusable Components Developed Using Design Patterns for a Generative Framework [Lung06]

Each step will be described in more detail as follows:

1) Identify similarities and differences at the architecture or design level

This step is an extension of the variability analysis in our previous work. The focus here is to be specific and concrete, since the ultimate goal is to build an actual tool instead of working only on the high-level artifact. For this particular case study, not much more was added other than the results shown in Figure 1. However, this is the first step that we advocate and we are also looking at applying the method to other areas in the future. In addition, we have examined a couple of systems using different design alternatives

2) Identify similarities and differences at the component level

The main purpose of this step is similar to that of the traditional detailed design phase. A set of reusable components were developed in our previous generative framework. Figure 2 shows those components which is not necessary for this step. However, with those components available, the tool may replace existing relevant code segments for reengineering or retrieve them for new systems. The components on the right in Figure 2 are reusable across all three alternatives with some modifications needed for specific patterns. The components on the left in the figure are either used specifically for HS/HA (component *x*) or for the network application layer (component *y*).

The *Dispatcher* or component *A* shown in Figure 2 handles synchronous or asynchronous demultiplexing usually realized by the *Reactor* or *Proactor*, respectively [Schmidt00]. The *Dispatcher* component handles connection completion event, incoming new connection, incoming data, and readiness of the socket for writing data. The *Connector* component, *B*, is refined from the *Connector* design pattern described in [Schmidt00]. It implements the strategy for actively establishing a connection and initializing its associated *ServiceHandler*. *ConnectionAccepter* (component *C*) is refined from the *Acceptor* design pattern. The main function of this component is to passively wait for connection requests from remote connectors. It then establishes the connection with the actual host machine. *DataAcceptor* (component *D*) is similar to the *ConnectionAcceptor* component, but it receives data from the *Dispatcher* when data arrives. *ServiceHandler* implements an application-specific service. In this case study, the application has to with message routing and forwarding in a network application. *ServiceHandler* is mainly used together with the Generator (component *y*) to emulate a network routing application for the application level.

3) Identify similarities and differences at the code level

The next step is to perform detailed analysis at the code level. As stated earlier, the goal is to develop a program translation tool, so it has to be precise. Therefore, it is required to identify the exact similarities and differences among the alternatives. Figure 3 shows an example of some differences between the HS/HA pattern (on the left) and the ST (on the right) design.

4) Conduct manual transform from ST to HS/HA or LFs, and verify the results

Before the tool was developed, we manually convert the system from the ST design to either the HS/HA pattern or the LFs pattern. This step is necessary to ensure that the converted systems will actually work properly. We have re-engineered the code and verified the correctness of the results before building the tool.

5) Develop a scheme and build the tool

The final step is the actual development of the tool, including design, implementation, and testing. The tool is written in Python and the target program language is Java, though we have also analyzed C++ programs. Section 4 discusses the tool in more detail.

4. Program Transformation Tool

The section describes the approach that we used for the tool development. The initial version of the tool is tailored to the generative framework that we used during the analysis process and the tool focuses on the transformation from ST to either HS/HA or LFs. Section 4.1 describes the steps that are involved and section 4.2 presents more detailed algorithm.

4.1 Transformation Steps

The process of developing the transformation tool involves three steps: extraction, insert, and re-composition. *Extraction* deals with extraction of classes and methods. In our approach, extraction is used to extract classes. The *insert* step adds specific key lines or methods identified during the difference analysis phase. Finally, the *re-composition* step reconstructs the methods and classes. These three steps will be elaborated more later in this section.

The software tool also has a library consisting of all the supporting files for implementing specific design patterns. Based on the design pattern selected by the user, the appropriate supporting files will be connected to the *Interfacer* class derived from the generative framework. The *Interfacer* class is used to instantiate a specific design option.

The supporting files in the library are used to construct the basic structure of the LFs pattern or the HS/HA pattern. The following is a list of the supporting files in the library used in the transformation:

- Worker pool
- Worker
- Queuing layer
- Asynchronous layer
- ServiceHandlerThread

The first two files are used for the LFs pattern; while the next three files are used for the HS/HA pattern.

Extraction reads in the single-thread *Interfacer* class and extracts the methods in the class. Note that it is not necessary to read in the entire source file as the differences have been identified already.

The *insert* phase adds new methods. Two methods are added to the ST method: *startLeaderFollower()* and *stopLeaderFollower()*. In addition, new associations are established: the association between the *LeaderFollowerInterfacer* and *WorkerPool*, and the association between *WorkerPool* and Worker classes.

From ST to HS/HA, one extra method is added: *startServiceHandlers()*. New associations, the association between *HSyncHaSyncInterfacer* and *ServiceHandlerThread*, and the association between *HSyncHaSyncInterfacer* and *AsynchronousLayer*, are also established.

Re-composition is a process of reconstructing the code. During the process, the existing system with newly added methods and re-used components are reconstructed. A new output file containing the generated program is produced after this process.

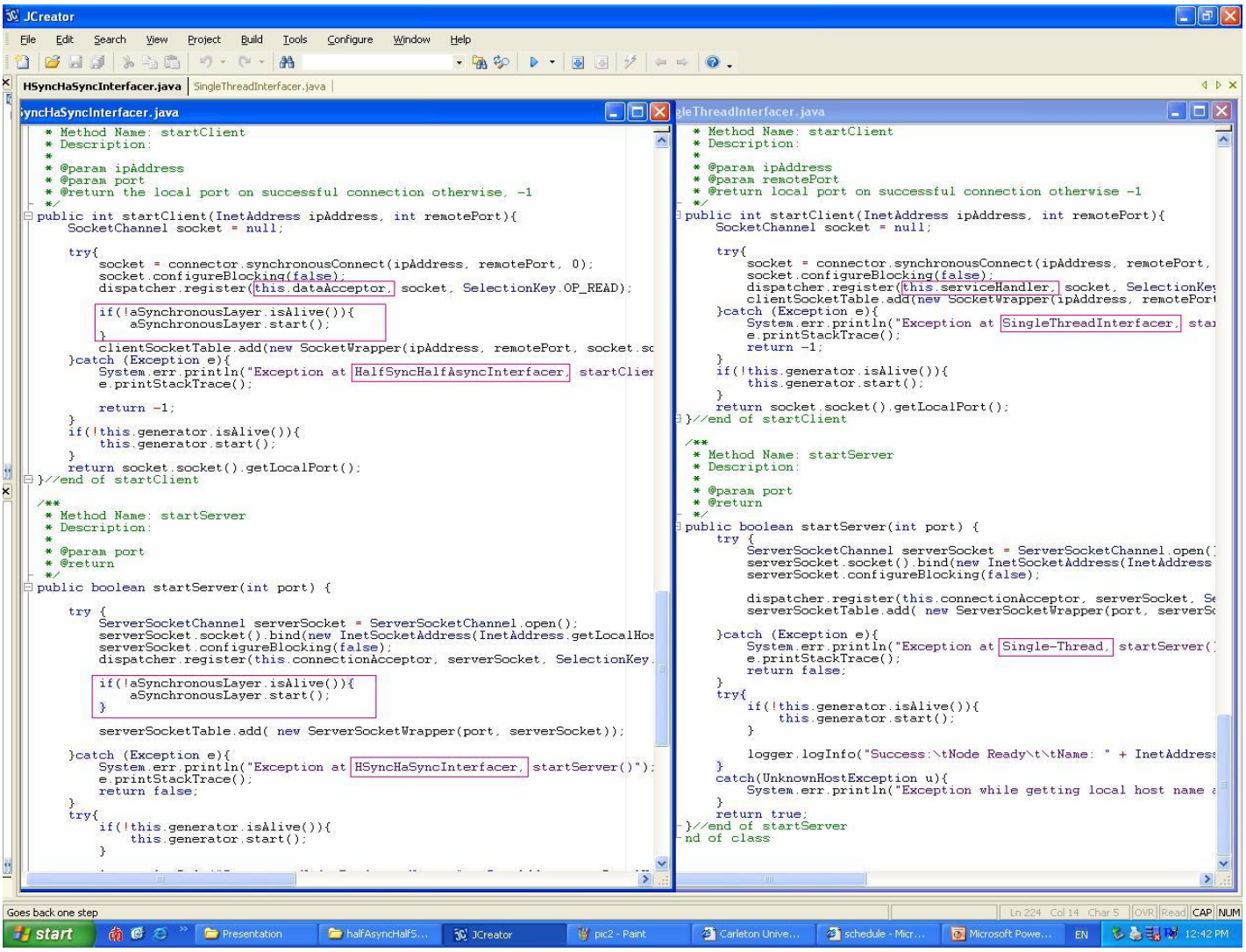

Figure 3. Differences between the HS/HA pattern and the ST Program: an Example

4.2 Detailed Algorithm

This section presents the algorithm used in the tool. Each tool keeps track of each method in the ST approach and the differences between the ST approach and the HS/HA or LFs patterns.

The following shows the algorithm in more detail:

Each method in the ST code is treated as a unit during the transformation. For each method, the tool will search the original ST code line by line for transformation. In addition, the tool stores the changes (identified previously during the analysis phase discussed in section 3) needed for each method, including header files, additional variables, and etc.

Algorithm:

```
main():
{
    Add additional header files if needed
    Insert additional variables if needed
    Read input file
    Read line
    Repeatedly find the start of a method that needs to be
    transformed and do
    {
        Read line until it finds the right line for change(s)
        based on keywords or key lines
        Call updateMethod()
    }
}

updateMethod():
{
    // Each method keeps track of the number of
    // changes that is needed, which is used for the following
    // loop
    Repeat for all identified changes
    {
        look for the key line to transform
        // Example: this.serviceHandler in the ST code
        // in Figure 3
        Replace the “original line” with the corresponding
        new line of code and/or insert a new code segment
        // see the example of HS/HA in Figure 3: the change
        // here is this.dataAcceptor for this.serviceHandler.
        // In addition, two additional lines are added to the
        // HS/HA design to start HS/HA
    }
}
```

A graphical user interface (GUI) in Python [Python06] was developed for ease-of-use which is an important feature in practice. Python programs generally take less time to develop. Python is suited as a "glue" language and is good for configuration management. Figure 4 shows the starting GUI of the tool. The selection of target system really depends on the user. The general guidelines and pros and cons are well documented. But it may be useful to compare them by collecting quantitative or more concrete data through the executable systems.

The output file contains the generated Java program. Figures 5 and 6 are snapshots of running a translated Java program using the HS/HA pattern. The 3 value depicted in Figure 5 is the number of worker threads.

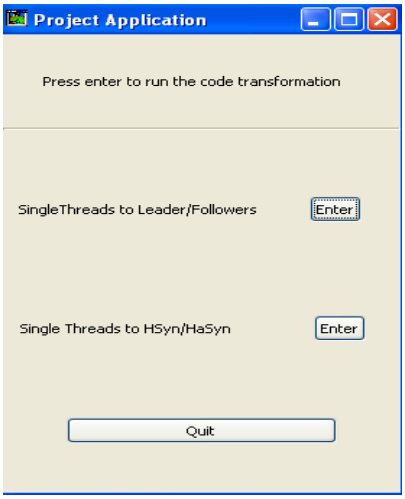

Figure 4. Starting GUI for Program Transformation

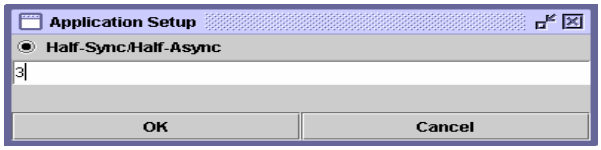

Figure 5. A Java Application Using the HS/HA Pattern (I)

5. Conclusions

Quality attributes or non-functional requirements are important factors in evolution, including maintainability and productivity. Evolution starting at the architecture level has better leverage to address those quality issues. In practice, productivity or evolution time needed usually has a dominating role for enterprises. This paper presented an architecture-centric program transformation approach with an aim to address both the traditional software quality and productivity issues. The target application area is distributed and concurrent systems.

Our initial tool was developed specifically based on an existing system under study, even though our analysis was conducted with a much larger scope. However, the tool can be enhanced for general Java programs. The concept could also be applied to other areas where multiple candidate patterns are potential architecture alternatives and especially the tool can be reused multiple times.

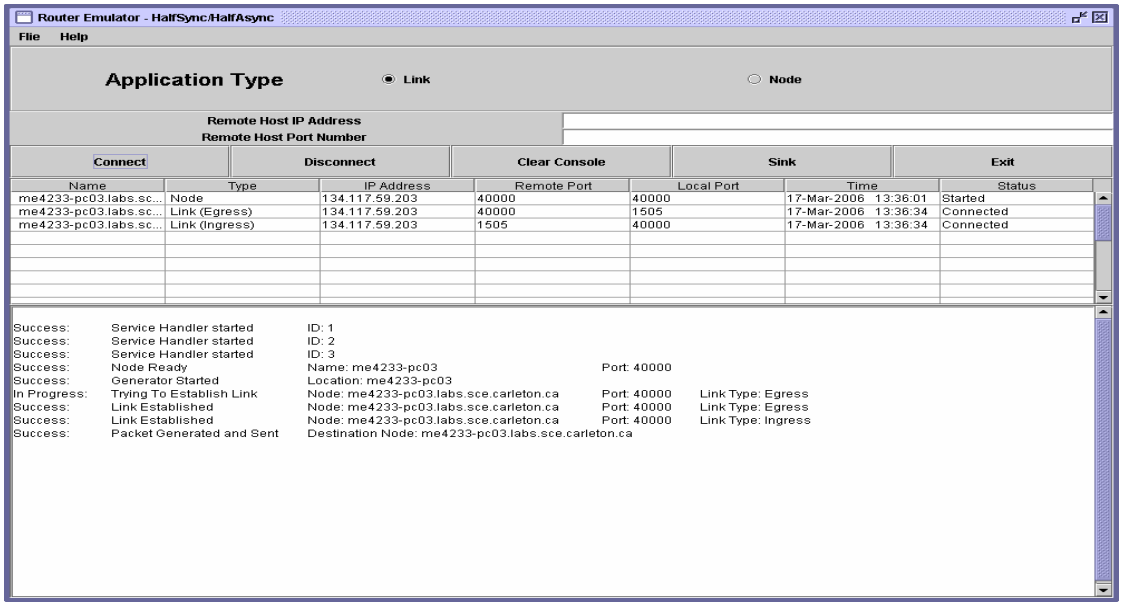

Figure 6. A Java Application Using the HS/HA Pattern (II)

References:

[Alhussaini04] Alhussaini, A., et al, A., *Software Restructuring and Performance Evaluation*, Project Report 2004, Dept. of Systems & Computer Eng., Carleton Univ., Ottawa, Canada.

[Barbacci03] Barbacci, M, et al, *Quality Attribute Workshops (QAW) : 3rd Edition*, Technical Report CMU/SEI-2003-TR-016, 2006.

[Balasubramaniam05] Balasubramaniam, B., Elankeswaran, P., Gopalasundaram, U., and Selvarajah, K., *Program Transformation and Building Reusable Components with Design Patterns*, Project Report 2005, Dept. of Systems & Computer Eng., Carleton Univ., Ottawa, Canada.

[Gamma95] Gamma, E., Helm, R., Johnson, R., Vlissides, J, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[Kazman98] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J., "The Architecture Tradeoff Analysis Method", *Proc. of the 4th Int'l Conf. on Engineering of Complex Computer Systems*, Aug. 1998, pp. 68–78.

[Lee04] Lee, J.-C. and Zhang, X., *Performance Investigation of a Network System on Different Linux Kernels*, Project Report 2004, Dept. of Systems & Computer Eng., Carleton Univ., Ottawa, Canada.

[Lung00] Lung, C.-H. and Kalaichelvan, K., "A Quantitative Approach to Software Architecture Sensitivity Analysis", *Int'l Journal of Software Eng and Knowledge Eng*, vol. 10, no. 1, Feb 2000, pp. 97–114.

[Lung03] Lung, C.-H., "Variability Analysis for Communications Software", *Proc. of the Int'l Workshop on Software Variability Management (SVM), International Conf. on Software Eng.*, May 2003, pp. 30–33.

[Lung04a] Lung, C.-H., Zhao, Q, Xu, H., Mar, H., Kanagaratnam, P., "Experience of Communications Software Evolution and Performance Improvement with Patterns", *Proc. of IASTED Software Engineering*, Feb. 2004.

[Lung04b] Lung, C.-H. and Zhao, Q., "Pattern-Oriented Reengineering of a Network System", *Journal of Systemics, Cybernetics and Informatics*, volume 2, no. 5, 2004.

[Lung06] Lung, C.-H., Balasubramaniam, B., Selvarajah, K., Elankeswaran, P, Gopalasundaram, U., "Architecture-Centric Software Prototype Generation: an Experimental Study", *submitted for publication*.

[Python06] The Python Programming Language, http://www.python.org/, Accessed April 2006.

[Schmidt00] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[Wu03] Wu, Pengfei, *A Performance Model for a Network of Prototype Software Routers*, MASc Thesis, Dept. of Systems & Computer Eng., Carleton University, Ottawa, Canada, July 2003.