to offer is the ability to explore a range of configurations and workload mixes, and to detect the causes of performance limitations just by solving the model, before proceeding to change the real system.

4. Software Designer's Interface

Our experience shows that software developers seldom use performance engineering tools - they rather depend on the services of specialized performance analysis groups in their own company, which leads to rather long delays in getting useful feedback. As we found out in the case of ServiceProvider, the software developers would like to use a performance tool tailored to their specific application, which could produce rapid feedback (as for example to predict the performance for a new mix of services, or the effect of changing the processing times of some software components). Here is a list of requirements for such a tool:

- to hide the performance model;
- to accept input parameters in the software domain (refering to the view from Figure 1) such as: execution times of different processes on behalf of a given type of service, length of various critical sections, number of replications for different processes, etc.;
- to control the execution of the model solver for a given range of service mixes and configurations, and to relieve the developers from the tedious task of managing model and result files;
- to provide a well chosen subset of results in a meaningful form (graphs, charts or tables);
- to provide a simple diagnosis of performance problems, such as a ranking of potential bottlenecks.

Today's general-purpose performance tools, such as analytic solvers, simulation packages and experiment controllers are built to satisfy the needs of the performance analysts. The inputs and outputs of these tools are expressed in the performance modelling domain, and they produce usually a large amount of results. A performance specialist has to analyze these results, extract the useful conclusions and present them to the interested parties in a concise, meaningful way (usually as graphs, charts or tables).

We propose to put in the hands of the software developers a so called Software Developer Interface (SDI), which hides the complexity of performance modelling and analysis of the system on which the developer works on, translates the inputs from the software to the performance domain, and the outputs the other way around. We are not proposing to change the performance engineering approach or the way the models are built and solved, but to add an abstraction layer between the developer and the performance model. By adding such a layer, we fill the semantic gap between the design and performance analysis domain and encapsulate the performance model and its analysis. A performance analyst still has to build the model, to plan the experiments, to specialize the SDI and to perform changes to the model when necessary. We are currently developing such an SDI for our system. A very first solution was implemented in the scripting language Perl and is completely tailored to the ServiceProvider model. The script obtains the input from the user, runs the experiments, manages the model changes and the result files, extracts the desired results and presents them in a pre-defined condensed form. Experience with the first SDI will lead to a set of requirements for a more general-purpose SDI, one that can be easily tuned to the needs of a certain application. .

5. References

[1] C.M. Woodside. "Throughput Calculation for Basic Stochastic Rendezvous Networks". *Performance Evaluation*, vol.9(1), April 1988.

[2] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar. "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software". *IEEE Transactions on Computers*, 44(1):20-34, January 1995.

[3] J.A. Rolia and K.C. Sevcik. "The Method of Layers". *IEEE Trans. on Software Engineering,* 21(8):689-700, August 1995.

[4] Greg Franks, Alex Hubbard, Shikharesh Majumdar, Dorina Petriu, Jerome Rolia and Murray Woodside. "A toolset for performance engineering and software design of client-server systems". *Performance Evaluation*, 24(1-2):117-135, November 1995.

[5] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, C.M. Woodside. "Performance Analysis of Distributed Server Systems". In *Proceedings of the 6th International Conference on Software Quality*, pp. 15-26, Ottawa, Canada, October 1996.

[6] J.Dilley, R.Friedich, T.Jin, J.Rolia, "Measuremnt Tool and Modelling Techniques for Evaluating Web Server Performance" in *Lectures Notes in Computer Science*, vol.1245, Springer, pp.155-168, R.Marie, B.Plateau, M.Calzarosa, G.Rubino (eds), Proc. of 9-th Int. Conference on Modelling Techniques and Tools for Performance Evaluation, June 1997.

[7] C.U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[8] Mary Shaw. "Some patterns for Software Architecture". In J. Vlissides, J. Coplien, and N. Kerth (eds), *Pattern Languages of Program Design2*, Addison-Wesley, 1996, pp.255-269.

[9] M.Shaw, D.Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[10] J.E.Neilson, C.M.Woodside, D. Petriu, and Shikharesh Majumdar. "Software bottlenecking in client-server systems and rendezvous networks". *IEEE Transactions on Software Engineering*, 21(19):776-782, September 1995.

[11]*** *Quantify User's Guide*. Rational Inc., 1995.

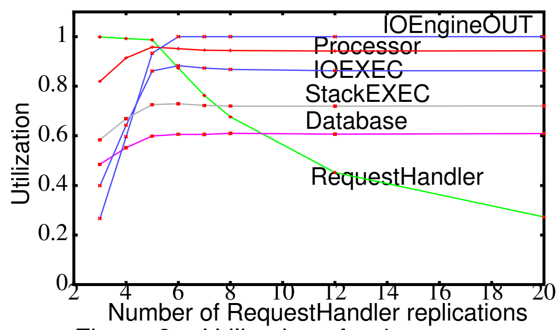

Figure 9a. Utilizations for 4 processors

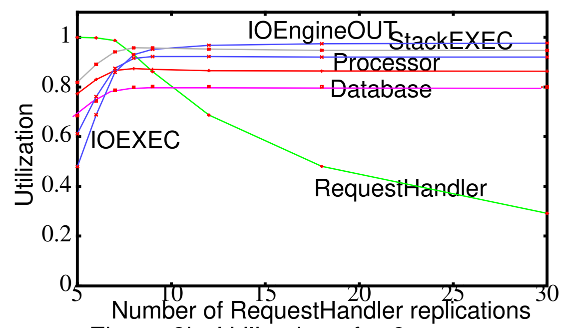

Figure 9b. Utilizations for 6 processors

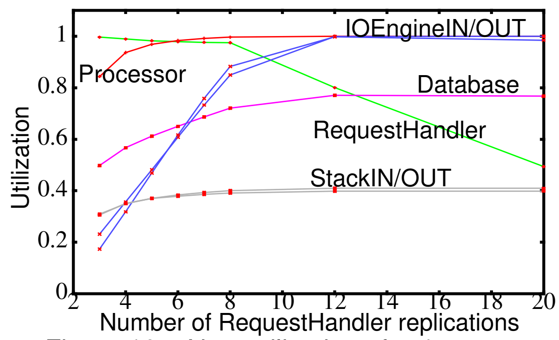

Figure 10a. New utilizations for 4 processors

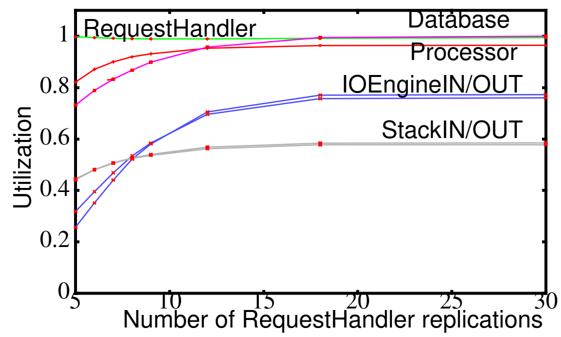

Figure 10b. New utilizations for 6 processors

do not reach the maximum utilization level. IOEngineOut and IOEngineIN reach critical levels of utilization, affected by serialization constraints in the software architecture which are starting to surface (Figure 9a). The serialization is primarily due to the fact that the input and output halves of IOEngine and Stack processes do not run concurrently, being executed by a single execution thread. Another serialization is caused by the interprocess communication using ShMem1, for which all IOEngineIN, RequestHandler and IOEngineOUT contend for. Thus, with increasing processing power, the system bottleneck is moving from hardware to software. This trend is even more visible in the case of the 6-processor configuration, as shown in Figure 9b. The bottleneck has definitely shifted from hardware to software, where the limitations in performance are due to constraints in the software architecture.

We have run a new set of experiments after eliminating the serialization constraints at both the Stack and the IOEngine processes, by removing both StackEXEC and IOEXEC tasks. Yet, no major performance improvement was observed. By examining the percentage of time IOEXEC is actually executing on the processor and its utilization (which includes waiting for lower level servers) we have found that about 90% of IOEngine utilization is spent waiting for the buffer ShMem1 and only the rest of 10% is due to useful work.

In an attempt to reduce the contention for ShMem1, we modified further the "deserialized" LQN model by splitting ShMem1 into two shared buffers, such that IOEngineIN and RequestHandler contend for one buffer for incoming requests, and IOEngineOUT and RequestHandler contend for the other buffer for outgoing replies.

Having applied these changes to the model, we solved the resulting LQN model and obtained the utilization results for 4-processor (Figure 10a) and 6-processor (Figure 10b) configurations, respectively. The processor utilization and the maximum achievable throughput went up. The throughput increase was small (only 2.7%) in the case of 4 processors, and somehow larger (of 10.3%) in the case of 6 processors, where there was more unused processing power.

Consequently, we have recommended to the development team to use two semaphores instead of one for the input and output message queues in the shared buffer ShMem1, and to split Stack and IOEngine processes into two concurrent processes each. Another important conclusion is the fact that the new software architecture doesn't scale up for hardware configurations with 6 processors or more. As we can see from Figure 10b, a new software bottleneck has emerged in the 6-processor case, namely the Database process (which is 100% saturated). The new bottleneck, caused by a low-level server, has propagated upwards, saturating all the software processes that are using it (all RequestHandler replications)

The final conclusion of the performance analysis is that different configurations will have different bottlenecks, and by solving the bottleneck in one configuration, we shift the problem somewhere else. What performance modelling has

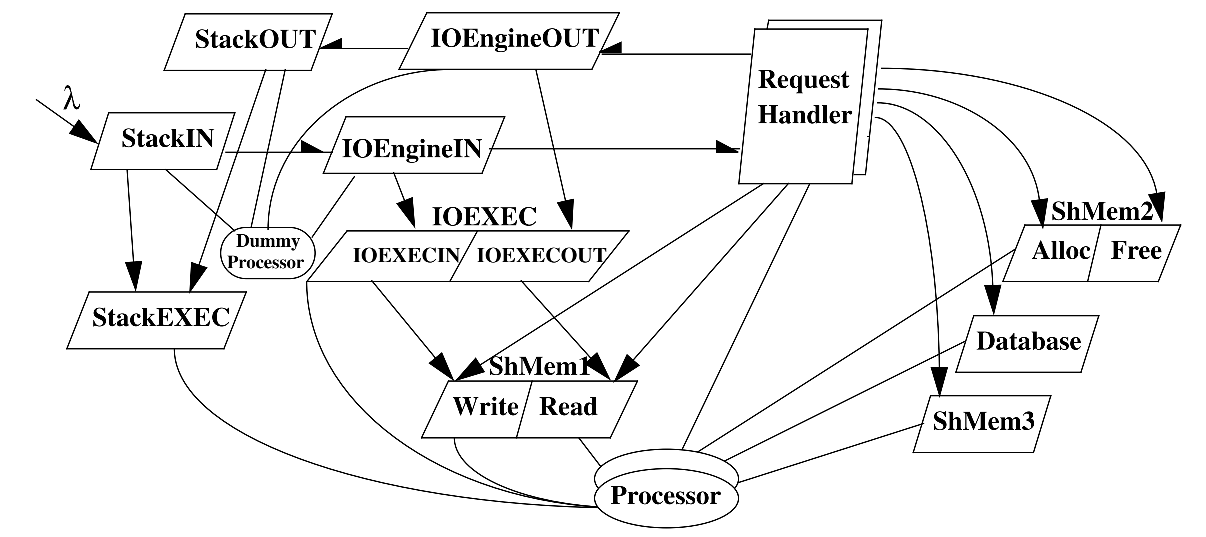

Figure 7. LQN base model of the system

By "configuration" we understand more specifically the number of processors in the multiprocessor system and the number of RequestHandler software replications. We tried to answer this question by exploring a range of configurations for a given service mix, determining for each the highest achievable throughput. Then the configurations whose maximum throughput are lower then the required values are discarded. The cheapest configuration that can insure satisfactory throughput and response time at an operating point below saturation will be chosen. Running the LQN model is a more efficient way to explore different configurations under a wide range of workloads then to run and measure the real system for all the cases under consideration.

The first step to performance analysis is to build the LQN model and to determine its parameters (average service time for each entry, and average number of visits for each request arc). To obtain the workload parameters, we have made use of measurements using *Quantify* [11] and the Unix utility *top*. The measurements for *Quantify* were obtained at very low loads of around a couple of requests/second, whereas *top* provided us with utilization figures for very high loads of hundreds of requests/second, close to the actual operating point. Measurements were run in a lab on two different hardware configurations, for one and four processors. We have used the measurements from *Quantify* to determine the execution model parameters, and the utilization results from *top* to validate our system. The utilizations measured and obtained from the model were very close. A more rigorous validation was hindered by the lack of response time measurements.

Although we have modelled the system for two types of services, this paper reports only some of the results for a single type of service due to space limitations. Since maximum achievable lossless request rate was the major concern, we have proceeded by closing the model and changing all asynchronous requests to forwarding as explained earlier. The closed model was run at saturation in order to get the maximum throughput. The model was analysed for three hardware configurations: with one, four and six processors, respectively. We chose one and four processors since the actual system had been run for such configurations, and six processors to see how the software architecture scales up.

When running the system on a single processor (see Figure 8), the replication of the RequestHandler does not increase the maximum achievable throughput. This is due to the fact that the processor is the system bottleneck. As known from [10], the replication of software processes brings performance rewards only if there is unused processing capacity (which is not the case here).

A software server is defined to be utilized for the duration of its service and that of all its nested services requested from lower level servers (including any queueing delays while waiting for lower level servers).

Interestingly enough, in the 4-processor case we notice that with more processing capacity in the system, the processors

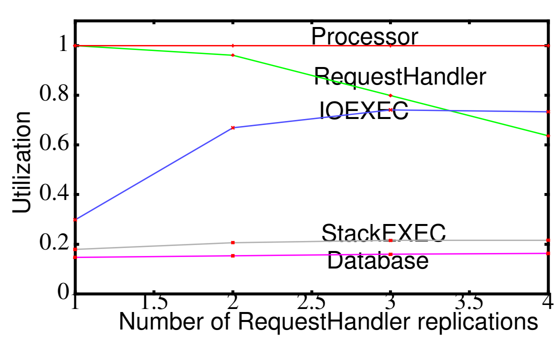

Figure 8. Utilizations for 1 processor

and squares are shared data stores). This architectural view undergoes a series of transformations (as shown in Figures 2 to 6) to end up as a LQN performance model (Figure 7).

As we found out, the software developers would like to use performance tools capable to produce quantitative answers to their questions, as long as they are spared from learning performance modelling techniques and from building performance models. The developers would like to express their questions and get the answers in the software domain (as opposed to the performance modelling domain).

This paper addresses the need to make performance analysis more accessible by hiding the complexity of the performance modell behind an interface which is able to translate back and forth between the software and the performance domains. In order to present our practical experience, the paper will describe how the LQN model was built in section 2, will discuss the most interesting analysis results in section 3, and will describe our first attempt to make performance analysis more accessible to the software developers in section 4.

2. LQN model of the system under study

The real time execution path we have modelled starts from the moment a request arrives to the system and ends after the service was completely processed and a reply was sent back. As shown in Figure 1, a request is passed through several stages of a pipeline: from the StackProcess to the IOEngine to the RequestHandler and all the way back. The main processing is done by the RequestHandler, which accesses a real-time database to fetch an execution "script" for the desired service, then executes the steps of the script accordingly. The script may vary in size and types of operations involved, and hence the workload varies largely from one type of service to another (by one or two orders of magnitude). Due to experience and intuition, the designers decided from the beginning to allow for multiple replications of the RequestHandler process in order to speed up the system.

The architectural patterns [8] used in the system (i.e., the *pipeline* pattern involving StackProcess, IOEngine and RequestHandler, and the *client/server* pattern involving RequestHandler and Database) provide the starting point to build our LQN model. The interprocess communication in the pipeline between IOEngine and RequestHandler is implemented through shared memory. The multiple replications of the RequestHandler compete for the unique database and common shared memory areas. ShMem1 is used as buffer for incoming requests and outgoing replies, whereas ShMem2 contains request control information.

LQN provides various modelling features. We will review in particular the features that have been used to produce the LQN model shown in Figure 7.

- Software processes shown in the architectural view are mapped directly to LQN software components named *tasks*. We begin by representing every process from Figure 1 by an LQN task. RequestHandler becomes a multi-server, whereas the others are single servers. The message-based communication between StackProcess and IOEngine pipeline stages is asynchronous, whereas that between RequestHandler and Database is synchronous.
- An LQN server may offer a range of services, each one with its own service time and number of visits to other servers. Every such service is modelled as an *entry* of the task (entries are represented as parallelogram "slices"). It is assumed that all the requests for all the entries of a task are queued in a common task queue (not explicitely drawn in the LQN notation).
- Software critical sections controlled by semaphores or locks are modelled as entries of an additional task, as shown in Figure 2.
- The shared-buffer communication between the pipeline stages IOEngine and RequestHandler is modelled by breaking it into two separate features: a semaphore pseudo-task to model the contention for the shared buffer, and an asynchronous request from IOEngine to RequestHandler as shown in Figure 3.
- LQN has an implemented protection against cyclic graphs for deadlock prevention purposes (see Figure 4). In order to get around that, we split IOEngine into two tasks and add an "executive" task to control the thread of execution. IOEXEC insures that the two "half-tasks" IOEngineIN and IOEngineOUT cannot execute simultaneously (see Figure 5).
- Any LQN open model can be turned into a closed model by replacing the open arrival with a fixed number of "pure" client tasks. We use this transformation by adding a very large number of clients in order to saturate the system without overflowing the task queues. In the case of open models with asynchronous messaging, we also must substitute the asynchronous requests with *forwarding* requests in the closed version. In the closed version shown in Figure 6, a Client sends a synchronous request to IOEngine, which processes the request then forwards it to the next task in the pipeline, RequestHandler. IOEngine is ready to continue its activity after forwarding the request, but the Client remains blocked until the forwarded request is completed by RequestHandler. More than one forwarding is allowed in LQN, in which case the Client waits until the last forwarded request is completed.

3. Performance Analysis

The first question of interest to developers was to find the "best" hardware and software configuration that can achieve the desired throughput for a given mix of services.

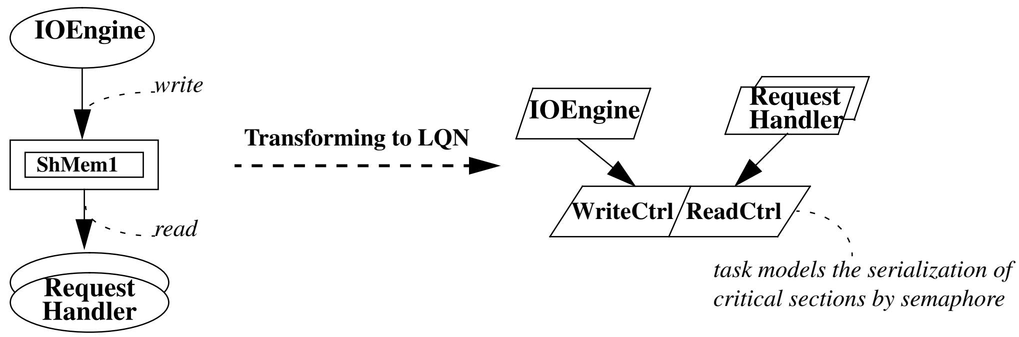

Figure 2: Mapping critical sections to LQN

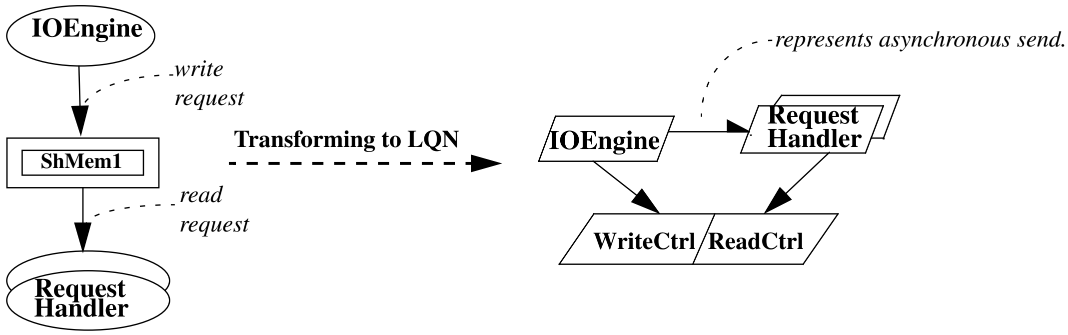

Figure 3. Mapping communication between pipeline stages

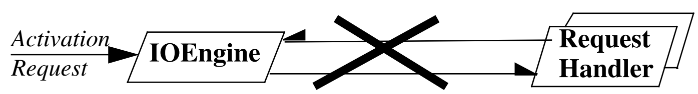

Figure 4: Illegal cyclic LQN model

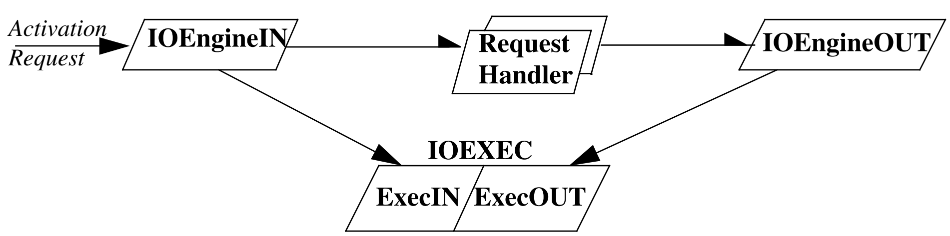

Figure 5. Valid acyclic LQN model

Figure 6: Transforming an open to a closed LQN model

Applying Performance Modelling to a Telecommunication System

Christiane Shousha, Dorina Petriu
Systems & Computer Engineering
Carleton University
Ottawa, Canada K1S 5B6
{shousha | pertriu}@sce.carleton.ca

Anant Jalnapurkar, Kennedy Ngo
Software Engineering Analysis Lab
Nortel
Ottawa, Canada K1Y 4H7
{jalanpur | softrel7}@nortel.ca

Abstract

As telecommunication software systems are growing in size and complexity, the effort of tuning them for performance becomes more costly and time-consuming. Performance modelling offers an efficient means of estimating software design and implementation alternatives, and is much easier to apply than actually changing the real system. Although it is widely recognized that performance engineering is very important to the telecommunications industry, modelling techniques are not easily accessible to software developers. The most important obstacle is the cognitive gap between the software and the performance domains. This paper presents the experience of the authors in modelling an existing telecommunication product and then in using its results for capacity planning and for feeding back recommendations to the software development team. The paper also describes an approach to make performance engineering more accessible to software developers.

1. Introduction

The Layered Queueing Network model (LQN) [1 to 6] has proven more natural to use than the classic queueing networks (QN) when studying the performance of software systems. LQN is an extended queueing network, where a server may require nested services from other servers in order to fulfil the requests of its clients. Thus, an LQN server can act both as client and as server to other components. LQN provides a more direct mapping from the software domain to the performance model, since all software processes are usually mapped directly to LQN components.

An LQN model is represented by an acyclic graph (as in Figure 7) whose nodes are either software tasks (drawn as parallelograms) or hardware devices (drawn as ellipses). Both software and hardware nodes can be single or multi-servers. An arc represents a service request from a "client" node to a "server" node. Such a request can be synchronous (i.e., the client is blocked until the server replies) or asynchronous (the client continues its work in parallel with the server). More details on the LQN features are given in section 2. The LQN toolset provides both analytical and simulation tools [4].

The emerging telecommunications field of intelligent networks is changing at a very fast pace. We conducted performance modelling and analysis of an existing telecommunication product, ServiceProvider, which is responsible for developing, provisioning and maintaining various intelligent network services, as well as for accepting and processing real-time requests for these services. According to the Software Performance Engineering methodology [7], we first identified the critical real-time scenario which correspons to service request processing, then build an LQN model. We used the model for capacity planning and for feeding back recommendations to the software development team. Another outcome of the experiment was to obtain feedback from industry about the usability of performance engineering techniques in the software development process. Although it is widely recognized that performance engineering is very important to the telecommunications industry, modelling techniques are not easily accessible to software developers, mainly because of the cognitive gap between the software and performance domains. For example, Figure 1 shows the software architecture of the real-time subsystem under study as seen by developers. (The notation from [9] is used: elipses are software processes

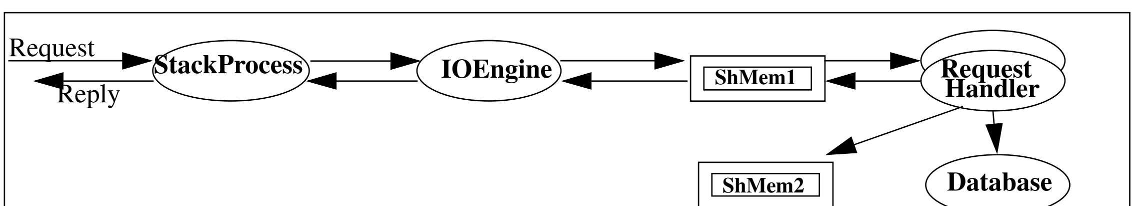

Figure 1. System Software Architecture