Architecture-Based Performance Analysis Applied to a Telecommunication System

Dorina Petriu, Christiane Shousha, and Anant Jalnapurkar

Abstract – Software architecture plays an important role in determining software quality characteristics, such as maintainability, reliability, reusability, and performance. Performance effects of architectural decisions can be evaluated at an early stage by constructing and analyzing quantitative performance models, which capture the interactions between the main components of the system as well as the performance attributes of the components themselves. The paper proposes a systematic approach to building Layered Queueing Network (LQN) performance models from a UML description of the high-level architecture of a system, and more exactly from the architectural patterns used for the system. The performance model structure retains a clear relationship with the system architecture, which simplifies the task of converting performance analysis results into conclusions and recommendations related to the software architecture. In the second part of the paper, the proposed approach is applied to a telecommunication product for which a LQN model is built and analyzed. The analysis shows how the performance bottleneck is moving from component to component (hardware or software) under different loads and configurations, and exposes some weaknesses in the original software architecture, which prevent the system from using the available processing power at full capacity due to excessive serialization.

Index Terms – software performance analysis, layered queueing networks, software architecture, architectural patterns, Unified Modeling Language (UML).

1. Introduction

Performance characteristics (such as response time and throughput) are an integral part of the quality attributes of a software system. There is a growing body of research that studies the role of software architecture in determining different quality characteristics in general [12], [1], and performance characteristics in special [15], [16]. Architectural decisions are made very early in the software development process, therefore it would be helpful to be able to assess their effect on software performance as soon as possible.

This paper contributes toward bridging the gap between software architecture and early performance analysis. It proposes a systematic approach to building performance models from the high-level software architecture of a system, which describes the main system components and their interactions. The architectural descriptions on which the construction of a performance model is based must capture certain issues relevant to performance, such as concurrency and parallelism, contention for software resources (as, for example, for software servers or critical sections), synchronization and serialization, etc.

Frequently used architectural solutions are identified in literature as *architectural patterns* (such as pipeline and filters, client/server, client/broker/server, layers, master-slave, blackboard, etc.) [3], [12]. A pattern introduces a higher-level of abstraction design artifact by describing a specific type of collaboration between a set of prototypical components playing well-defined roles, and helps our understanding of complex systems. The paper proposes a systematic approach to building a performance model by transforming each architectural pattern employed in a system into a performance sub-model. The advantage of using patterns is that they are already identified and catalogued, so we can build a library of transformation rules for converting patterns to performance models. If, however, not all components and interactions of a high-level architecture are covered by previously identified architectural patterns, we can still describe the remaining interactions as UML mechanisms [2] and proceed by defining ad-hoc transformations into performance models.

The formalism used for building performance models is the *Layered Queueing Network* (LQN) model [11, 17, 18], an extension of the well-known Queueing Network model. LQN was developed especially for modelling concurrent and/or distributed software systems. Some LQN components represent software processes, others hardware devices. One of the most interesting performance characteristics of such systems is that a software process may play a dual role, acting both as a client to some processes /devices, and as a server to others (see section 3 for a more detailed description). Since a software server may have many clients, important queueing delays may arise for it. The server may become a software bottleneck, thus limiting the potential performance of the system. This can occur even if the devices used by the process are not fully utilized. The analysis of an LQN model produces results such as response time, throughput, queueing delays and utilization of different software and hardware components, and indicates which components are the system bottleneck(s). By understanding the cause for performance limitations, the developers will be able to concentrate on the system's "trouble spots" in order to eliminate or mitigate the bottlenecks. The analysis of LQN models for various alternatives will help in choosing the "right" changes, so that the system will eventually meet its performance requirements.

Software Performance Engineering (SPE) is a technique introduced in [14] that proposes to use quantitative methods and performance models in order to assess the performance effects of different design and implementation alternatives, from the earliest stages of software development throughout the whole lifecycle. LQN modelling is very appropriate for such a use, due to fact that the model structure can be derived systematically from the high-level architecture of the system, as proposed in this paper. Since the high-level architecture is decided early in the development process and does not change frequently afterwards, the structure of the LQN model is also quite stable. However, the LQN model parameters (such execution times of the high-level architectural components on behalf of different types of system requests) depend on low-level design and implementation decisions. In the early development stages, the parameter values are estimations based on previous experience with similar systems, on measurement of reusable components, on known platform overheads (such as system call execution times) and on time budgets allocated to different components. As the development progresses and more components are implemented and measured, the model parameters become more accurate, and so do the results. In [14] it is shown that early performance modelling has definite advantages, despite its inaccurate results, especially when the model and its parameters are continuously refined throughout the software lifecycle.

LQN was applied to a number of concrete industrial systems (such as database applications [5], web servers [7], telecommunication systems, etc.) and was proven to be useful for providing insights into performance limitations at software and hardware levels, for suggesting performance improvements in different development stages, for system sizing and for capacity planning. In this paper, LQN is applied to a real telecommunication system. Although the structure of the LQN model was derived from the high-level architecture of the system, which was chosen in the early development stage, we used model parameters obtained from prototype measurements, which are more accurate than the estimated values available in pre-implementation phases. The reason is that we became involved with the project when the system was undergoing performance tuning, and so we used the best data available to analyze the high-level architecture of the system (which was unchanged from the early design stages). We found some weaknesses in the original architecture due to excessive serialization, and used the LQN model to assess different architectural alternatives, in order to improve the performance by removing or mitigating software bottlenecks.

The paper proceeds as follows: section 2 discusses architectural patterns and the UML notation [2] used to represent them. Section 3 gives a brief description of the LQN model. Section 4 proposes transformations of the architectural patterns into LQN sub-models. Section 5 presents the telecommunication system case study and its LQN model. Section 6 analyzes the LQN model under various loads and configurations, shows how the bottleneck moves around the system and proposes improvements to the system. Section7 gives the conclusions of the work.

3. Architectural Patterns

According to [1], a *software architecture* represents a collection of computational *components* that perform certain functions, together with a collection of *connectors* that describe the interactions between components. A *component type* is described by a *specification* defining its functions, and by a set of *ports* representing logical points of interaction between the component and its environment. A *connector type* is defined as a set of *roles* explaining the expected behaviour of the interacting parties, and a *glue* specification showing how the interactions are coordinated.

A similar, even though less formal, view of a software architecture is described in the form of *architectural patterns* [3, 13] which identify frequently used architectural solutions, such as pipeline and filters, client/server, client/broker/server, master-slave, blackboard, etc. Each architectural pattern describes two inter-related aspects: its *structure* (what are the components) and *behaviour* (how they interact). In the case of high-level architectural patterns, the

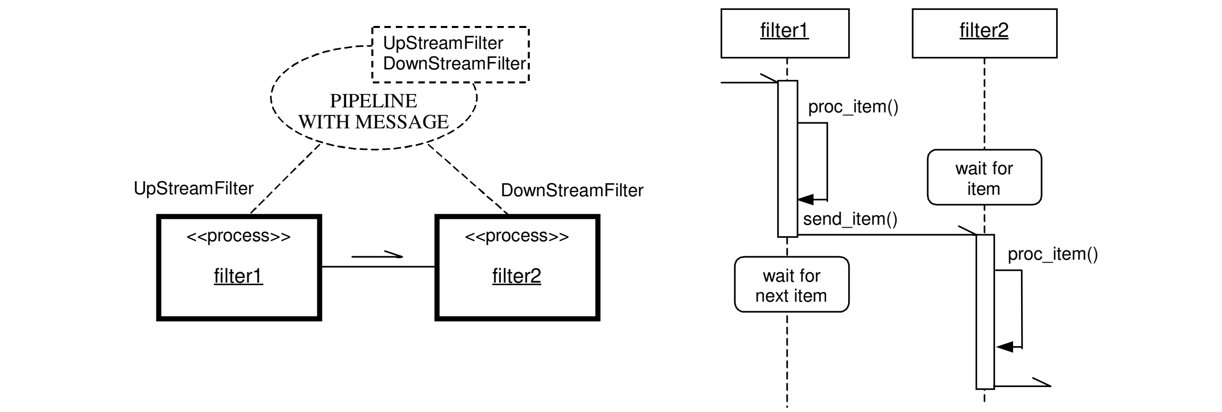


Figure 1. Structural and behavioural views of the collaboration for PIPELINE WITH MESSAGE

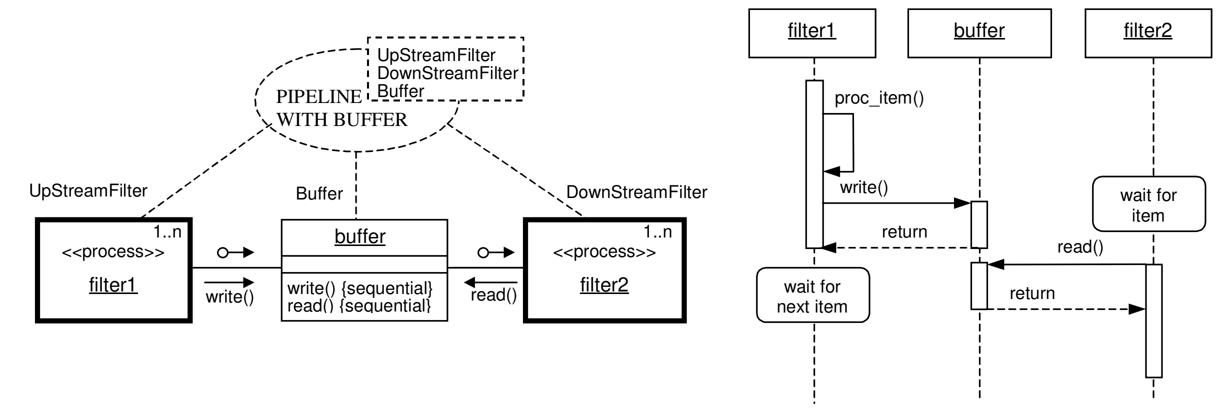


Figure 2. Structural and behavioural view of the collaboration PIPELINE WITH BUFFER

components are usually concurrent entities that execute in different threads of control, compete for resources, and interact in a prescribed manner which may require some kind of synchronization. These are aspects that contribute to the performance characteristics of the system, and therefore must be captured in the performance model.

The paper proposes to use high-level architectural patterns as a basis for translating software architecture into performance models. A subset of such patterns, which are later used in the case study, are described in the paper in the form of UML *collaborations* (not to be confused with UML *collaboration diagrams*, a type of interaction diagrams). According to the authors of UML, a collaboration is a notation for describing a *mechanism* or *pattern*, which represents "a society of classes, interfaces, and other elements that work together to provide some cooperative behaviour that is bigger than the sum of all of its parts" [2]. A collaboration has two aspects: structural and behavioural. Fig. 1 and 2 illustrate these aspects for two alternatives of the *pipeline and filters* pattern. Each figures contains on the left a UML class/object diagram describing the pattern structure, and on the right a UML sequence diagram illustrating the pattern behaviour. A brief explanation of the UML notation used in the paper is given below (see [2] for more details). The notation for a class or object is a rectangle indicating the class/object name (the name is underlined for objects); the rectangle may contain optionally a section for the class/object operations, and another one for its attributes. The multiplicity of the class/object is represented in the upper right corner. A rectangle with thick lines represents an active class/object, which has its own thread of control, whereas a rectangle with thin lines represents a passive one. An active object may be implemented either as a process or as a thread (identified by the stereotype <<process>> or <<thread>>, respectively). The constraint {sequential} attached to the operations of a passive object, as in Fig. 2, indicates that the callers must coordinate outside the passive object (for example, by the means of a semaphore) so that only one calls the passive object's operations at any given time. The UML symbol for collaboration is an ellipse with a dashed line that may have an "embedded" rectangle showing template classes. The collaboration symbol is connected with the classes/objects with dashed lines, whose labels indicate the roles played by each component. A line connecting two objects, named link, represents a relationship between the two objects which interact by exchanging messages. Depending on the kind of interacting objects (passive or active), UML messages may represent either operation calls, or actual messages sent between different flows of control. Links between objects may be optionally annotated with arrows showing the name and type of messages exchanged. For example, in Fig.1 an arrow with a half arrowhead between the active objects filter1 and filter2 represents an asynchronous message, whereas in Fig.2 the arrows with filled solid arrowheads labeled "write()" and "read()" represent synchronous messages implemented as calls to the operations indicated by the label. When relevant, the "object flow" carried by a message is represented by a little arrow with a circle (as in Fig.2), while the message itself is an arrow without circle. A synchronous message implies a reply, therefore it can carry objects in both directions. For example, in Fig.2, the object flow carried by the message read() goes in the reverse direction than the message itself.

A sequence diagram, such as on the right side of Fig.1 and 2, shows the messages exchanged between a set of objects in chronological order. The objects are arranged along the horizontal axis, and the time grows along the vertical axis, from top to bottom. Each object has a lifeline running parallel with the time axis. On the lifeline one can indicate the period of time during which an object is performing an action as a tall thin rectangle called "focus of control", or the state of the object as a rectangle with rounded corners called "state mark". The messages exchanged between objects (which can be asynchronous or synchronous) are represented as horizontal directed lines. An object can also send a message to itself, which means that one of its operations invokes another operation of the same object.

Architectures using the pipeline and filters pattern divide the overall processing task into a number of sequential steps which are implemented as filters, while the data between filters flows through unidirectional pipes. We are interested here in active filters [3] that are running concurrently. Each filter is implemented as a process or thread that loops through the following steps: "pulls" the data (if any) from the preceding pipe, processes it, then "pushes" the results down the pipeline. The way in which the push and pull operations are implemented may have performance consequences. In Fig.1 the filters communicate through asynchronous messages. A filter "pulls" an item by accepting the message sent by the previous filter, processes the item by invoking its own operation proc_item(), passes the data on to the next filter by sending an asynchronous message, after which goes into a waiting state for the next item. In Fig.2, the filters communicate through a shared buffer (one pushes by writing to the buffer, and the other pulls by reading it). Whereas the filters are active objects with a multiplicity of one or higher, the buffer itself is a passive object that offers two operations, read() and write(), which must be used one at a time (as indicated by the constraint {sequential}).

When defining the transformations from architectural patterns into LQN sub-models, we use both the structural and the behavioural aspect of the respective collaborations. The structural part is used directly, in the sense that each software component has counterpart(s) in the structure of the LQN model (the mapping is not bijective). However, the behavioural part is used indirectly, in the sense that it is matched by the behaviour of the LQN model, but is not represented graphically.

3. LQN Model

LQN was developed as an extension of the well-known Queueing Network (QN) model, at first independently in [17, 18] and [11], then as a joint effort [4]. The LQN toolset presented in [4] includes both simulation and analytical solvers that merge the best previous approaches. The main difference of LQN with respect to QN is that a server, to which customer requests are arriving and queueing for service, may become a client to other servers from which it requires nested services while serving its own clients. An LQN model is represented as an acyclic graph whose nodes (named also *tasks*) are software entities and hardware devices, and whose arcs denote service requests (see Fig.3). The software entities are drawn as parallelograms, and the hardware devices as circles. The nodes with outgoing and no incoming arcs play the role of *pure* *clients*. The intermediate nodes with incoming and outgoing arcs play both the role of client and of server, and usually represent software components. The leaf nodes are *pure servers*, and usually represent hardware servers (such as processors, I/O devices, communication network, etc.) A software or hardware server node can be either a single-server or a multi-server (composed of more than one identical servers that work in parallel and share the same request queue). A LQN task may offer more than one kind of service, each modelled by a so-called *entry*, drawn as a parallelogram "slice". An entry has its own execution time and demands for other services (given as model parameters). Although not explicitly illustrated in the LQN notation, each server has an implicit message queue, where the incoming requests are waiting their turn to be served. Servers with more then one entry still have a single input queue, where requests for different entries wait together. The default scheduling policy of the queue is FIFO, but other policies are also supported. Fig. 3 shows a simple example of an LQN model for a three-tiered client/server system: at the top there are two client classes, each with a known number of stochastic identical clients. Each client sends requests for a specific service offered by a task named Application, which represents the business layer of the system. Each Application entry requires services from two different entries of the Database task, which offers in total three kinds of services. Every software task is running on a processor node, drawn as a circle; in the example, all clients of the same class share a processor, whereas Application and Database share another processor. Database uses also two disk devices, as shown in Fig.3. It is worth mentioning

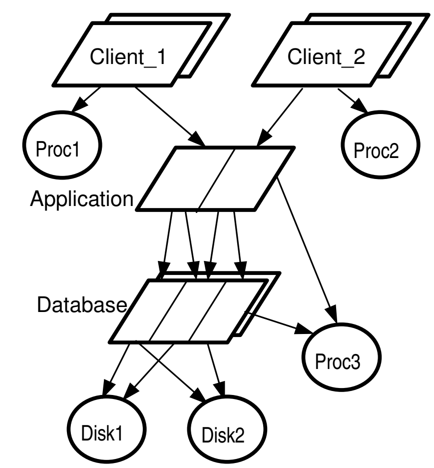
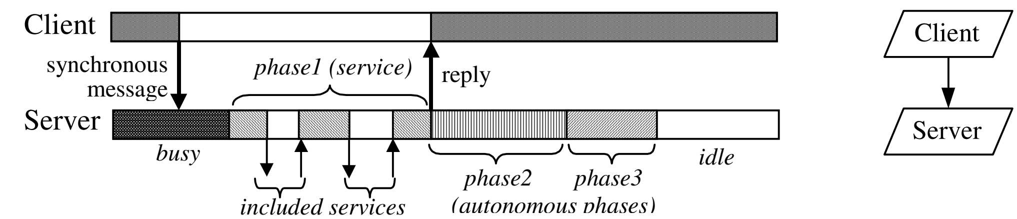


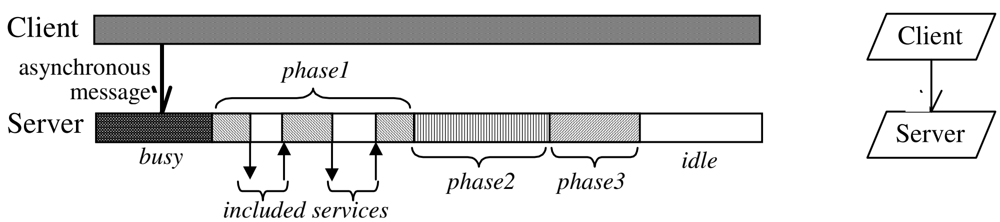
Figure 3. Simple LQN model

that the word *layered* in the name of LQN does not imply a strict layering of the tasks (for example, a task may call other tasks in the same layer, or skip over layers).

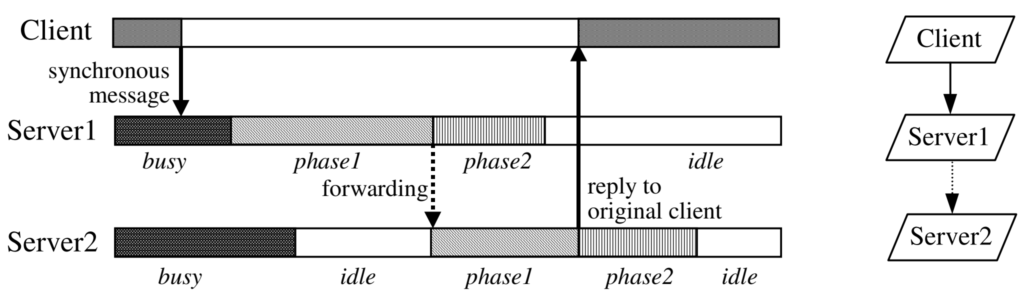
There are three types of LQN messages, synchronous, asynchronous and forwarding, whose effect is illustrated in Fig.4. A *synchronous* message represents a request for service sent by a client to a server, where the client remains blocked until it receives a reply from the provider of service (see Fig.4.a). If the server is busy when a request arrives, the request is queued and waits



a) LQN synchronous message



b) LQN asynchronous message



c) LQN forwarding message

Figure 4. Different types of LQN messages

its turn. After accepting a message, the server starts to serve it by executing a sequence of phases (one or more). At the end of *phase 1*, the server replies to the client, which is unblocked and continues its work. The server continues with the following phases, if any, working in parallel with the client, until the completion of the last phase. (In Fig.4.a, a case with three phases is shown). After finishing the last phase, the server begins to serve a new request from the queue, or becomes idle if the queue is empty. During any phase, the server may act as a client to other servers, asking for and receiving so-called "included services". In the case of an *asynchronous* message, the client does not block after sending the message, and the server does not reply back, only executes its phases, as shown in Fig.4b. The third type of LQN message, named *forwarding message* (represented as a dotted request arc) is associated with a synchronous request that is served by a chain of servers, as illustrated in Fig. 4.c. The client sends a synchronous request to Server1, which begins to process the request, then forwards it to Server2 at the end of phase1. Sever1 proceeds normally with the remaining phases in parallel with Server2, then goes on to another cycle. The client, however, remains blocked until the forwarded request is served by Server2, which replies to the client at the end of its phase 1. A forwarding chain can contain any number of servers, in which case the client waits until it receives a reply from the last server in the chain.

The parameters of a LQN model are as follows:

- customer (client) classes and their associated populations or arrival rates;
- for each software task entry: average execution time per phase;
- for each software task entry seen as a client to a device (i.e., for each request arc from a task entry to a device): average service time at the device, and average number of visits per phase of the requesting entry;

- for each software task entry seen as a client to another task entry (i.e., for each request arc from a task entry to another task entry): average number of visits per phase of the requesting entry;
- for each request arc: average message delay;
- for each software and hardware server: scheduling discipline.

Typical results of an LQN model are response times, throughput, utilization of servers on behalf of different types of requests, and queueing delays. The LQN results may be used to identify the software and/or hardware bottlenecks that limit the system performance under different workloads and configurations. Understanding the cause for performance limitations helps the development team to come up with appropriate remedies.

4. Transformation from architecture to performance models

A software system contains many components involved in various architectural connection instances (each described by a pattern/collaboration), and a component may play different roles in connections of various types. The transformation of the architecture into a performance model is done in a systematic way, pattern by pattern. As expected, the performance of the system depends on the performance attributes of its components and on their interactions (as described by patterns/collaborations). Performance attributes are not central to the software architecture itself, and must be supplied by the user as additional information. They describe the demands for hardware resources by the software components: allocation of processes to processors, execution time demands for each software component on behalf of different types of system requests, demands for other resources such as I/O devices, communication networks, etc. We will specify more clearly what kind of performance attributes must be provided for each pattern/collaboration. The tranformations from the architecture to the performance modelling domain are discussed next.

LQN model for Pipeline and Filters. Fig. 5 and 6 show the transformation into LQN submodels of the two Pipeline and Filters collaborations described in Fig.1 and 2, respectively. The translation takes into account, on one side, the structural and behavioural information provided by the UML collaboration, and on the other side the allocation of software components

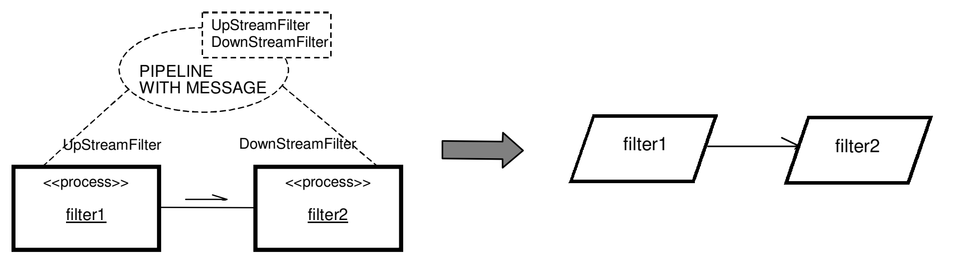
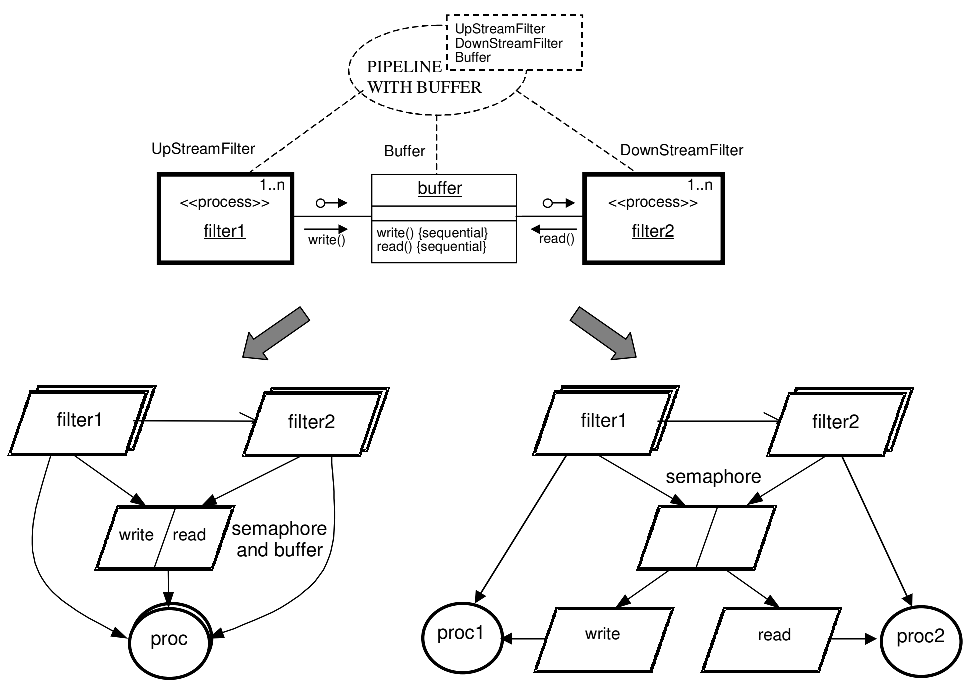


Figure 5. Transformation of the PIPELINE WITH MESSAGE into an LQN submodel



a) All filters are running on the same processor node

b) The filters are running on different processor nodes

Figure 6. Transformation of the PIPELINE WITH BUFFER into an LQN submodel

to processors, which will lead to different LQN submodels for the same pattern (see Fig.6). The tansformation rules are as follows:

a) Each active filter from Fig.5 and 6 becomes an LQN software server with a single entry, whose service time includes the processing time of the filter. The filter tasks will receive an asynchronous message (described in Fig. 4.b) and will execute its phases in response to it. A typical distribution of the work into phases is to receive the message in phase 1, to process it in phase 2, and to send it to the next filter in phase 3.

b) The allocation of LQN tasks to processors mimics the real system. The way the filters are allocated on the same or on different processor nodes does not make any difference for the pipeline with message (reason for which the processors are not represented in Fig.5), but it affects the model for a pipeline with buffer, as explained below.

c) In the case of a pipeline with message (Fig.5) all aspects related to the pipeline connector between the two filters are completely modelled by the LQN asynchronous message. The CPU times for send and receive system calls are added to the execution times of the phases in which the respective operations take place. If we want to model a network delay for the message, it can be represented by a delay attached to the request arc.

d) In the case of a pipeline with buffer (Fig.6), an asynchronous LQN arc is necessary, but is not sufficient to model all the aspects concerning the pipeline connector. Additional LQN elements are required to take into account the serialization delay introduced by the constraint that buffer operations must be mutually exclusive. A third task that plays the role of semaphore will enforce this constraint, due to the fact that any task serializes the execution of its entries. The task has as many entries as the number of critical sections executed by the filters accessing the buffer (two in this case, “write” and “read”). Since the execution of each buffer operation takes place in the thread of control of the filter initiating the operation, the allocation of filters to processors matters. If both filters are running on the same processor node (which may have more than one processor) as in Fig.6.a, then the read and write operations will be executed on the same processor node. Thus, they can be modelled as entries of the semaphore task that is, obviously, co-allocated with the filters. If, however, the filters are running on different processor nodes, as in Fig.6.b, the mutual-exclusive operations read and write will be executed on different processor nodes, so they cannot be modelled as entries of the same task. (In LQN, all entries of a task are executed on the same processor node). The solution is shown in Fig.6.b: we keep the semaphore task for enforcing the mutual exclusion, but its entries are only used to delegate the work to two new tasks, each one responsible for a buffer operation. Each new task is allocated on the same processor as the filter initiating the respective operation.

The LQN models for both pipeline and filters collaborations from Fig. 5 and 6 can be generated with a forwarding message (as in Fig.4.c) instead of an asynchronous one (as in Fig.4.b), if the source of requests for the first filter in a multi-filter architecture is closed instead of open. A closed source is composed of a set of client tasks sending synchronous requests to the first filter, and waiting for a reply from the last filter. Since a LQN task may send a forwarding message exactly at the end of its phase1, all the work done by a filter task must take place in the first phase.

Client-Server Pattern is very frequently used in today's distributed systems. In Fig.7 is illustrated a case where the client communicates directly with the server through synchronous requests (described in Fig.4.a). A server may offer a wide range of services (represented in the architectural view as the server's class methods) each one with its own performance attributes. From a performance modelling point of view it is important not only to identify theses services,

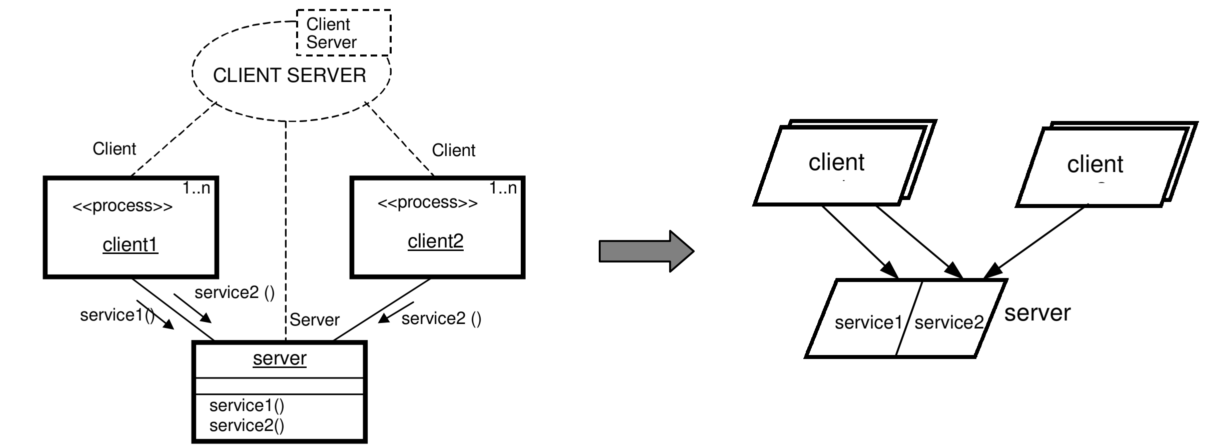


Figure 7. Transformation of the client/server pattern into a LQN submodel

but also to find out who is invoking them and how frequently. The UML class diagram contains a single association between a client and a server, no matter how many server methods the client may invoke. Therefore, we indicate here in addition to the line that represents the client-server association, the messages sent by the client to the server (used mostly in collaboration diagrams) to indicate all the services a client will invoke at one time or another.

There are other ways in which client/server connections may be realized, which are not described in the paper as they do not apply to our case study. A well-known example is the use of midware technology, such as CORBA, to interconnects clients and servers running on heterogeneous platforms across local or wide-area networks. CORBA connections introduce very interesting performance implications and modelling issues [9].

LQN was originally created to model client/server systems, so the transformation from the client-server pattern to LQN is quite straightforward. An LQN server may offer a range of services (object methods in the architectural view), each with its own CPU time and number of visits to other servers (these are performance attributes that must be provided). Each service is modelled as an entry of the server task, as shown in Figure 7, and will contribute differently to the response time, utilization, and throughput of the server. A client may invoke more than one of these services at different times. The performance attributes for the clients include their average CPU time demands, and the average number of calls for each entry of the server. As in the pipeline connection case, the CPU time required to execute the system call for send/receive/reply are added to the service times of the corresponding entries. The allocation of tasks to processors is not shown in Fig. 7, because the transformation does not depend on it. Each LQN task is allocated exactly as its architectural component counterpart.

Critical section. This is a collaboration at a lower-level of abstraction than the previous architectural patterns, but very frequently used. It describes the case where two or more active objects share the same passive object. The constraint {sequential} attached to the methods of the shared object indicates that the callers must coordinate outside the shared object (for example, by the means of a semaphore) to insure correct behaviour. Such synchronization introduces performance delays, and must be represented in the LQN model. For simplicity reasons, Fig.8 illustrates a case where each user invokes only a method of the shared object, but this can be extended easily to allow each user to call a subset of methods.

The transformation of the critical section collaboration produces either the model given in Fig. 8.a or in Fig. 8.b, depending on the allocation of user processes to processor nodes (similar to the pipeline with buffer case). The premise is that the shared object operations are mutually exclusive, that an LQN task cannot change its processor node, and that all the entries of a task are executed on the task's processor node. In the case where all users are running on the same processor node, the shared object operations can be modelled as entries of a task that plays the role of semaphore (see Fig.8.a), which is running on the same processor node as the users. The

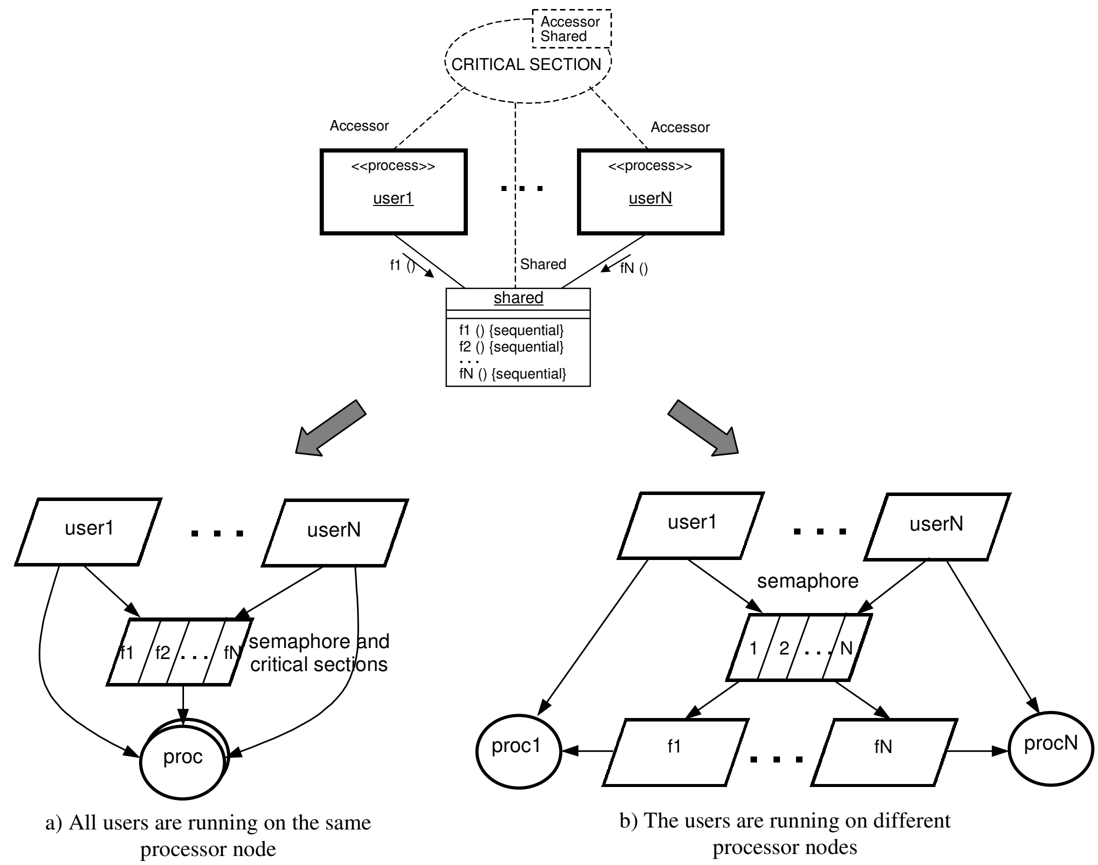


Figure 8. Transformation of the critical section collaboration to a LQN submodel

generalization for allowing a user to call a subset of operations (entries) is straightforward: the user is connected by a request arcs to every entry in the subset.

If the users are running on different processor nodes as in Fig. 8.b, then the shared object operations (i.e., critical sections) are executed by different threads of controls corresponding to different users that are running on different processors. Therefore each operation is modelled as an entry of a new task responsible for that operation that is running on its user's node. (If a user is to call more shared operations, its new associated task will have an entry for every such operation. This means that an operation called by more than one user will be represented by more than one entry.) However, these new tasks must be prevented from running simultaneously,

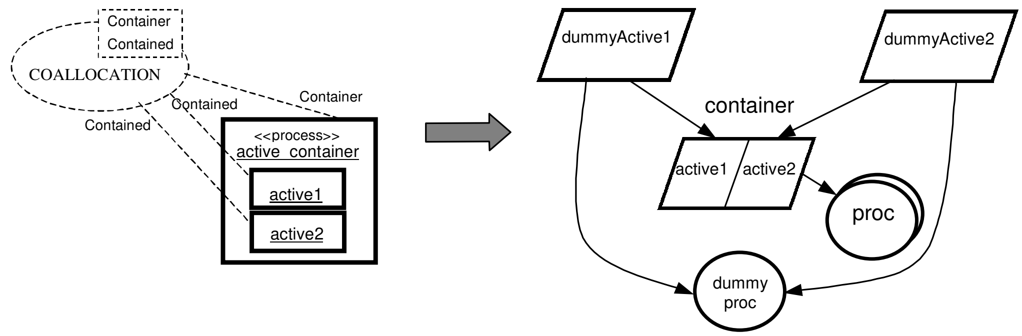


Figure 9. LQN submodel of COALLOCATION collaboration

so a semaphore task, with one entry for each user, is used to enforce the mutual exclusion. An entry of the semaphore task delegates the work to the entries modelling the required operations. The performance attributes to be provided for each user must specify the average execution times for each user outside and inside the critical section separately.

Co-allocation collaboration. Fig. 9 shows a the so-called co-allocation collaboration, where two active objects are contained in a third active object, and are constrained to execute only one at a time. The container object may be implemented as a process. This is an example of architectural connection from our case-study system, which is not necessarily an architectural pattern, but is quite frequently used. The most obvious solution to model the two contained objects as entries of the same task presents a disadvantage: it cannot represent the case where each of the two contained objects has its own request queue. (An LQN task has a unique message queue, where requests for all entries are waiting together). One reason for which we may need separate queues is to avoid cyclic graphs, which could not be accepted by the LQN solver used for this paper. The solution presented in Fig.9 represents each contained active object as a separate "dummy" task that delegates all the work to an entry of the container task, which serializes all its entries. The dummy tasks are allocated on a dummy processor (not to interfere with the scheduling of the "real" processor node).

5. LQN Model of a Telecommunication System

We conducted performance modelling and analysis of an existing telecommunication system 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 [14], we first identified the critical scenarios with the most stringent performance constraints (which correspond in this case to real-time processing of service requests). Next we identified the software components involved in, and the architectural patterns exercised by the execution of the chosen scenarios (see Fig.10).

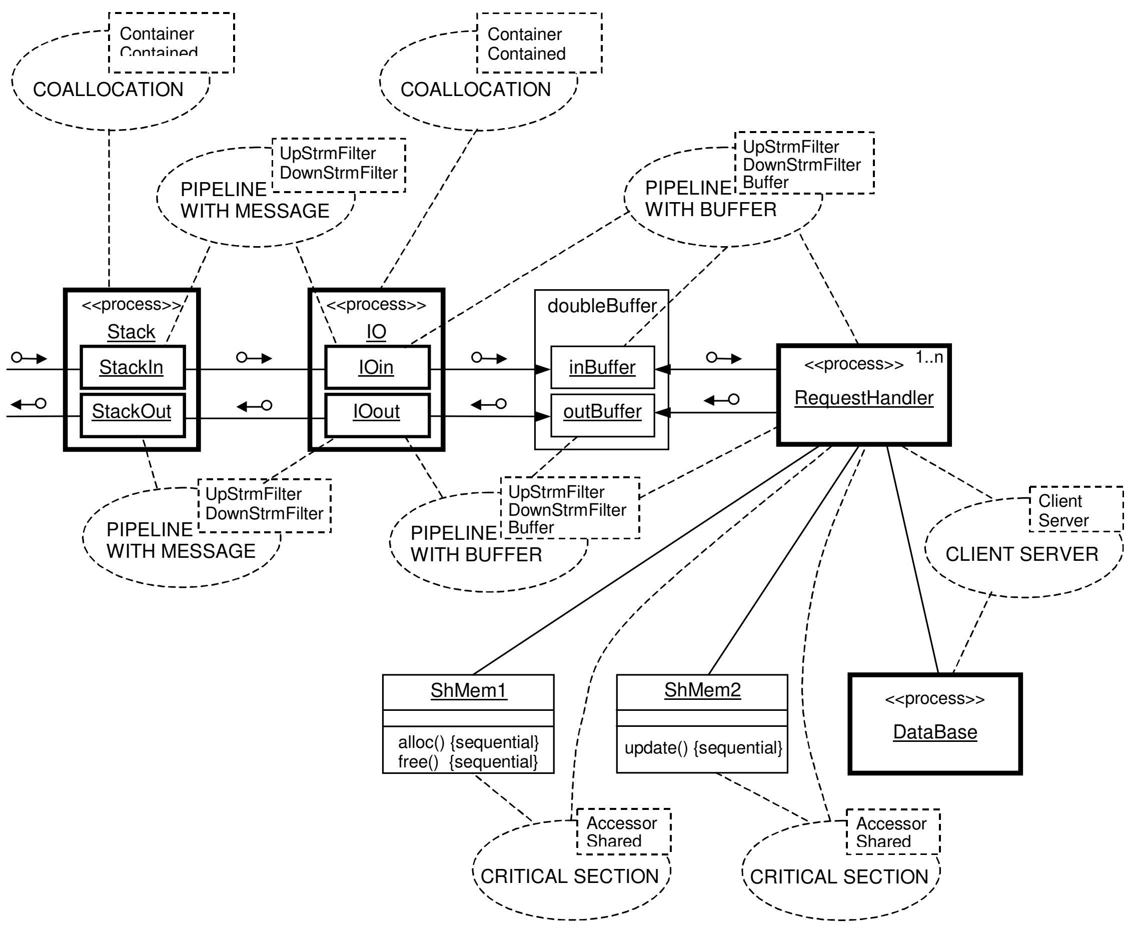


Figure 10. UML model of a telecommunication system

The real time scenario 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 Fig. 10, a request is passed through several filters of a pipeline: from Stack process to IO process to RequestHandler and all the way back. The main processing is done by the RequestHandler (as it can be seen from Fig.12 and Table 1), 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). Based on 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. Two shared objects, ShMem1 and ShMem2, are used by the multiple RequestHandler replications. The system was intended to be run either on a single-processor or on a multi-processor with shared memory. Processor scheduling is such that any process can run on any free processor (i.e., the processors were not dedicated to specific tasks). Therefore, the processor node was modelled as a multi-server. By

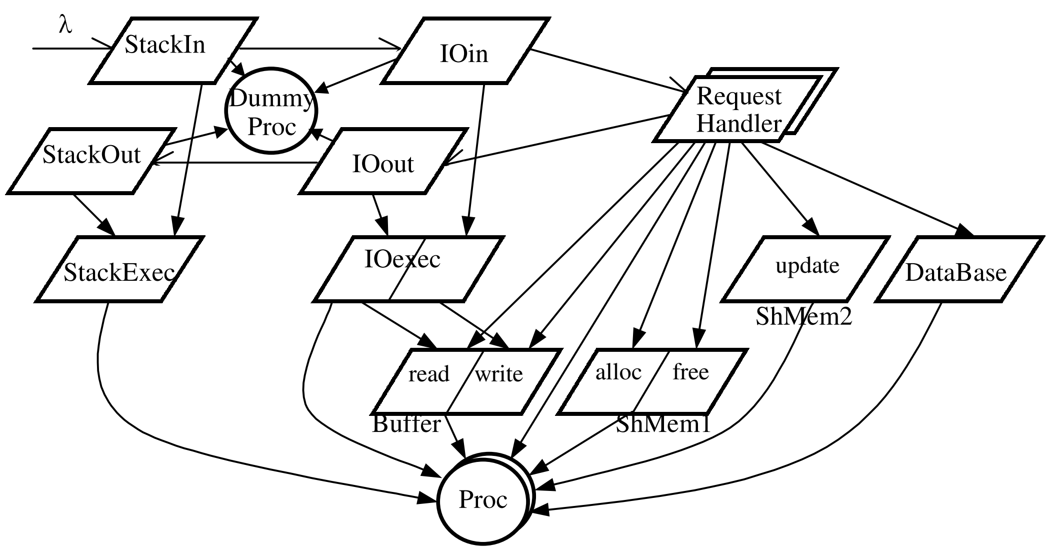


Figure 11. LQN base case model of the telecommunication system

applying systematically the transformation rules described in the previous section to the architectural patterns/collaborations used in the system, as shown in Fig. 10, the LQN model shown in Fig.11 was obtained.

The next step was to determine the LQN model parameters (average service time for each entry, and average number of visits for each request arc) and to validate the model. We have made use of measurements using Quantify [19] and the Unix utility *top*. The measurements with Quantify were obtained at very low arrival rates of around a couple of requests/second. Quantify is a profiling tool which uses data from the compiler and run-time information to determine the user and kernel execution times for test cases chosen by the user. Since we wanted to measure average execution times for different software components on behalf of a system request (see Table 1 in the Appendix), we have measured the execution of 2000 requests repeated in a loop, then computed the average per request. Although we have not computed confidence intervals on the measurements, repeated experiments were in close agreement. The *top* utility provided us with utilization figures for very high loads of hundreds of requests/second, close to the actual

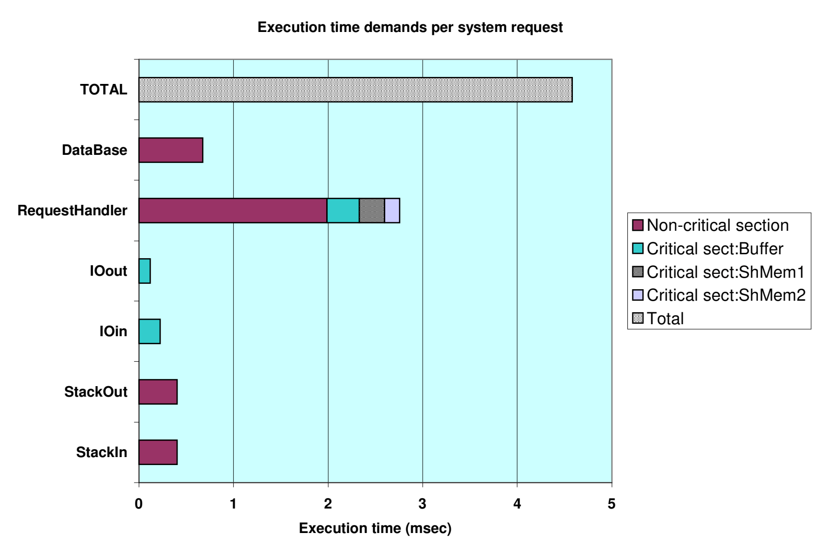


Figure 12. Distribution of the total demand for CPU time per request over different software components

operating point. These measurements were done on a prototype in the lab for two different hardware configurations, with one and four processors. Again, repeated measurements were in close agreement. We have used the execution times measured with *Quantify* (given in Table 1 in the Appendix) to determine the model parameters, and the utilization results from *top* to validate our model. The utilization values obtained by solving the model were within 5% of the measured values. Unfortunately, a more rigorous validation was hindered by the lack of response time measurements.

6. Performance Analysis of the Telecommunication System

Although the LQN toolset [4] offers both analytical and simulation solvers, the model results used in this section were obtained by simulation. The reason is that one of the system features, namely the scheduling policy by polling used for the RequestHandler multi-server, could not be handled by the analytical solver. All the simulation results were obtained with a confidence interval of plus/minus1% at the 95% level.

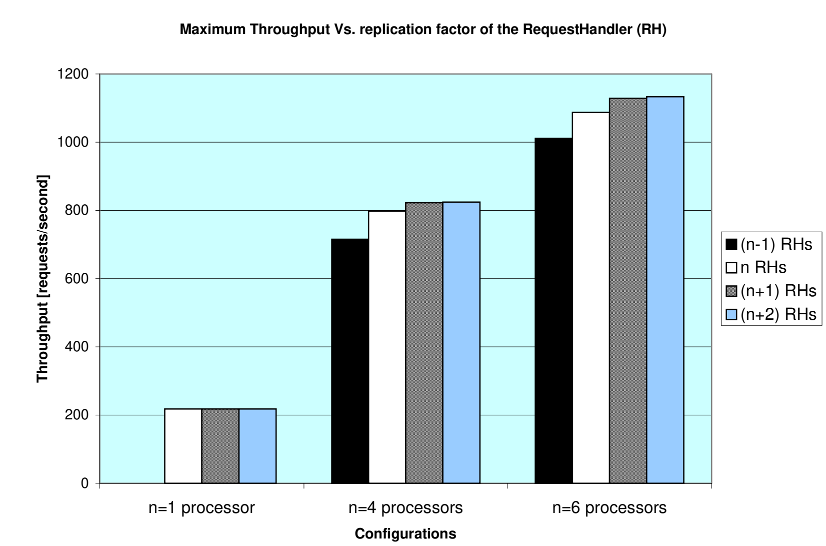


Figure 13. Maximum achievable throughput for different hardware and software configurations and a single class of service requests

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. 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 (as in Fig. 13). Then the configurations with a maximum throughput 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. Solving the LQN model is a more efficient way to explore different configurations under a wide range of workloads then to measure the real system for all these cases.

Although we have modelled the system for two classes of services, we selected to report here only results for a single class because they illustrate more clearly how the bottleneck moves around from hardware to software for different configurations. The model was analyzed 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 Fig. 14), 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 [8], 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 said to be “utilized” when is doing effective work and when is waiting to be served by lower level servers (including the queueing for the included services). Fig. 17 and 18 show different contributions to the utilization of two tasks, IOout and RH, when the system is saturated (i.e., works at the highest achievable throughput given in Fig. 13) for different numbers of RH replications. It is easy to see that for more than five RH copies, one task (i.e., IOout) has a very high utilization even though it does little useful work, whereas at the same time another task (i.e., a RH copy) has a lower utilization and does more useful work..

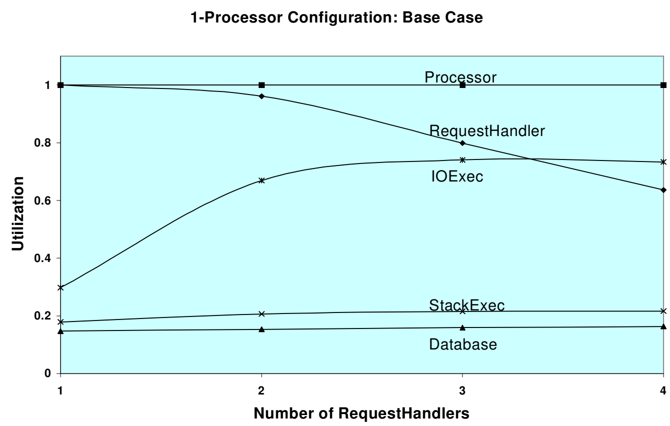


Figure 14. Task utilizations for 1-proc base case

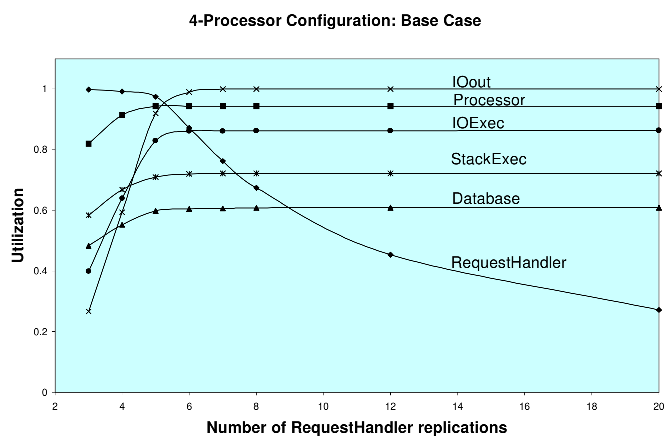


Figure 15. Task Utilizations for 4-proc base case

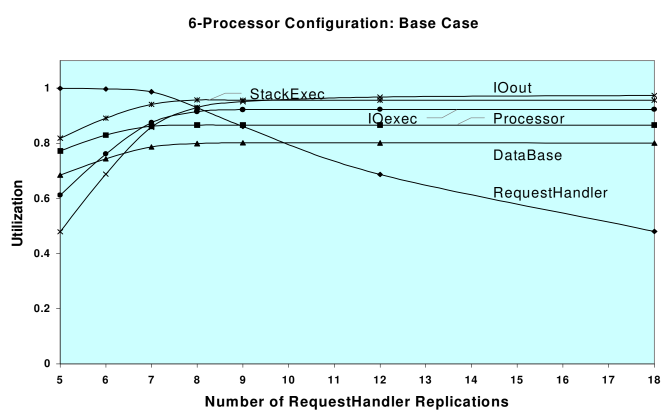


Figure 16. Task utilizations for 6-proc base case

Interestingly enough, in the case of 4-processor configuration we notice that with more processing capacity in the system, the processors do not reach the maximum utilization level, as shown in Fig.15. Instead, two software tasks IOout and IOin (which are actually responsible for little useful work on behalf of a system request) reach critical levels of utilization due to serialization constraints in the software architecture. There are two reasons for serialization: i) IOin and IOout are executed by a single thread of control (which in LQN means waiting for task IOexec), and ii) contend for the same buffer. Thus, with increasing processing power, the system bottleneck is moving from hardware to software. This trend is more visible in the case of 6-processors configuration, where the processor utilization reaches only 86.6%, as shown in Fig.16. The bottleneck has definitely shifted from hardware to software, where the limitations in performance are due to constraints in the software architecture.

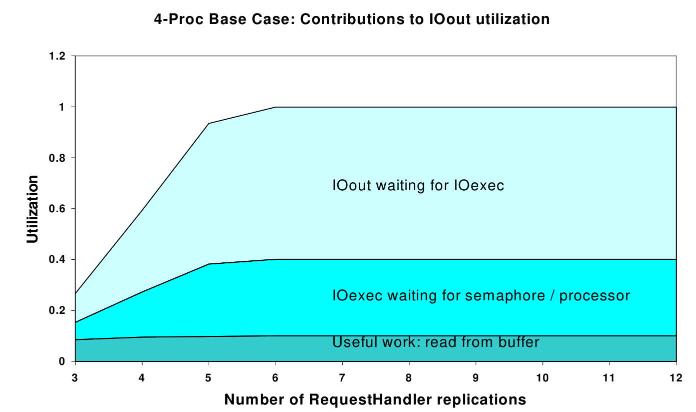


Figure 17. Contributions to IOout utilization when the system is saturated, in function of the RH replication level

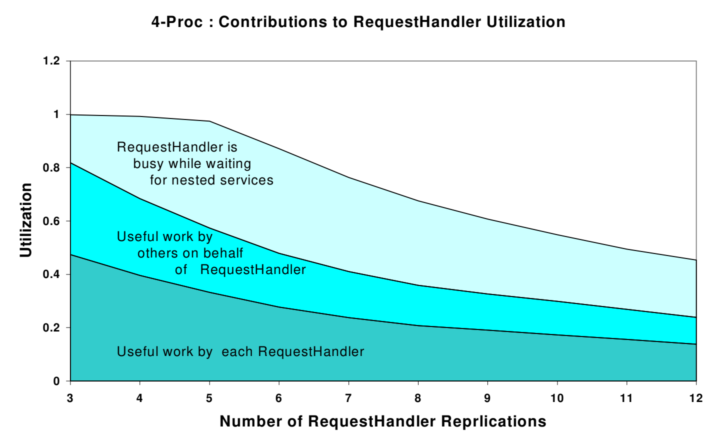


Figure 18. Contributions to the utilization of a RH copy when the system is saturated, in function of the RH replication level

We tried to eliminate the serialization constraints in two steps: first by making each filter a process on its own (i.e., by removing StackExec and IOexec tasks in the LQN model), then by splitting the pipeline buffer sitting between IO process and RequestHandler in two buffers. The LQN model obtained after the 2-step modifications is shown in Fig. 19. The results of the "half-way" modified system (after the first step only) are given in Fig. 20, and show no major performance improvement (the processor is still used below capacity). By examining again the utilization components of IOin and IOout (which are still the bottleneck) we found that they

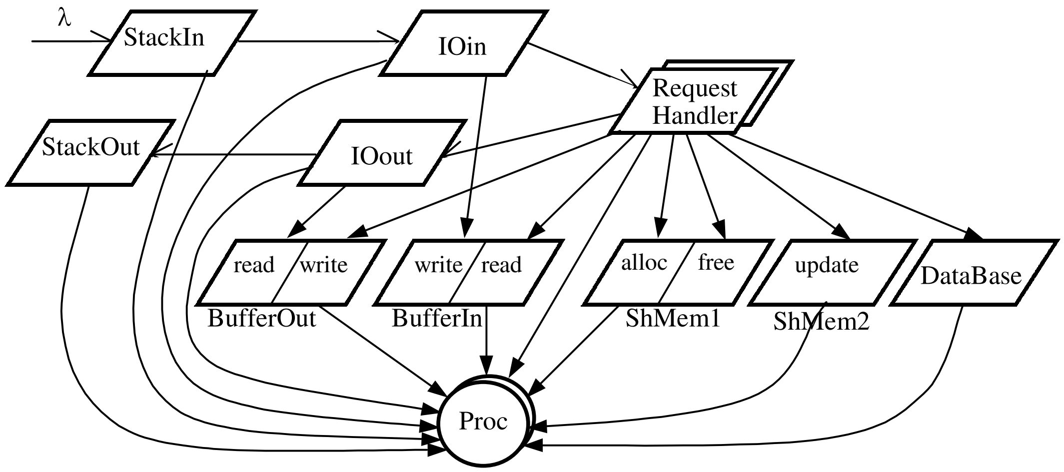


Figure 19. LQN model of the modified system

wait most of the time to gain access to the Buffer (IOout is waiting about 90% of the time and IOin 80%). After applying both modification steps, though, the software bottleneck due to excessive serialization in the pipeline was removed, and the processor utilization went up again, as shown in Fig. 21 for the 4-processor and in Fig. 22 for the 6-processor configuration.

As expected, the maximum achievable throughput increased as well. The throughput increase was rather small in the case of 4 processors (only 2.7%), and larger in the case of 6 processors (of 10.3%), where there was more unused processing power. We also realized that in the case of 6 processors a new software bottleneck has emerged, namely the Database process (which is now 100% utilized). 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 to offer is the ability to explore a range of design alternatives, configurations and workload mixes, and to detect the causes of performance limitations just by analyzing the model, before proceeding to change the real system.

7. Conclusions

This paper contributes toward bridging the gap between software architecture and performance analysis. It proposes a systematic approach to building performance models from the high-level software architecture of a system, by transforming each architectural pattern employed in the system into a performance sub-model. There is on-going work to formalize the kind of transformations presented in the paper from the architecture to the performance domain by using formal graph transformations based on the graph-grammar formalism [9].

The paper illustrates the proposed approach to building LQN models by applying it to an existing telecommunication system. The performance analysis exposes weaknesses in the original

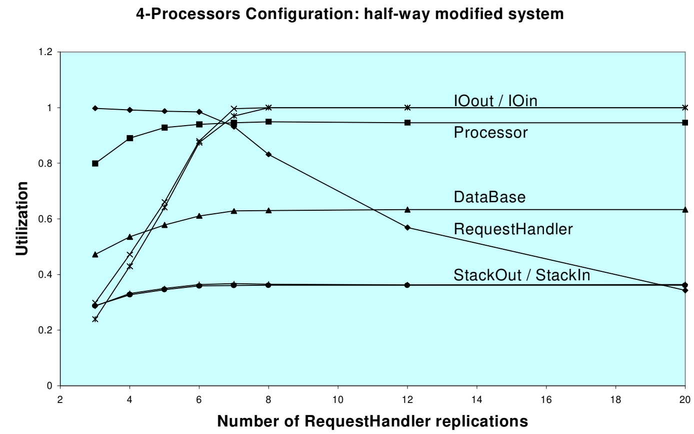


Figure 20. Task utilizations for the 4-proc half-way modified system

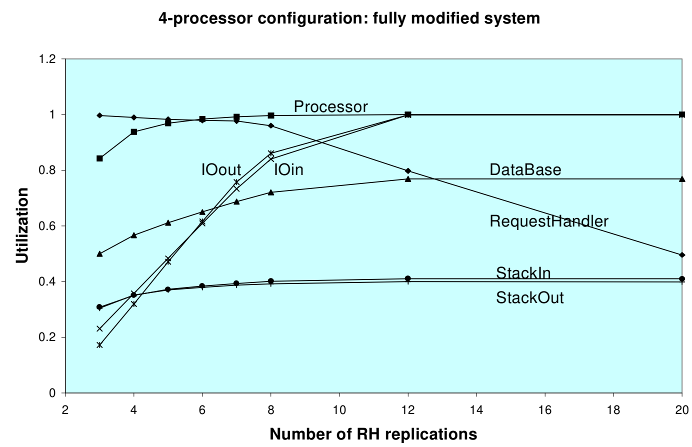


Figure 21. Task utilizations for the 4-proc fully modified system

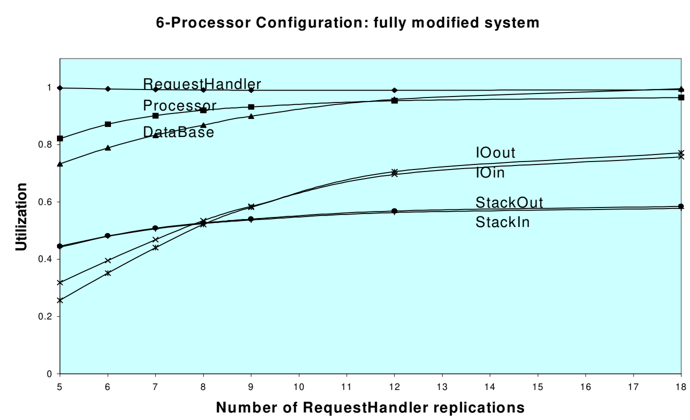


Figure 22. Task utilization for the 6-proc fully modified system

architecture due to excessive serialization, which show up when more processing power is added to the system. Surprisingly, software components that do relatively little work on behalf of a system request can become the bottleneck in certain cases, whereas components that do most of the work do not. After removing the serialization constraints, a new software bottleneck emerges, which leads to the conclusion that the software architecture as it is does not scale up well. This case study illustrates the usefulness of applying performance modelling and analysis to software architectures.

References

[1] R.Allen, D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering Methodology*, Vol.6, No.3, pp 213-249, July 1997.

[2] G.Booch, J.Rumbaugh, I.Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[3] F. Buchmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley Computer Publishing, 1996.

[4] G. Franks, A. Hubbard, S. .Majumdar, D. Petriu, J. Rolia, C.M. Woodside, "A toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation*, Vol. 24, Nb. 1-2, pp 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] G. Franks, C.M.Woodside, "Performance of Multi-Level Client-Server Systems with Parallel Service Operations", *Proceedings of the First International Workshop on Software and Performance*, Santa Fe, USA, pp.120-130, Oct. 1998.

[7] 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.

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

[9] D. Petriu, X.Wang, "Deriving Software Performance Models from Architectural Patterns by Graph Transformations", *The Sixth International Workshop on Theory and Applications of Graph Transformations* TAGT'98, pp.340-347, Paderborn, Germany, Nov. 1998 (to appear in a special volume of *Lecture Notes of Computer Science*, Springer Verlag).

[10] S.Ramesh, H.G.Perros, "A Multi-Layer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages*", Proceedings of the First International Workshop on Software and Performance*, Santa Fe, USA, pp.107-119, Oct. 1998.

[11] J.A. Rolia, K.C. Sevcik, "The Method of Layers", *IEEE Trans. On Software Engineering*, Vol. 21, Nb. 8, pp 689-700, August 1995.

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

[13] M. Shaw, "Some Patterns for Software Architecture" in *Pattern Languages of Program Design 2* (J.Vlissides, J. Coplien, and N. Kerth eds.), pp.255-269, Addison Wesley, 1996.

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

[15] B.Spitznagel, D.Garlan, "Architecture-Based Performance Analysis", Proc. of the Int. Conference on Software Eng. and Knowledge Eng. SEKE'98, pp. 146-151, 1998.

[16] L.G Williams, C.U.Smith, “Performance Evaluation of Software Architectures”, *Proceedings of the First International Workshop on Software and Performance*, Santa Fe, USA, pp.164-177, Oct. 1998.

[17] C.M. Woodside. "Throughput Calculation for Basic Stochastic Rendezvous Networks". Performance Evaluation, vol.9(2), pp. 143-160, April 1988.

[18] 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, Vol.44, Nb.1, pp 20-34, January 1995.

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

Appendix

Table 1. Average execution time measurements for different software components

Component	*Average execution time per visit*	*Visit count per request*
Request handler (non-critical section)	1.988 ms	1
DataBase	0.6749 ms	1
StackIn	0.4028 ms	1
StackOut	0.4028 ms	1
IOin (includes Buffer_write)	0.121 ms	1
IOout (includes Buffer_read)	0.226 ms	1
Buffer_read (includes semaphore acquire/release)	0.120 ms	2
Buffer_write (includes semaphore acquire/release and item allocation)	0.225 ms	2
ShMem1_alloc (includes semaphore acquire/release)	0.1328 ms	1
ShMem1_free (includes semaphore acquire/release)	0.1328 ms	1
ShMem2_update (includes semaphore acquire/release)	0.1576 ms	1