

A Reflective QoS-enabled Load Management Framework for Component-Based Middleware

Octavian Ciuhandu, John Murphy
Performance Engineering Laboratory
Dublin City University
+353 1 700 7644

{ciuhandu,murphy}@eeng.dcu.ie

ABSTRACT

A new reflective QoS-enabled load management framework for component oriented middleware is presented. The proposed framework offers the possibility of automatically selecting the optimal load distribution algorithms and changing the used load metrics at runtime, according to workload time evolution. QoS service level agreements are being offered at application level, transparent to the managed application.

Categories

C.4 [Performance of Systems]; D.2.8 [Software Engineering]: Metrics --- Complexity measures, performance measures; D.2.12 [Software Engineering]: Interoperability --- Distributed objects.

Keywords

Load, distribution, QoS, platform, middleware, optimization, adaptation, management, component.

General Terms

Algorithms, Management, Measurement, Performance, Reliability, Standardization

1. INTRODUCTION

The increasing demand for distributed applications and increasing load on existing distributed systems leads to a growing need for improved throughput and scalability. Many types of applications rely heavily on predictable computing and networking services for performing their jobs in a timely manner.

In the context of this paper we will refer to Quality of Service (QoS) as being the set of those quantitative and qualitative characteristics of a distributed application which are necessary in order to achieve the required functionality. In this context, the discriminating parameter for QoS is client response time. Based on this parameter, different service levels can be defined (e.g. premium – guaranteed maximum response time, standard – guaranteed response time and best-effort – no guarantees).

Distributed applications require a broad range of features, such as service guarantees and adaptive resource management for supporting a wider range of QoS aspects, such as dependability, predictable performance, secure operation and fault tolerance [1]. As a result, the focus is shifting from programming to integration.

Component and application QoS proves to be insufficient, end-to-end (i.e. client to server) QoS support being needed.

2. MOTIVATION

An architecture that integrates and coordinates the existing QoS technologies is needed for providing end-to-end QoS. Integration and coordination must take place across all system resources, at all system levels and on all time scales of system development, deployment and operation.

Although decisions for managing QoS are being made throughout applications' life-cycle (i.e. design time, configuration / deployment time and runtime), the runtime requirements are the most challenging ones due to the timely manner in which decisions need to be taken.

Most distributed systems are increasingly required to use commercial off-the-shelf (COTS) components. The newly available component technologies allow clients to invoke operations ignoring details such as component location, programming language, operating system platform, communication protocols or interconnects and hardware. Nevertheless, the lack of support in these components for QoS specification and enforcement features, technology, performance, predictability and scalability optimizations has resulted in a very limited development rate for advanced distributed applications [2]. This also causes an increase in complexity for the load management problem.

Adaptive middleware offers the possibility of modifying an applications' functional and QoS-related properties dynamically as well as statically. Static configuration is used for leveraging specific platforms' capabilities and enabling functional subsetting and minimizing hardware/software infrastructure dependencies. Dynamic reconfiguration offers the possibility of adapting system configuration and optimizing its responses to changing environments or requirements.

Reflective middleware extends the concepts of adaptive middleware by permitting automated examination and adjustment of the offered QoS capabilities.

3. PROPOSED APPROACH

A new reflective QoS enabled load management service for component-based middleware is proposed. It consists of a load monitoring module, a load prediction module and a QoS control module. They all feed information to a load evaluator module, which selects the optimal load distribution policy, and a load distribution module that uses the selected policy for request distribution. The architecture of the framework is presented in Figure 1.

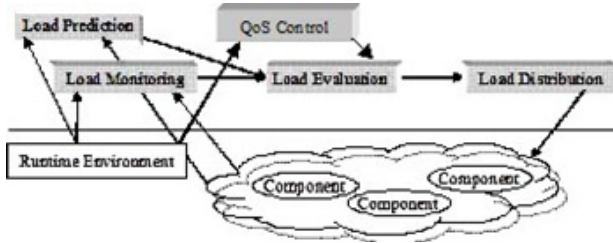


Figure 1. Framework architecture

The framework modules are presented in the following sections (3.1 to 3.5). All modules have well-defined interfaces. The modules are independent and according to distributed applications' complexity the optimal load management service configuration can be selected.

3.1 Load Monitoring Module

For simple systems, observing only a few basic load metrics, (e.g. CPU, memory, network usage) is sufficient for making good load distribution decisions. Nevertheless, for complex systems, the performance can be influenced by resource contention and other complex factors, thus component instances, have to be monitored in order to select optimal load distribution.

Every load monitor requires system resources, thus instantiating a large number of monitors is not a valid approach. In addition, load distribution decisions taken for a group of component instances can severely influence the performance of other groups. The optimal set of monitors has to be identified.

Our proposed solution employs pluggable load monitoring modules (e.g. CPU, memory, response time monitors, component instances monitors) that can be activated, deactivated or swapped at runtime, according to system requirements.

Different performance monitoring and prediction frameworks are being developed [3], [5], [6], [7], [8]. Parts of these frameworks could be used as plug-ins for our load management service. In [3], a complex framework for system monitoring, modeling and prediction is described. The monitoring part uses component proxies that can be enabled or disabled at runtime, according to the requirements. These monitors could also be shared by the proposed framework for detecting existing bottlenecks.

3.2 Load Prediction Module

For some distributed systems, workload time evolution is fairly deterministic, thus predictions for its evolution in the near future can be easily made. For other systems, creating such a model is a difficult if not impossible task.

A load prediction module that uses the information it receives from the runtime environment and from the load monitoring modules is being proposed. Information about time evolution of client requests is also gathered. These represent the entries for a modeling algorithm that creates a model, based on which predictions are being made.

The model is continuously updated and validated. The module can be deactivated if not required (e.g. for simple applications) or if the workload evolves in such a way that a reliable model can not be created.

3.3 QoS Control Module

The implementation of QoS as a part of the load management system is a key design issue (being totally transparent to the load-managed distributed application). The QoS service levels are at

application level, i.e. different users accessing the distributed application can avail of different service levels but a user can not avail of different QoS service levels for different operations. The latter option can be introduced when complex frameworks for application monitoring and modeling [9] become available and can be incorporated in the service, so that application architecture is known, but this is beyond the scope of this paper.

The QoS levels and their performance guarantees are specified in a standardized format at deployment time and can be changed during runtime. This is an important requirement since it offers the possibility of changing/adding QoS guarantees without the need of changing any code in the distributed application.

Based on the specified QoS levels, the QoS module establishes the end-to-end QoS requirements (for the required service level for that particular user) for every newly created connection and continuously monitors the application response times.

If end-to-end QoS requirements are not available for a connection (due to the infrastructure not fully supporting QoS) local QoS requirements are enforced and the connection is continuously monitored. If increased communication delays are detected the algorithm adapts the enforced QoS policies trying to maintain the service level agreement. While this cannot offer full hard end-to-end guarantees, it offers the best solution for this situation.

3.4 Load Evaluator Module

Most load distribution algorithms are designed targeting specific load conditions, for which they realize an optimal distribution. A load distribution algorithm might not be able to handle degenerated load conditions (unstable applications) and the use of specifically designed algorithms for restoring system equilibrium is required [4]. Thus, a key requirement for any load management system is the possibility of changing the load distribution algorithm. The selection is made dynamically, at runtime, for ensuring high availability. For automatic load distribution algorithm selection, it is required that all algorithms include a standard description of the workload type for which they are most suitable. Since complex load distribution algorithms can have multiple tuning parameters, for the moment the algorithm is considered to include a specified set of values for these parameters. Thus, the same algorithm with different tuning parameters values is considered to be a separate algorithm. As a further extension to the framework, the possibility of reflective parameter tuning can be investigated but this is beyond the scope of this paper.

In order to ensure that the modules of the load management system are exchangeable, the particular combination of metrics used for load monitoring should be completely unimportant, only the magnitude of the metric being considered while optimizing the load distribution policy should be considered. This is a critical requirement for allowing dynamic load distribution algorithms and load monitoring modules runtime replacement.

Based on predictions from the load prediction module, the system can preemptively adjust its configuration, minimizing the response time and optimizing the load distribution and system response for high-priority service levels.

Servers can be activated/deactivated, according to the information received from the load monitoring and load prediction modules (especially for farms of servers hosting a number of distributed applications, as presented in section 3.6)

3.5 Load Distribution Module

The policy selected by the load evaluator module is used by the load distribution module for forwarding the incoming requests to the servers.

The most important feature of this module is failover protection. A good load distribution module should always be available and should be able to distribute the incoming requests even if other modules of the load distribution service fail. The solution proposed is the inclusion of a simple load distribution algorithm, like round robin, in the load distribution module. If the load evaluator module fails due to some unknown errors, the load distribution module reverts to the simple algorithm, ensuring that the system keeps on running albeit with degraded performance.

The load distribution module has to take into account all levels of load management, namely initial placement, migration and replication.

Initial placement represents the creation of new component instances on replicas where sufficient resources are available for efficient execution. Note that this does not refer to placement of component instances at application deployment time; rather it refers to placement of component instances during runtime when the load management system determines that extra component instances are required.

Migration of running instances deals with the movement of existing component instances to another replica that offers the required resources for more efficient execution. In the case of stateful components the state must be continuously synchronized.

Replication of component instances involves the creation of new component instances from an existing one. The new instances must be identical with the source. This applies only to stateful component instances, for stateless ones only simple instantiation being required. A complex problem in this case is replicating the state of the original component instance to the newly instantiated ones. The state must also be continuously kept synchronized among all existing replicas.

3.6 Framework Characteristics

The load management service has an instance active on every server. Load distribution decisions can be taken in a collaborative way, all instances of the load management service participating in this process, or using an elected coordinator (the other servers periodically reporting to it and checking its state). The latter, while reducing the load and network overhead as well as distribution algorithm complexity, does not guarantee availability. Any coordinator failure will result in requests being dropped until the other replicas detect the failure and select a new coordinator. Both options are available and according to application requirements the most suitable one is selected at configuration time.

Key features of the proposed load management service are that it provides application-level QoS, its modules can be activated / deactivated / exchanged at runtime and the service can be extended by adding new load distribution algorithms and new monitors at runtime. Any framework module can be upgraded at runtime when new versions become available (e.g. new prediction methods).

Farms of servers hosting multiple distributed applications have been considered as a possible target environment thus the possibility of activating/deactivating replicas (i.e. servers on which a copy of the distributed application is active) has been included.

The framework offers the possibility of being interconnected with similar services. The resulting group of services, each managing the workload of one or more distributed applications, can share the available resources for maximizing the performance of all distributed applications. The group can temporarily transfer the control of a (group of) replica(s) from a low loaded system to an overloaded system.

4. RESULTS

For evaluating the performance of different distribution algorithms, a simulation model was created using Hypermix Workbench (Figure 2). This tool has been selected because it is a general-purpose modeling and simulation tool, it offers a graphical programming language for performance modeling and it is a powerful and easy tool to use to construct simulation models. It is suited for prototyping design issues for performance because it offers early possibilities for exploring the solution space.

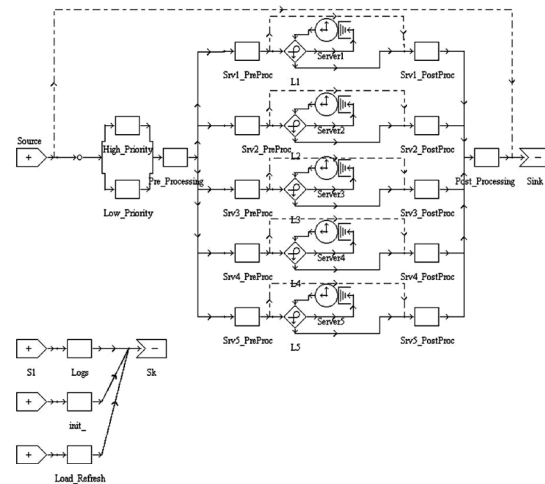


Figure 2. Simulation model

The system we have chosen to model consists of five servers, each of them with equal processing power, and a transaction generator. Every transaction that enters the system has a certain load associated with it (i.e. a number), which will represent the workload on the designated server. For any application server, the service time is dependent on the number of transactions processed simultaneously and on the total workload (i.e. the sum of the load of all transactions) on that application server.

Our model consists of five service nodes (Server1...Server5) that represent the available servers, each of them with equal processing power and with a service time that dynamically updates, according to the total load on the server (each time a transaction enters or leaves the server).

The Source Node generates transactions in bursts, during every time unit. The number of transactions generated is a random variable with a Poisson distribution function. From the Source Node, the transactions are sent, according to the selected probability (5% / 10% / 15%) to the High_Priority node, the rest being sent to the Low_Priority node, where transactions are labeled and the destination server is selected.

The Pre_Processing and Post_Processing nodes (and the Srv_i_PreProc and Srv_i_Post_Proc nodes) are used to compute results. In the Pre_Processing node a certain workload is associated with every transaction, according to a Lognormal distribution.

The Poisson and Lognormal distributions were selected because they are the most frequent distributions used in the literature for modeling the arrival rate / load distribution.

In the first setup, the model had no knowledge of priorities and was used to evaluate the transaction response times for different algorithms. The algorithms implemented for the first step were Round-Robin, Weighted Round-Robin (a weight of 2 was assigned to the first server) and Least-Loaded. These algorithms were chosen because they are some of the mostly widely used algorithms in this field, are fairly simple to implement and do not add important delays in the system. The results are presented in Table 1, showing the Time Units (TU) for each algorithm.

Algorithm	Response Time [TU]
Round Robin	1.02 TU
Weighted Round-Robin	0.712 TU
Least-Loaded	0.665 TU

Table 1. Performance of load distribution algorithms

The Least-Loaded algorithm produced average response times representing 65.2% of the average response times produced by the Round-Robin algorithm.

In the second setup, the best performing algorithm from the first setup was chosen and the concept of priorities was introduced: all transactions were flagged, as either low or high priority, and the chosen algorithm was adapted to take such information into account. Two situations were considered: in the first case, one server was serving only high priority transactions; in the second case, the server processing high priority transactions was also used to serve some low priority transactions, in order to reduce the load on the other machines.

The case	High Priority Percentage	Priority	Response Time [TU]
Separate Server	5%	High	0.274
		Low	2.292
	10%	High	0.347
		Low	1.623
	15%	High	0.425
		Low	1.325
Weighted Least Loaded	5%	High	0.456
		Low	0.9
	10%	High	0.506
		Low	0.875
	15%	High	0.576
		Low	0.846

Table 2. Shared Server Improvements

From Table 2, it can be observed that, with 5% high-priority transactions, an increase of 0.182 Time Units (TU) in the average response time for the high priority transactions and a decrease of 1.392 TU in the average response time for the low-priority transactions is obtained. With 15% high-priority transactions, an increase of 0.151 TU in the average response time for the high priority transactions and a decrease of only 0.479 TU in the average response time for the low-priority transactions is observed.

5. CONCLUSIONS

A framework for a modular QoS-enabled load management system was presented. According to the complexity of the managed application, only some of the modules included in this

framework might be required and thus activated. The framework offers the possibility of optimally selecting the desired point of tradeoff between distribution overhead and performance improvements.

The load management service offers the possibility of connecting to other similar services. The group of services, each managing the workload of a distributed application, can cooperate and share available resources in order to maximize the performance of all distributed applications. The group can maintain a shared list of available servers and can temporarily transfer the control of a (group of) server(s) from a low loaded system to a high loaded system.

The results have shown that, as expected, algorithms that have knowledge about the system have better performance (34.8% improvement for response time). If the assumption of uniform transaction workload distribution is not valid, the improvements are expected to be even higher.

When priorities were introduced, it was shown that small increases in the response time for high priority transactions (that can be eliminated if the weights assigned to the servers are tuned), lead to substantial improvements in the response times for low priority transactions.

6. ACKNOWLEDGMENTS

The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2002.

7. REFERENCES

- [1] R. Schantz and D. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications", The Encyclopedia of Software Engineering, J. Wiley & Sons. December 2001, pp. 801-813.
- [2] V. Kachroo, Y. Krishnamurthy, F. Kuhns, R. Akers, P. Avasthi, S. Kumar, and V. Narayanan. "Design and Implementation of QoS Enabled OO Middleware", Internet2 QoS Workshop, February 2000.
- [3] A. Mos and J. Murphy, "Understanding Performance Issues in Component-Oriented Distributed Applications: The COMPAS Framework", WCOP 2002, at ECOOP, Malaga, Spain, 2002.
- [4] C.-C. Hui and S. T. Chanson, "Improved Strategies for Dynamic Load Balancing", IEEE Concurrency, vol.7, July 1999.
- [5] F. Lange, R. Kroeger and M. Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System", IEEE Transactions on Parallel and Distributed Systems, Nov. 1992.
- [6] B. Sridharan, S. Mundkur, A. P. Mathur, "Non-intrusive Testing, Monitoring and Control of Distributed CORBA Objects", TOOLS Europe 2000.
- [7] R. Weinreich and W. Kurschl, "Dynamic Analysis of Distributed Object-Oriented Applications", Proc. Hawaii International Conference on System Sciences, Hawaii, Jan. 1997
- [8] Precise, Precise/Indepth for the J2EE platform, www.precise.com/Products/Indepth/J2EE/Papers/
- [9] A. Mos and J. Murphy, "Performance Management in Component-Oriented Systems using a Model Driven Architecture Approach", Proc. of 6th IEEE International Enterprise Distributed Object Computing (EDOC) Sep. 2002, EPFL, Switzerland