2002

QOS Monitoring in Middleware

Peter Bodorik
Dalhousie University, bodorik@cs.dal.ca

Shawn Best
IBM Canada Ltd., sbest0@ca.ibm.com

Dawn N. Jutla
St. Mary's University, Canada, dawn.jutla@stmarys.ca

QOS MONITORING IN MIDDLEWARE

Peter Bodorik

Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada
Phone: 902.494.6452; Fax: 902.492.1517
bodorik@cs.dal.ca

Shawn Best

E-Access Solutions, IBM Canada Ltd., Markham, Ontario, Canada
Phone: 905.316.009
sbest0@ca.ibm.com

Dawn Jutla

Faculty of Commerce, Saint Mary's University, Halifax, Nova Scotia, Canada
Phone: 902.420.5157; Fax: 902.496.8101
dawn.jutla@stmarys.ca

ABSTRACT

Monitoring the system and services is unavoidable if quality of service (QoS) is to be provided. This paper describes monitoring of middleware components, which express the business logic of e-business applications, and the environment in which they execute. Middleware components have become important because of their prominent role in the EAI (Enterprise Application Integration) and implementation of n-tier web applications. We have developed a framework and a supporting toolkit that enables a QoS specialist/engineer to facilitate monitoring and reporting. Two types of monitored data are collected. The QoS engineer identifies critical activities withing a component so that their delays can be captured. Also collected are data from probing sub-systems forming the environment in which the components execute. We discuss in detail our approach to instrumentation of monitoring probes. Probes habe been successfully instrumented for a number of (sub)systems that form the environment in which the componenets execute, systems that include operating, network, and DB systems.

1. INTRODUCTION

That QoS in software is important is supported by the wide attention given to QoS in many applications of software ranging from those dealing with multimedia, such as video-on-demand, distance education, and telemedicine, to traditional data processing applications. QoS refers to the non-functional requirements of the systems, such as the number of frames transferred per unit of time, availability, and the transaction throughput. Some applications have *hard* QoS requirements that the system must satisfy, while others have *soft* requirements in that a system is still considered to be functional even though it does not meet the negotiated level of QoS [Schantz 2000].

Provisioning of QoS has been facilitated thus far using two general approaches. In one, as part of the development process when the application is created, the developer includes provisions for QoS by

monitoring appropriate parameters and explicitly managing resources internal to the application. In other words, the QoS is considered to be one of the dimensions of the application software development effort. This approach is taken in, for instance, [Katchabaw 1999] and [Ye 1999]. It also appears in current application servers, such as Visibroker or Orbix, however, as part of infrastructure. Each server includes a certain (minimum) number of threads. As the number of client requests rises, the number of threads that are serving client requests is increased. Sharing of DB connections by various servers providing access to DBs is another example of this approach.

Another approach is to perform monitoring of subsystems for the purposes of managing resources allocated to them. This is widely utilized in managing network infrastructure and also in managing various servers, such as web or application servers. The load on the servers is monitored and if load approaches critical levels, appropriate actions are taken. For instance, if an application (or a web) server load is reaching critical levels it may be replicated and the load distributed. Many companies, such as IBM/Tivoli HP with OpenView, have adopted this approach. Relatively recently these companies have been interested in QoS from the user-perspective and have developed or are developing products for measuring the QoS from the perspective of the end-user in a manner as pioneered by Keynote Systems.

QoS is provided in an N-tier environment in the following way [Schantz 2000, Pal 2000, Le Tien 1999, van Morsel 1999]. First, the client and the server negotiate an expected level of QoS. Once an agreement is reached and the service is being provided, it is monitored. If the negotiated level of QoS cannot be provided, then it is renegotiated. The client and the server systems can thus respond to the changes in the environment that affect the service. Of course, the QoS and various system parameters must be monitored and appropriate data must be collected. This is used not only to determine whether the QoS agreements are satisfied, but also for allocation of resources so that appropriate QoS can be provided to the right clients.

The above paragraph alludes to a sleuth of problems that need to be addressed in order to provide effective QoS and some of these problems are more important to some applications than to others. We shall just note that monitoring services underpins all of the QoS mechanisms. Because of possible failures, this is the case even in systems in which resources are pre-allocated. It is the monitoring aspect of QoS that we are interested here. Data collected by monitoring is forwarded to some agency that uses it to manage resources and/or to renegotiate current contracts and negotiate new ones.

This paper concentrates on monitoring aspects of (soft) QoS provisioning in the environment of middleware, particularly business middleware that access various data sources. We developed a framework for this approach and then validated it on a Java platform, which was chosen because it is flexible and portable, by developing a toolkit to facilitate monitoring. Middleware components operate in a heterogeneous environment supported by various hardware/software platforms that communicate over networks. It is thus important that gathering of environmental parameters be extensible. We used the toolkit successfully to probe/monitor a large number of subsystems that comprise the environment in which the middleware components execute. We concentrate here on describing how we achieved the ease and generality of probing the various subsystems. The paper is organized as follows. Section two provides general assumptions, architecture and the toolkit. The next two sections describe creation and instrumentation of probes. The final sections contain related work, and summary and conclusions.

2. ASSUMPTIONS, ARCHITECTURE, AND TOOLKIT

We briefly overview context material that can be found in [Bodorik 2002], which describes a framework and a toolkit, and also in [Best 2001], which provides details on implementation and design decisions.

2.1 Architecture and Assumptions

A simple architecture, which has been utilized, amongst others, in [Schantz 2000, Pal 2000, vanMorsel 1999], is shown in Figure 1. Services, provided to clients by server components, are monitored internally while the environment in which the server components execute is also monitored. Data reports gathered by monitoring are sent to some c*entral agency (CA)* that negotiates/renegotiates QoS contracts between the clients and servers and manages resources to ensure provision of appropriate levels of services to the right clients. In [Schantz 2000, Pal 2000] the CA is called a kernel, while in [vanMorsel 1999], the monitoring reports are recorded in a *database (DB)*. It is assumed that a component is executed by a server that has a number of threads that actually provide services requested by clients. The server contains data structures used to collect data gathered by the monitoring code. The figure also shows a controlling agent that communicates with and receives instructions from CA and sets controlling variables according to these instructions. The variables control the behaviour of the reporting and probing agents. A probing agent periodically invokes methods that probe the state of the system. The frequency is governed by the control variables.

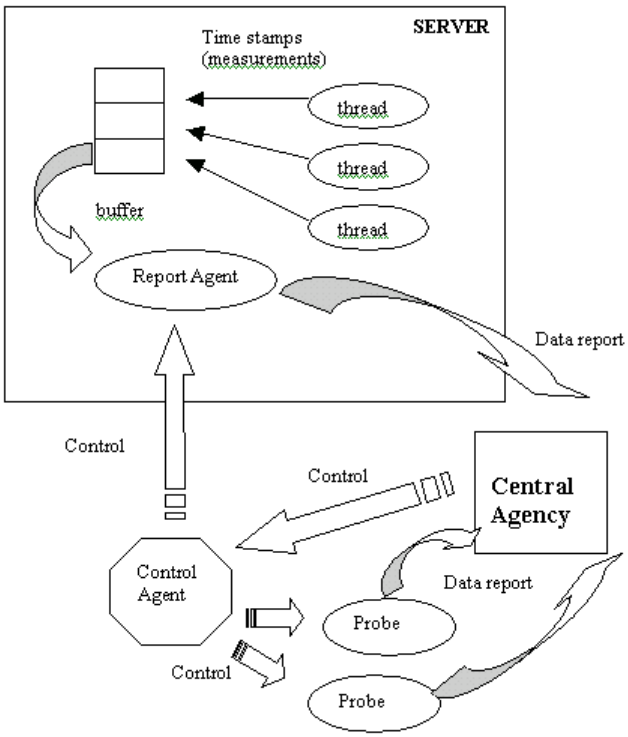

Figure 1: Architecture to Monitor Server Process

Since here we are concentrating monitoring and collecting of obserations, how the collected data is used in terms of negotiation of contracts and management of resources is outside the scope of this paper. We simply assert that monitoring of services is required and the data that we collect is needed. The CA may be a part of the server itself. For instance, if it is within the server, then it, the CA, may determine the allocation of resources within the server. If reporting is to some CA outside the server, then decisions may be made on the replication and/or migration of the server.

2.2 Toolkit for QoS Monitoring

The toolkit is based architecture outlined above and the two-staged approach to QoS provisioning. First, the server code/components are developed and tested using the usual methods and tools. Following this, the QoS engineer uses the toolkit to extend the server software to instrument

appropriate monitoring and reporting. It may be that the developer herself performs the role of the QoS engineer. This two-stage approach has been tried, amongst others, in [Schantz 2000, van Morsel 1999, Hafid 1996] for QoS provisioning and in [Jutla 2001] for automating the development of services provided by distributed objects.

Components are now utilized as a standard approach when developing services and applications. Not only do the components support re-use of software but also allow the application developer to concentrate on the business logic while relieving him/her responsibility for dealing with issues that can be provided by infrastructure components. For instance, when a service, being developed for an application, requires access to a number of data sources while preserving transactional properties, without the support by infrastructure components, the developer has to address various issues dealing with allocation of resources, access to different data sources, and preservation of transactional properties. With the support of appropriate infrastructure based on components, the business developer is relieved of these tasks. The components that implement the business logic are placed in containers, provided by the development platform, that provide support for preservation of transactional properties and resource management. However, expert knowledge is required to customize the container properties match the needs of the application. Java 2 platform and COM+ (now part of .NET) by Microsoft are examples of platforms that provide appropriate components and run-time environment support for this approach. We have also adopted this approach that separates the business application logic from the infrastructure logic. The application developer creates the application code and then, with the help of the QoS engineer and the toolkit, (s)he should modifies the application to include provisioning of the QoS.

We assume that the monitored software is a server built using Java technology and has the usual structure. Furthermore, when internally monitoring the component's execution, we are concentrating on measuring selected activities in terms of delays. Other types of observed data can also be reported, but these are application specific. Besides monitoring the components internally, monitoring of general parameters describing the environment in which the components operate is also facilitated by the toolkit. To measure delays of specific activities performed in the task of servicing the client, time-stamps are gathered at specific points (i.e., time-stamps before and after the monitored activity) and inserted in a common data structure (buffer). The action of taking time-stamps and buffering time-stamps is performed by code, referred to as a *monitoring* code, inserted by a QoS engineer into the component using the toolkit. It should be noted that the monitoring code is not responsible for the delivery of the collected data to CA – this is a task of a reporting agent facilitated by the toolkit. The reporting agent/thread simply reports the data gathered by monitoring to CA. The frequency of reporting is according the control variables that are set upon instructions from CA.

The other type of monitoring data that is collected is on the state of selected parts of the system. For instance, we may require the current state of a link by examining the communication software. How probing of the state of subsytems is facilitated by the toolkit is described in the subsequent section.

3. PROBING SYSTEM STATE

To collect the various (sub)system parameters we use probing agents that periodically issue probes as initially requested by the QoS engineer and as controlled by the CA through control variables. The probes are scheduled using a timer. A probing agent simply issues a probe, waits for the probe's result/data and reports it to the CA. A probe-record has data that is specific to the probe and its results. It also contains information that identifies the probe. Once one probe is handled, the next probe of that type is scheduled, if at all, at a time that is determined by control variables. Thus, there is an agent for each probe type. In essence, a probe is simply an execution of a diagnostic program provided by (available on) the (sub)system that we are probing. There are various diagnostic tools available for Unix and Windows platforms that provide the current state of the operating system and the network interfaces and these two platforms are our main target. The probing agent periodically launches a probe, that is it connects to the specified system and starts a diagnostic program, receives a

report(s) from it, and buffers it and then reports it to the CA. As a consequence, standard I/O operations on the used diagnostic program have to be available, otherwise the program is not suitable for our purposes. For instance, a windows-based program that obtains system parameters, but only interacts with the user using a GUI and does not pass them through standard I/O operations, is not suitable. In addition, care must be taken as processes that execute on some native platforms lock when there is an insufficient buffer size for standard I/O streams.

There are many system and diagnostic tools/programs available for Unix and Windows-based platforms and the following list shows samples in three categories: system, network and DB. Each diagnostic program/command is described very briefly while the platform on which the program operates is shown in parentheses.

- *System Diagnostic Tools:*
  - TOP – (Unix) provides details on top ten processes executing and an overall analysis of system performance.
  - PS – (Unix) reports on processes that are currently executing in memory.
  - VMSTAT – (Unix) provides a summary on the current state of the operating system.
  - MEM – (Windows) offers the current memory states on the MS-DOS system.
  - TLIST – (Windows NT) lists all current processes and their states.
  - MEMSNAP –(Windows NT) provides a snap shot of the memory and other resources.
- *Network Diagnostic Tools*
  - PING – (Unix and Windows) determines the availability of a connection.
  - NETSTAT – (Unix and Windows) provides information on the network interface.
  - TTCP – (Unix) platforms, tests a connection delay between two machines.
- *SQL Execution Diagnostic Tools*
  - EXPLAIN PLAN SQL – In Oracle and DB2, it retrieves the execution plan from the optimizer.
  - SQL TRACE – retrieves resource consumption for any SQL statement.

We have successfully used our toolkit on each of the above programs. To probe a (sub)system, its diagnostic command/program is executed. The command returns its data that is buffered and then periodically delivered to the CA by a reporting agent. A diagnostic program reports on various detailed aspects of the particular subsystem. The CA might be interested only in specific pieces of the data reported by the command or only in a particular summary of the reported data. Furthermore, diagnostic programs supply detail information using various data types. The toolkit must provide assistance not only deployment of probes but also in collection of the data.

The probe is instrumented in the following way using the toolkit. First, the diagnostic command is executed directly. The command's output is parsed and shown to the user/engineer, who has the ability to choose which fields/entries are to be discarded, which are to be captured, and also perform rudimentary summaries, such as adding the entries in rows, columns or counting the rows. These manipulations are captured in a probe specification and the probe can be re-issued and its new output can be examined. The toolkit also provides for saving of data in a DB table. We discovered that saving the data reported by the diagnostic programs in a structured form in a DB, that the QoS engineering can query and work with, was by far more preferred and easier then when the engineer worked with the reported data stored in a text file (or reported to the CA in a text format). Further presentation will thus concentrate on storage of reported data in a DB – we shall just note that it is the engineer's option to specify where the reporting agent should send/report the buffered data: file, DB, or CA. Thus, it is expected that the engineer, when developing a probe, would first select storing of probe's data in a relational DB. When the probe is successfully tested, only then would the engineer instrument the probe to report to the CA, using a TCP/IP connection, instead of the DB.

4. PROBE INSTRUMENTATION -- EXAMPLE

Figure 2 shows an example of the toolkit's Probe Panel in use by an engineer. The engineer had executed the MEMSNAP command. The command's output is loaded into a Jtable (identified in the figure by the title "Command Output") organized as a two-dimensional array. Thus the text data, returned by the command, is parsed while using blanks as separators for column entries in a row, and end-of-line indicators serving as new-row indicators. All of the commands that we have listed in the previous section display properly in such a two-dimensional array using this technique. Note that the first row of the table starts with column entries that are headings produced by the command: Process, ID, Proc.Name, Wrkng.Set, etc. Also, a time-stamp is shown in column K. It has been added by the toolkit so that the engineer can specify that a time-stamp is to be generated by the probe, added to the data reported by the command and reported to the CA (or stored in a file or a DB).

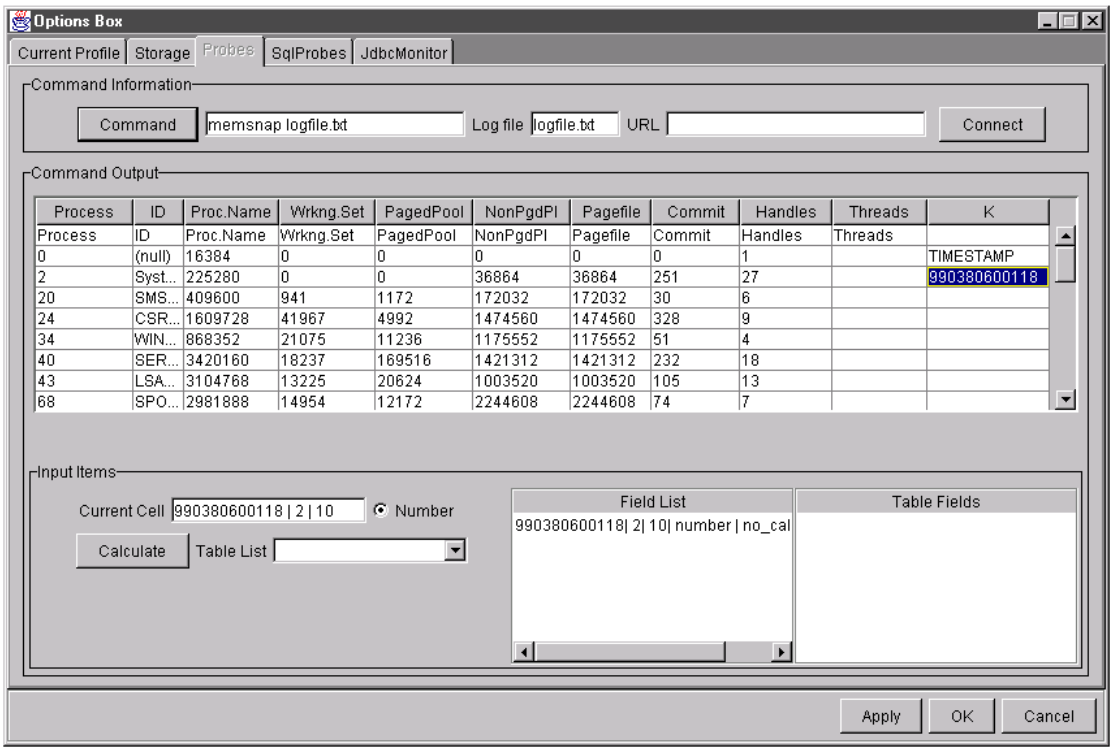

Figure 2: Data Produced by the MEMSNAP Diagnostic Command Execution

What the engineer needs to specify is which cells are to be captured, as not all may be relevant and thus should not be reported to CA. The engineer can specify which data is relevant by double-clicking on a cell and data in that cell (column) will be placed in the Field List. The figure shows that the engineer has double-clicked on the time-stamp in the column K of the Command Output table and therefore the time-stamp has been added to the panel labelled Field List. This simple method can be used by the engineer to identify which data is to be captured from the command's output. If the data is not to be saved in the DB (i.e., it is to be either reported to the CA or stored in a file), the engineer can press the apply button and the configuration is saved and the probes can be scheduled. However, the engineer may only want some summary of the generated data produced by the command. For instance, in the case of the MEMSNAP command, only the number of processes, threads, and the name of the system may be of interested and thus should be captured and reported to the CA.

To capture data in a relational table, the engineer must first create the table using facilities of a "regular" DBMS, such as, for instance, Oracle. Recall that a command/probe is executed periodically and the periodic report is stored in a relational table. In our MEMSNAP sample probe, the table is called System. We (the engineer) have created a number of tables to capture data from some of the commands listed in the previous section. The tool can use a Storage panel to see these tables and to retrieve data from them. Figure 3 shows the Storage panel in which names of four tables are shown (Network, Query, QueryCost, and System). For each table, its attributes (columns) are listed. For instance, for the table System, attributes syst_timestamp, no_processes, no_threads, no_handles, and c_name are listed (where the prefix "no_" stands for the "number-of" and thus no_processes is the number-of-processes). It is in this table System that we want to save data reported by the MEMSNAP command. The Storage Panel provides for connecting to the DB and examining the schemas and contents of the tables.

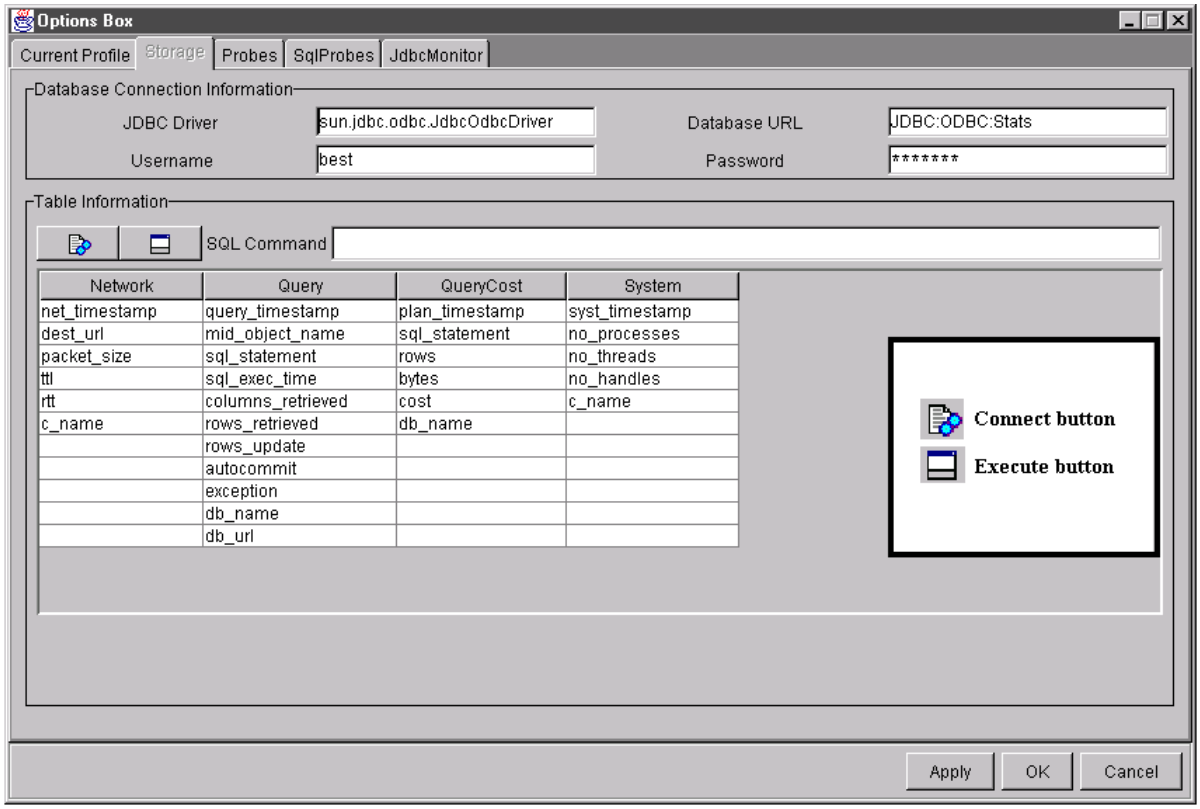

Figure 3: Storage Panel Showing DB Relations Used to Store Monitored Data

Figure shows that the table System was indeed chosen by the user and is thus shown in Table List, which is a drop-down-list. As a consequence of this, the fields/attributes of the System table are listed in the panel labelled Table Fields (also shown in Figure 4). The MENSNAP command generates detailed data on the processes, their working sets, and such. This is more detailed than the engineer may want. For instance, the probe should simply report the number of processes, threads, etc. What the engineer needs to do is to count the number of processes (number of rows shown in the Command Output), count the number of threads (this time add entries in a column as each entry represents the number of threads per process), and count the number of handles (add entries in a column). We provide facilities for just that. The engineer has already selected that a time-stamp is to be saved in the first entry in the panel labelled Field list. What was also specified is counting of rows in a column to count/determine the number of processes (using selection count_columns), and adding entries in a column (using selection add_columns) to calculate the total number of threads – these are also shown

in the Field List as second and third entries (of Figure 4). The figure shows that the engineer is in the process of specifying that numbers in a column (labelled as Handles) are to be added to count the total number of handles (by choosing the option add_columns). Once this is done, to indicate which fields (stored in the Field List) are to be stored in which fields of the System table (shown in Table Fields), the entries in the Field List are simply dragged and dropped onto the entries in the panel Table Fields. The result of this is shown in Figure 5 (which only shows a bottom part of a Probe Panel). The entry c_name, in Table Fields, is assigned a default value of Ironhide – the process of doing this is also shown in Figure 5.

The user can then modify the control variables that control the frequency of probes and reporting of monitored data and then save the configuration and launch probes.

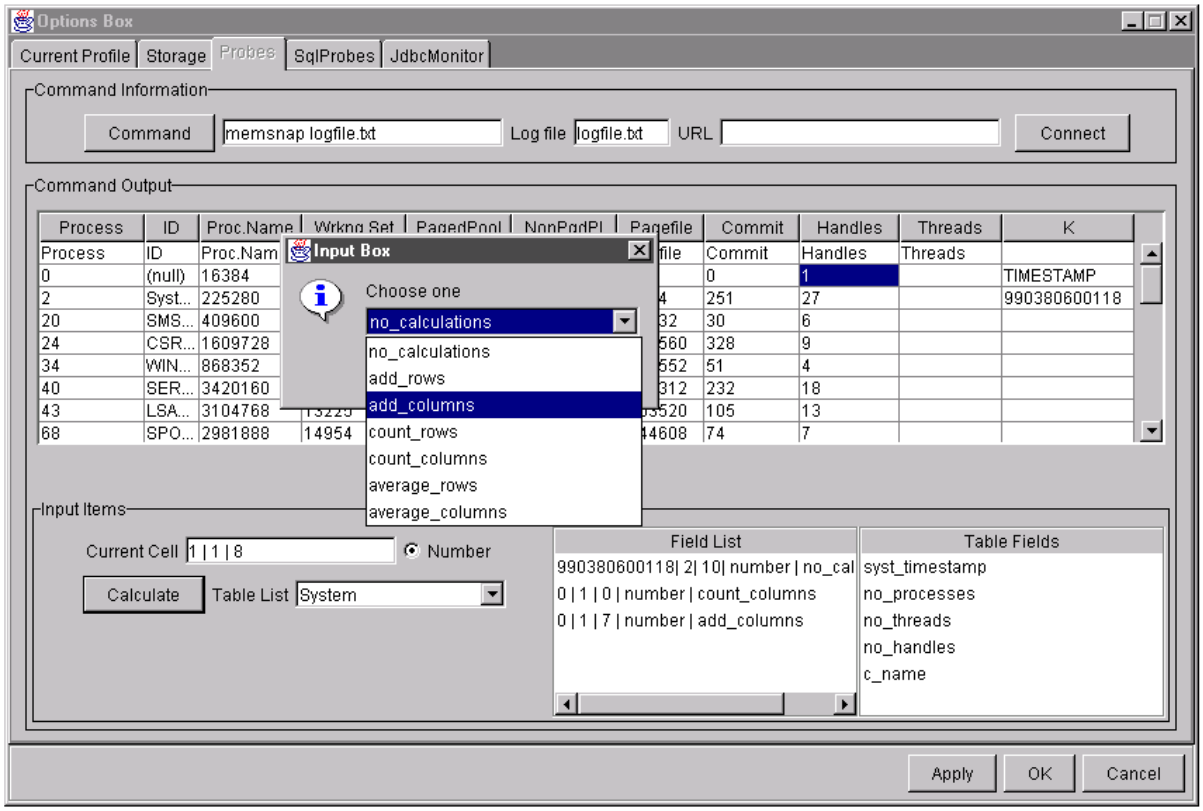

Figure 4: The Probes Panel – Command Execution and Specification of Output Capture

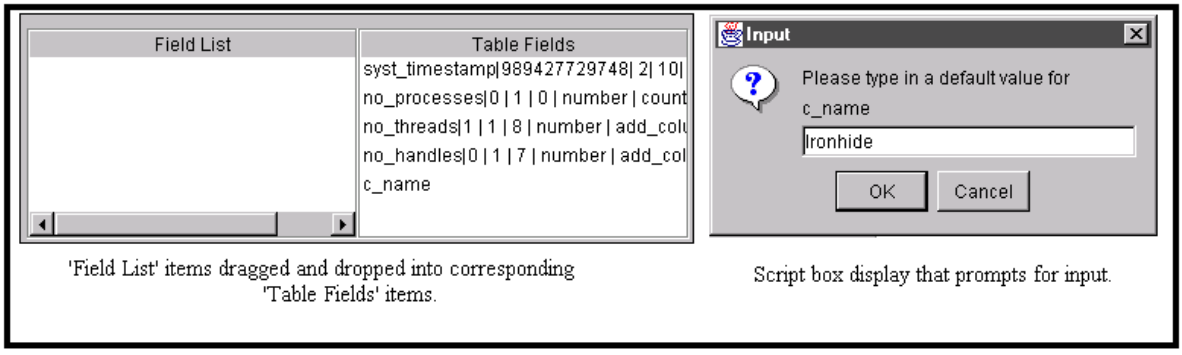

Figure 5: Saving Field List in Table Fields of the Probe Panel

We also used and extended the toolkit to facilitate capturing results of diagnostic tools for DBs, in particular EXPLAIN PLAN SQL and SQL TRACE. Interested reader is referred to [Best 2001] for further details. We shall just mention that in terms of the control variables, the toolkit only provides for setting their initial values – at this time it does not provide the engineer with the facility to modify the control variables while probing is in progress.

5. RELATED WORK

Closest to our work is project QuO, Quality Objects, which provides a framework for QoS for CORBA (Common Object Request Broker Architecture) distributed objects [Schantz 2000, Pal 2000]. It provides for negotiation and monitoring of contracts. Our approach is similar with QuO in that the code is developed first and then a developer using a toolkit adds the QoS mechanisms. We enable detailed measurements of various activities within a specific service provided by a distributed object. We also provide for monitoring of pertinent parameters that define the system environment in which servers execute. QuO, however, is comprehensive in that it supports negotiations of contracts between the clients and servers while we concentrate on the monitoring aspects.

In [Malenkamp 2000] probes are used to determine delays associated with particular activities. For instance, to determine a delay associated with a transfer of data between two network points, a probe is used that results in actual sending of a message. The message solicits a response while the round-trip delay is measured. We adopt a more generic notion of a probe. A probe may be just that, sending a message, but a probe may solicit information from the specific subsystem in question by executing a diagnostic program on that subsystem. For instance, a probe may be asking a processor that hosts the server about its current load.

There is much literature on the subject of QoS, but we would like to mention [Hafid 1996, Ye 1999] that deal with QoS for DB activities. They propose that, in general, query execution strategies are static; that is, once they are formulated they remain unchanged regardless of changes in a system. As a query is executed, the environment in which the strategy executes should be monitored so that it can be modified in response to the changes in the environment. The monitoring process and adaptation to the changing environment, however, is enacted internally by the DB system. This is similar to the notion of dynamic or adaptive query processing in [Bodorik 1991]. Here, we assume that the QoS is provided externally to DB systems and thus we do not have control on how a DB system performs its work. We do probe the DB system, however, asking it about properties of the queries that are to be executed.

6. SUMMARY AND CONCLUSIONS

This paper described provisioning of QoS monitoring of execution of middleware components. Two types of measurements are available, both of which are required. In one, time-stamps are used to measure delays of critical activities. The difficulty with this approach is that QoS engineer that instruments the monitoring must know which activities are critical to QoS and hence must be measured. In the second one, probes are used to determine parameters that characterize the state of the system. The probes are used to solicit parameters from subsystems, such as the load on the system. System requests for Unix and Windows OS environments are supported. Details on a toolkit that facilitates instrumentation of the probes have been presented. The toolkit has been used successfully to probe OS, network, and DB systems.

Although the toolkit presented here is a good solution, for wide acceptance standardization efforts for particular middleware applications are required. Ultimately, QoS for Java applications should be provided in a manner similar to transactional properties provided by containers of the Java 2 platform. Applications should be developed independently by the application programmers while the QoS properties would be provided by containers.

REFERENCES

Best, S. (2001). Framework for Monitoring Middleware Components in Support of QoS. Master of Computer Science Thesis, Dalhousie University, Halifax, Nova Scotia, April 2001.

Bodorik P., J.S. Riordon, and J. Pyra (1991). Deciding to Correct Distributed Query Processing. In Transactions on Knowledge and Data Engineering, IEEE, Vol. 4, No. 3, 1991, 253-265.

Bodorik, P., S. Best, and D. N. Jutla (2002). Toolkit for QoS Monitoring in Middleware. In Proceedings of the 4th International Conference on Enterprise Information Systems, April 2002 (to appear).

Hafid A., G. Bochmann, and B. Kerherve (1996). A Quality of Service Negotiation Procedure for Distributed Multimedia Presentational Applications. In Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing, 1996, 330-339.

Jutla, D.N., P. Bodorik, and Y. Cay (2001). Interoperability for Accessing DBs by E-commerce Applications. In Proceedings of HICSS 2001, CD-ROM (IEEE Press), 10 pages.

Katchabaw, M., S. Howard S., H. Lutfiyya, A. Marshall, and M. Bauer (1999). Making Distributed Applications Manageable through Instrumentation. In Computer Communications, Vol. 45, 1999, 81-97.

Le Tien, D., O. Villin, and C. Bac (1999). Lightweight Objects for QoS Management in CORBA. In 10th International Workshop on DB and Expert Systems Applications, 1999, 914-918.

Molenkamp G., M. Katchabaw, H. Lutfiyya, M. Bauer M., (2000). Managing Soft QoS Requirements in Distributed Systems. In Proceeding of the International Workshops on Parallel Processing, 461-468.

Pal, P., J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier (2000). Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2000, (ISORC 2000), 310-319.

Schantz, R., J. Zinky, J. Loyall, R. Shapiro, and J. Megquier (2000). Adaptable Binding for Quality of Service in Highly Networked Applications. In Proceedings of the International Conference on Advances for Infrastructure for Electronic Business, Science, and Education on the Internet, July 31-Aug 6, 2000.

van Morsel, A.P.A. (1999). The 'QoS Query Service' for Improved Quality-of-Service Decision Making in CORBA. In Proceedings of the Symposium on Reliable Distributed Systems, 1999, 274-285.

Ye, H., B. Kerherve, and G. Bochmann (1999) QoS-aware Distributed Query Processing. In Proceedings of the Tenth International Workshop on Database and Expert Systems Applications, 1999, 923-927.