Experimenting Middleware-Level Monitoring Facilities to Observe Component-based Applications

Eduardo Fonseca, Sand Corrêa, Renato Cerqueira

[1]Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente, 225 – Rio de Janeiro – Brazil

{efonseca,scorrea,rcerq}@inf.puc-rio.br

Abstract. *As distributed component-based applications increase in size and complexity, on-line application monitoring becomes a crucial issue for assuring quality of service. In this paper we propose a monitoring architecture built into the component system itself that allows a fine-grained observation of the resources being in use by an application. Then, we provide an evaluation of the overhead imposed by the architecture carrying out experiments and observing the resource-usage behavior of two different types of application (data-intensive and message-intensive). The results show that the monitoring cost can be very affordable.*

1. Introduction

Component-based systems have been widely used to build distributed, service-oriented environments, both in academic and industrial projects. In such environments, a key concern is the assurance of the quality of service, which traditionally has been treated during the developing phase using, for example, regular tests and debugging tools. Theses approaches are still very important for component-based applications. However, as these applications increase in size and complexity, completely anticipating their runtime configurations at developing phase inevitably becomes a brittle task. This difficulty in anticipating all runtime configurations in which a component, or a set of components, will be used is due to two main reasons. First, software components are binary units of independent deployment and versioning and, therefore, they can be subjected to third-party composition [Szyperski 2002]. For this reason, in component-based programming, service quality is not associated only to the software component itself but also to its interactions. Second, as the number of components increases, the dynamic interactions among them become more complex, mostly due to problems related to heterogeneity, concurrency and, more generically, to specific details of their execution environment. These challenges make on-line monitoring an important issue for component-based applications. Furthermore, many new application requirements, such as self-management [Kephart and Chess 2003] and context awareness [da Silva Santos et al. 2007], have on-line monitoring of system's properties as part of their solution approaches.

In this context, the goal of our research is to explore abstractions and methods to develop on-line monitoring and behavior observation techniques for component-based systems. Toward this end, in this paper we focus on an architecture for observing the behavior of component-based applications using a monitoring infrastructure built into the middleware itself. Particularly, our architecture has the following features: (1) it is driven by an observing mechanism that allows to monitor system and application-level metrics;

and (2) it relies on the deployment entities provided by the component system that fully controls the execution environment. Bringing to the middleware the task of monitoring the execution environment has the following advantages: (1) the monitoring mechanism is kept independent from component implementations and therefore, it is possible to rely on an unique mechanism to monitor the entire system; (2) it is possible to design monitoring facilities that are flexible enough to be useful to different applications; and (3) the solution provides a transparent way to monitor and control the environment, without changing the applications.

In order to evaluate our architecture, we first provide a qualitative assessment of the proposed monitoring infrastructure. We then provide a more quantitative evaluation, measuring the overhead imposed by the monitoring mechanism on the overall performance of the observed applications. We carried out experiments monitoring two different types of application: MapReduce [Dean and Ghemawat 2004] and PingPong. The former is a data-intensive application while the later is built from message-intensive components.

The rest of the paper is organized as follows: Section 2 gives an overview of the component system we have used in this work. Section 3 describes the proposed monitoring architecture in detail. In Section 4, we present an evaluation of our architecture, which includes the qualitative assessment and the quantitative experimental results. Section 5 describes the related work. Section 6 presents conclusions and ideas for future work.

2. SCS

In this section, we introduce briefly SCS (Software Component System) [SCS 2008], the component system we have used to represent and deploy a service (or set of services) in this work. SCS is a CORBA-based component system, conceived to provide a simple infrastructure to easily support the configuration, distribution and deployment of component-based applications. To achieve this goal, SCS design was inspired on some aspects of COM [Box 1997] and the OMG CCM Specification [Wang et al. 2001], but it tried to avoid most of the complexity imposed by these models. For this reason, in SCS, the core functionalities of a component system are represented as a small set of interfaces, grouped into two categories: a SCS core component model and a runtime environment.

The core component model defines the common behavior of all SCS components, as well as the way they interact with each other. The component model provides two types of ports for interaction among components: facets (the service provider ports) and receptacles (the required service ports). In general, a component has as many facets and receptacles as it needs. However, the model offers three specific interfaces to compose the behavior of a component: (1) *IComponent*, a mandatory interface that defines the basic operations for all SCS components; (2) *IMetaInterface*, which defines operations for component introspection; and *(*3) *IReceptacles*, which defines operations for managing the receptacle connections in a component.

The SCS runtime environment provides a standard deployment model in which components are instantiated, loaded, configured and executed. This support is implemented by two key SCS services: Execution Node and Container. A Container service defines a host process for a SCS component or group of components. Its main goal is to provide an isolated address space for component execution, preventing that failures in a

component do not take down the entire component system. In this way, containers control component creation and loading. An Execution Node is the SCS system's entry point in a specific computer and is responsible for the creation and management of component containers in a host.

3. The Monitoring Architecture

Figure 1 illustrates our architecture for enabling monitoring capabilities to a component-based environment. The architecture comprises two layers. The application layer hosts the execution environment. SCS manages this layer, controlling the conditions in which components are deployed and executed. Applications are composed from SCS components, loaded within containers instantiated in a specific execution node. The monitoring layer, on the other side, monitors the execution environment observing properties that evolve over time. Monitoring is achieved via an information model and an observing mechanism, as described below.

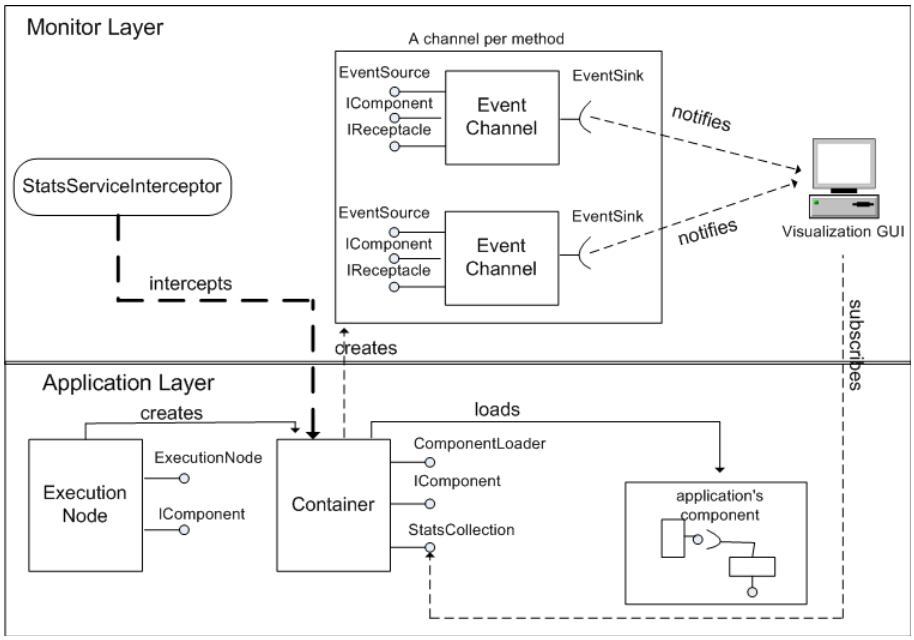

Figure 1. The monitoring architecture approach

The Information Model

A crucial issue in monitoring infrastructures is the information model, that is, the description of the data to be observed. In the context of quality assurance, the information model describes the set of information to be observed in order to evaluate the quality of the services. In our monitoring approach this model consists of two groups of information: request-reply message and resource usage information. As it will be explained later, both groups of information hold statistics about events that happened inside a container.

Request-reply message observations are application-level metrics that hold statistics about resources used during a method invocation. They are essential service quality sensors since components can serve multiple clients at the same time and, thus, some requests may influence the response of others. We have used three structures, namely

methodName	identifies the method the structure refers to.
callCounts	contains the number of times the method was invoked inside the container.
cpuTime	contains the estimated CPU time (in milliseconds) spent by the method since the container's creation.
elapsedTime	contains the total execution time of the method (in milliseconds) since the container's creation.

Table 1. *MethodStats* structure field descriptions

MethodStats, *InterfaceStats* and *InterfaceStatsSeq*, to represent request-reply message observations. Each structure represents a different level of granularity of the information. *MethodStats* is the finest-grained structure and represents the statistical information of a method execution. This structure consists of four fields whose names and content descriptions are listed in Table 1. The structures *InterfaceStats* and *InterfaceStatsSeq* summarize the request-reply message observations obtained from an interface and group of interfaces (for example, the interfaces belonging to a component) respectively. They represent the statistical information of each method belonging to an interface and group of interfaces.

Resource usage observations are system-level metrics that hold statistics about the resources used by a container during its execution. They are also important quality sensors, since they can evaluate the performance of the process hosting the services. We use the *ContainerStats* structure to represent the resource utilization observed from a container. This structure consists of four fields whose names and content descriptions are listing in Table 2.

In addition to the CPU and memory usage information, I/O and network usage is also reported in our information model. However, due to space limitation, we will not present it here. Finally, it is worth noting that request-reply message and resource usage information do not exhaust the set of possibilities to control service quality but we consider them as the most important. Other service quality sensors can be useful to monitoring mechanisms, as pointed out in [Wang et al. 2007].

The Observing Mechanism

The observing mechanism is responsible for monitoring the metrics defined in the information model. To achieve this goal, it relies on two facilities: the container process itself and the interceptor mechanism [Debusmann et al. 2002] [Marchetti 2001]. The former is responsible for obtaining its resource usage information (described in Table 2) by periodically invoking calls to the operating system. The later is a mechanism that allows to insert control functionalities into the invocation path between the client and server components. For this reason, in our monitoring mechanism, interceptors are attached to containers to provide request-reply information of all incoming and outgoing remote method calls.

The monitoring layer of Figure 1 represents the observing mechanism. *StatsServiceInterceptor* is the interceptor introduced to the architecture to get details of all messages to and from the container it is attached to. *StatsServiceInterceptor* provides a snapshot of the resource usages of a specific method in a given moment, that is, the request-reply message information described above. To achieve this goal, it: (1) intercepts any request targeting a component loaded in the monitored container; (2) using the invocation context,

containerName	identifies the container the structure refers to.
cpuTime	contains the CPU time (in milliseconds) spent by the container since its creation.
cpuUsage	represents the container's CPU utilization since its creation
memoryUsage	represents the container's current memory space in Kilobytes.

Table 2. *ContainerStats* structure field descriptions

extracts the name of the method invoked as well as the interface it belongs to; (3) obtains a snapshot of the CPU usage and time before the method being executed; (4) redirects the invocation to the target object; (5) intercepts the reply, obtain a new snapshot after the execution and estimate the CPU usage and elapsed time; (6) updates the statistics of the method and redirect the reply to the client entity.

To allow an external agent to obtain statistics related to the container process and the execution of its methods, the *StatsCollection* interface is provided as a facet of the container service. As shown in Listing 1, this interface defines operations to expose request-reply message information for a specific interface (*getInterfaceStats*) or group of interfaces (*getComponentStats*), as well as container's resource usage information (*getContainerStats*). In addition to these methods, a *subscribeMethodNotification* operation is also defined in order to allow interested clients to register themselves to receive notification about request-reply information of a specific method. In response to a client subscription, the container creates an event channel (using the SCS support for event-based communication) through which the client will be notified whenever a new information is made available. Currently, the container creates one channel for each method under observation request. Figure 1 shows an example of such clients, the *Visualization GUI*, a Graphical User Interface (GUI) provided along with the framework to allow the visualization of monitored properties.

As the monitoring infrastructure is built into the component system itself (through its entities of deployment), the use of the proposed architecture is straightforward. The monitoring functionality is enabled or disabled using a flag attribute set in the component system configuration process. By enabling this flag, each container created by the application will be endowed with the monitoring capabilities described here. Furthermore, users can configure the period in which the observing information will be generated.

Listing 1. IDL definition of the StatsCollection interface

```
1  interface StatsCollection {
2    MethodStatsSeq getInterfaceStats(in string interfaceName)
3    InterfaceStatsSeq getComponentStats()
4    ContainerStats getContainerStats();
5    boolean subscribeMethodNotification(in string clientName,
6           in string ifname, in string method, in event_service::EventSink sink);
7  }
```

4. Evaluation

We have implemented the proposed monitoring architecture using the Java implementation of SCS, which is based on Sun JDK 1.5 ORB. The request-reply message information

is obtained using the Sun ORB support for interception, while the resource usage information is extracted from the operating system using a C library. Communication between the Java Virtual Machine and the native C code is handled by the Java Native Interface (JNI). Next, we provide a qualitative assessment of the proposed framework. Then we provide a quantitative evaluation of the impact of the monitoring mechanism's overhead on the overall performance of two types of monitored applications: a data-intensive application and an application built from message-intensive components.

4.1. Qualitative Assessment

The proposed architecture is driven by an information model and an observing mechanism built at middleware-level. With respect to the information model, it allows observations on different level of granularity, ranging from operations and interfaces to components and processes hosting these components. This information model was sufficient to help us to identify most of the performance bottleneck we have faced in our experiments. An example was the cluster configuration, which was made using the framework support after several experiments observing the elapsed time of the methods in different combination of machines. There are however some limitations to the information model. The observation of more fine-grained programming elements, such as operation parameters and results, are not supported yet.

With respect to the observing mechanism, the choice for a middleware-level approach brought the advantage of decoupling the monitoring infrastructure and the application implementation. Particularly, we make use of containers and interceptors to achieve this goal. The data collection procedure invoked by the container service is implemented in C and is adapted for each operating system. We chose this approach because it is independent from virtual machine technologies and can be applied to different implementations of the component model. The use of interceptors made possible to keep track of the remote invocations and to manipulate them according to ours needs, transparently. On the other hand, indeed interceptors introduce an overhead problem. However, as it will be verified in the experiments, this overhead can be very affordable when compared to other system latencies.

4.2. Quantitative Evaluation I: The MapReduce Application

This experiment deals with a MapReduce application and the case study was derived from [Dean and Ghemawat 2004], which describes a Google's library to support parallel computation over large data set on unreliable clusters of computers. MapReduce is based on two concepts from functional languages to express data-intensive algorithms: map and reduce functions. The Map function processes the input data and generates a set of intermediate $<key,value>$ pairs. The Reduce function, then, merges the intermediate pairs that have the same key. An overall MapReduce application's dataflow is shown in Figure 2. In the map phase, the user input data is split into smaller pieces and assigned to workers (copies that execute on a cluster of machines). These workers, then, execute user-defined map functions to produce the intermediate $<key,value>$ pairs. Also during this phase, partitions and sort functions are executed to, respectively, distribute the pairs around the key and sort them, so that occurrences with the same key are kept together. In the reduce phase, workers iterates over the sorted data executing user-defined reduce functions. Those functions produce one or more outputs, which can be merged to produce a single output.

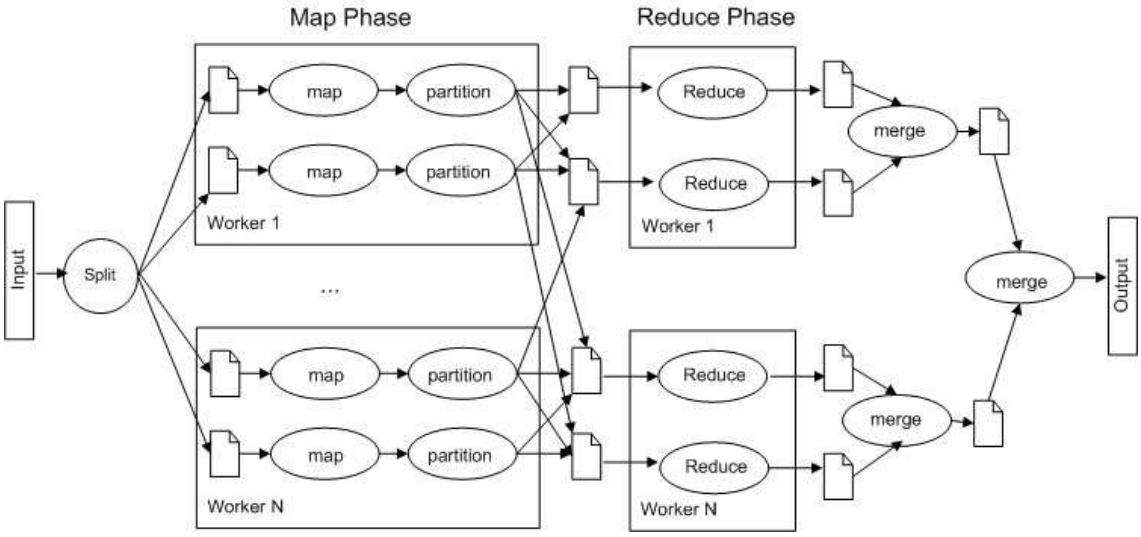

Figure 2. A MapReduce application dataflow

Figure 3 shows our implementation of a MapReduce framework using SCS components. Broadly the framework comprises two types of component: master and worker. The master is a special component that controls the entire dataflow, picking idle workers and assigning them a map or reduce task. The master executes within its own component container, while worker components execute user-defined map or reduce functions and are instantiated in different execution nodes, in individual containers. Master assigns tasks to workers, invoking an *execute* method (defined in the *Worker* interface) and passing as parameters the input location, the task type (map or reduce), the output location and a SCS event channel. The later is used by the worker to notify the master that the task has been completed. In this way, when the application is executed enabling the monitoring flag, each worker has its request-reply and resource usage information periodically monitored by the container in which it is instantiated. This information can be further used to, for example, evaluate the performance of each node where the components are executing and help to make decision about worker instantiations.

In order to evaluate the proposed monitoring architecture with respect to its performance overhead, we run a MapReduce application to count the number of occurrences of each word in a 100 MB data file. The application was executed in a cluster consisting of 10 machines. Each machine had two 3 GHz Intel Pentium IV processors, 3 GB of memory and a 1 Gigabit Ethernet link. The input data was split into approximately 2 MB pieces processed by 30 workers (3 workers per machine, each one instantiated in its own container). The output was placed in 6 files (therefore, from the 30 workers 6 of them also performed reduce functions). We configured the data collection procedure (collecting the resource usage information described in Table 2) to be fired at 5-second intervals.

We have executed the application in three situations: enabling the monitoring capability (full instrumentation), disabling the monitoring capability (no instrumentation) and finally, enabling the monitoring capability to perform a simple pure interception (an interception that does nothing). This last experiment allowed to evaluate the overhead caused by the data collection procedure. The experimental result is shown in Figure 4. It is easy to see that the cost of monitoring is very affordable: the monitoring cost in the

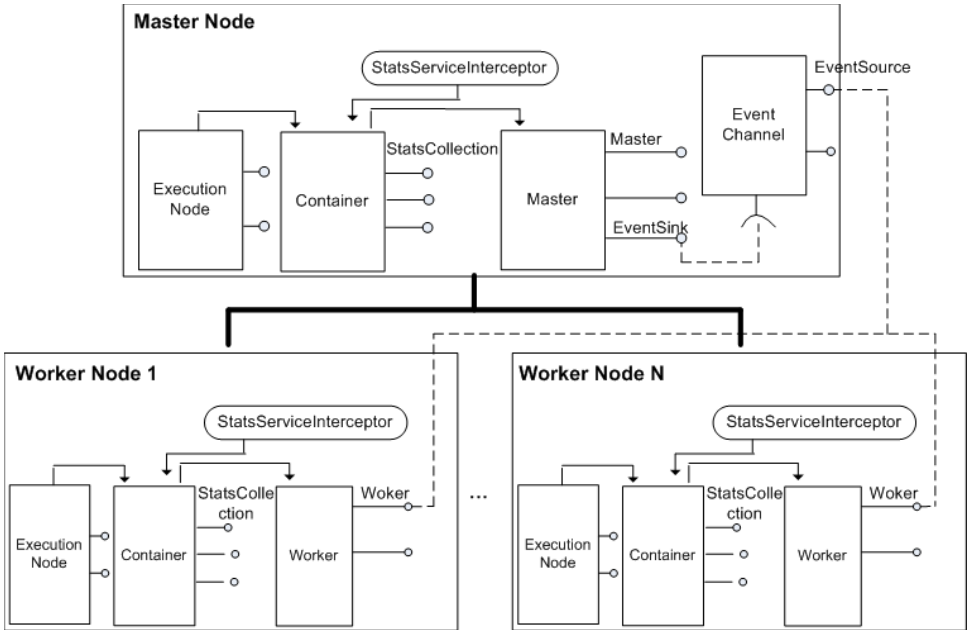

Figure 3. A MapReduce framework implemented using SCS components

three experiments was really close, showing that the monitoring overhead penalty was dominated by communication and I/O latencies. Since MapReduce is a data-intensive application, this domination tends to increase as the data input file increases. We also measured the average time spent to setup a single container in the three experiments. As shown in Figure 5, the average setup time ranges from 610 to 614 ms. Again, the difference among the setup times is insignificant, showing that the use of interceptors does not incur penalties for the container startup process.

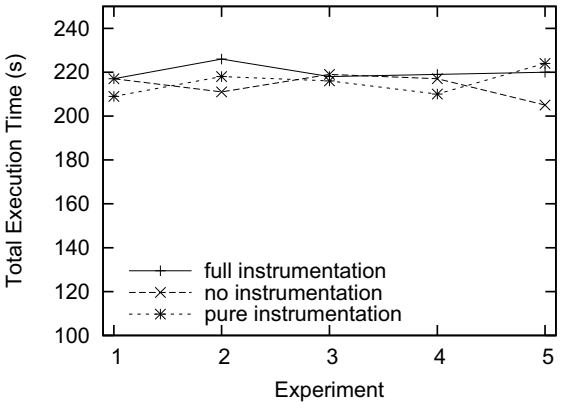

Figure 4. Application response time in the MapReduce experiment

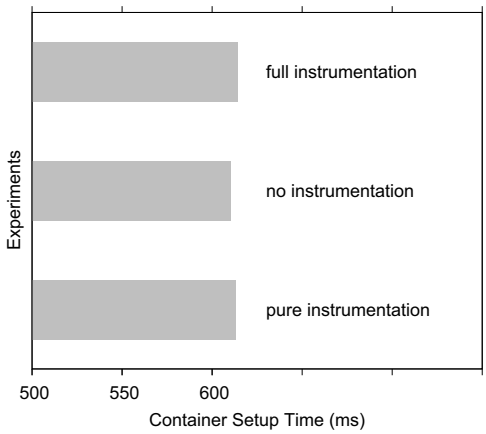

Figure 5. Container startup time

4.3. Quantitative Evaluation II: The PingPong Application

In this experiment we evaluate the overhead introduced by the monitoring framework in an application built from message-intensive comonents. The case study consists of an application comprising two *pingpong* components connected to each other and each component merely forwards any message it receives to the other component. The components

are instantiated in different nodes as shown in Figure 6. In this experiment, we configured the data collection procedure to be fired at 1-second intervals and the application was configured to exchange 1000 (pong) messages. Similarly to the previous experiment, we analyzed the response time of the application in three situations: enabling the monitoring capability, disabling the monitoring capability and performing a simple pure interception. The experimental result is shown in Figure 7. This scenario involves only communication latency and the infrastructure's performance overhead, since each application component just forwards every received method call to the other component. This experiment gives an idea of the worst case of the monitoring mechanism and can help to evaluate if the performance overhead is affordable or not for a given application. It is important to mention that the small difference in the response time of the full and pure instrumentations indicates that the data collection procedure doesn't impose a larger burden on the monitoring mechanism.

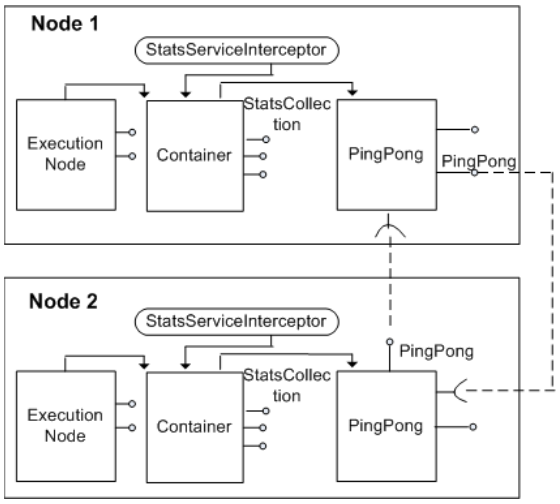

Figure 6. PingPong application implemented using SCS components

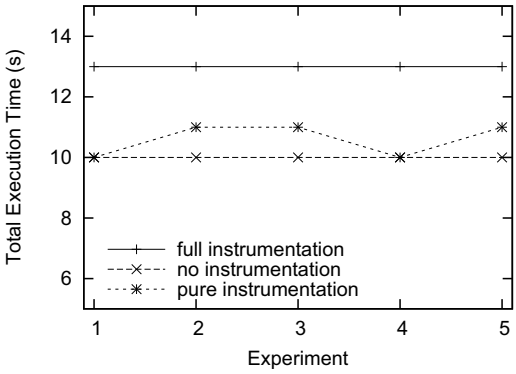

Figure 7. Application response time in the PingPong experiment

5. Related Work

There has been a lot of work in the domain of on-line monitoring in distributed systems [Schroeder 1995] [Fickas and Feather 1995]. In recent years, researches both in academia and industry have focused on the issue of introducing on-line monitoring capabilities for component-based applications. Some commercial frameworks, such as [WebSphere 2008], have already been devised to achieve this purpose. However, most of the commercial approaches currently available are toolkits targeted to specific technologies (specially EJB). Conversely, the monitoring infrastructure presented in this paper relies on the deployment entities of the component system and thus it can be easily extended to different implementations of the model. The main requirement imposed by our architecture is the support for interceptors in the middleware layer. However, nowadays only few middleware technologies, such as Java RMI, do not provide support for interceptors.

Apart from the commercial frameworks, some academic initiatives have been proposed. [Diakov et al. 2000] implements a monitoring support that is responsible for submitting reports on the occurrence of remote calls and life cycle events. Contrary to our

approach, where the monitoring support is implemented in the middleware via containers and interceptors, their basic support for monitoring is implemented by a software library bound with every application component. This library is responsible for reconfiguring the proxy object to notify the monitoring support, in the application component, about the ongoing remote invocation. Similarly to our approach, in [Rackl et al. 2000] a monitoring framework built at the middleware-level is proposed. However, in contrast to our work, the framework implementation is based on non-standardized interfaces on the ORB.

More recently, reflective middlewares have been used to support the implementation of control requirements, such as monitoring capabilities, by allowing the inspection and reconfiguration of its internal engine. A good example is provided in [Kon et al. 2000]. The work describes *dynamicTAO*, a CORBA reflective ORB that supports dynamic configuration. The work presents a flexible monitoring service as one of the scenarios to explore system reconfiguration. However, the level of granularity is the CORBA object, whereas our monitoring architecture supports the component abstraction.

6. Conclusion and Future Work

On-line monitoring is an important issue for distributed component-based applications. The monitoring architecture we presented supports monitoring of resource usage in different programming elements (method, interface and components). Furthermore the monitoring mechanism is built into the component system and thus the application is relieved of the monitoring issue. Additionally, we have implemented two different experiments that show that the communication and I/O latencies can overcome the overhead imposed by the monitoring framework.

Work on our monitoring architecture will continue in several directions. We have already started the introduction of mechanisms to extend the collected metrics. In this extension, users can write a class to collect specific data, like the CPU temperature, and provide the class implementation to the container service. The later will use this implementation to collect the new metric. Thus, it will be possible to extend the monitored properties dynamically. As stated before, monitoring is just one of the steps for quality assurance. We are also investigating reasoning capabilities to be introduced in our architecture so that the monitored information can be used to support some level of automated decision making [Correa et al. 2008].

References

Box, D. (1997). *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA. Foreword By-Grady Booch and Foreword By-Charlie Kindel.

Correa, S., Fonseca, E., and Cerqueira, R. (2008). A self-diagnosis approach for performance problem localization in component-based applications. In *Proceedings of NOMS'08*, pages 931–934, Los Alamitos, USA. IEEE Computer Society.

da Silva Santos, L., Ramparany, F., Costa, P., Vink, P., Etter, R., and Broens, T. (2007). A service architecture for context awareness and reaction provisioning. In *Proceedings of Services 2007*, pages 25 – 32, Washington, USA. IEEE Computer Society.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI 2004*, pages 137–150, Berkeley, USA. USENIX Association.

Debusmann, M., Schmid, M., and Kroeger, R. (2002). Measuring end-to-end performance of CORBA applications using a generic instrumentation approach. In *Proceedings of ISCC's 02*, page 181, Washington, USA. IEEE Computer Society.

Diakov, N. K., Batteram, H. J., Zandbelt, H., and Sinderen, M. J. (2000). Design and implementation of a framework for monitoring distributed component interactions. In Scholten, H. and van Sinderen, M., editors, *IDMS*, volume 1905 of *Lecture Notes in Computer Science*. Springer.

Fickas, S. and Feather, M. (1995). Requirements monitoring in dynamic environments. In *Proceedings of 2nd IEEE Int. Symposium on Requirements Engineering*, page 140, Los Alamitos, CA, USA. IEEE Computer Society.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, 36(1):41–50.

Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalha, C., and Campbell, R. H. (2000). Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of Middleware 2000*, volume 1795 of *Lecture Notes in Computer Science*, pages 121–143. Springer.

Marchetti, C. (2001). CORBA request portable interceptors: A performance analysis. In *Proceedings of DOA '01*, page 208, Washington, USA. IEEE Computer Society.

Rackl, G., Lindermeier, M., Rudorfer, M., and Süss, B. (2000). MIMO - an infrastructure for monitoring and managing distributed middleware environments. In Sventek, J. S. and Coulson, G., editors, *Proceedings of Middleware 2000*, volume 1795 of *Lecture Notes in Computer Science*, pages 71–87, New York, USA. Springer.

Schroeder, B. A. (1995). On-line monitoring: A tutorial. *IEEE Computer*, 28(6):72–78.

SCS (2008). SCS: Software Component System. `http://www.tecgraf.puc-rio.br/~scorrea/scs`. [Last accessed, January 2008].

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman, Boston, MA, USA.

Wang, N., Schmidt, D. C., and O'Ryan, C. (2001). *Overview of the CORBA Component Model*, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., Boston, USA.

Wang, Q., Liu, Y., Li, M., and Mei, H. (2007). An online monitoring approach for web services. In *Proceedings of COMPSAC 2007*, pages 335–342, Washington, USA. IEEE Computer Society.

WebSphere (2008). WebSphere Monitoring. `http://manageengine.adventnet.com/products/applications_manager/websphere-monitoring.html`. [Last accessed, June 2008].