MIMO – An Infrastructure for Monitoring and Managing Distributed Middleware Environments*

Günther Rackl, Markus Lindermeier, Michael Rudorfer, and Bernd Süss

LRR-TUM
Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik
Technische Universität München, 80290 München, Germany
rackl@in.tum.de

Abstract. This paper presents the MIMO MIddleware MOnitoring system, an infrastructure for monitoring and managing distributed, heterogeneous middleware environments. MIMO is based on a new multi-layer-monitoring approach for middleware systems, which classifies collected information using several abstraction levels. The key features of MIMO are its openness, flexibility, and extensibility. MIMO's research contribution is to enable easy integration of heterogeneous middleware platforms, to be suited for large classes of online tools covering both monitoring and management functionality, and therefore to be applicable for tools supporting the complete software lifecycle. In addition to the core MIMO system we outline exemplary instrumentation techniques for integrating CORBA and DCOM platforms, and present the MIVIS visualization tool demonstrating the features of the MIMO infrastructure.

1 Introduction and Overview

Developing and maintaining large distributed software environments is one of the major challenges in computer science at the time. The usage of middleware platforms abstracting from diverse and heterogeneous computing platforms is a common approach to handle the complexity of such systems. Middleware platforms include general purpose distributed object-computing environments [1] like CORBA or DCOM, message-oriented middleware (MOM), transaction processing monitors (TPMs), or meta-computing infrastructures like Globus [2].

A main drawback deploying all kinds of these platforms is the lacking support for online tools which allow to monitor and manage the environments, especially when various types of middleware products are combined within one computing environment. Monitoring and management tools should cover the whole software lifecycle, i.e the development and the deployment phases of middleware-based software products.

In the past, several monitoring systems have been developed for specific kinds of middleware products [3,4]. But, most systems are limited to one single type

* Research supported by German Science Foundation (DFG) SFB 342 (TP A1).

J. Sventek and G. Coulson (Eds.): Middleware 2000, LNCS 1795, pp. 71–87, 2000.

of middleware platform, only concentrate on specific aspects of these platforms, and are therefore only suited for a small class of tools.

This paper presents the MIMO MIddleware MOnitor infrastructure, a monitoring and management system that addresses the following issues:

- *Support for the whole software lifecycle:* In order to be able to build tools for the complete software lifecycle, information on all abstraction levels of the system has to be gathered. This includes low-level information, e.g. needed for debugging purposes during software development, as well as high-level information for application management issues during software deployment. MIMO solves this problem by introducing a multi-layer-monitoring model that is used to classify data collected from the observed system.
- *Integration of monitoring and management functionality:* Supporting the complete software lifecycle allows us to use a single system for monitoring and management tasks; as the term "monitoring" is mostly used for low-level aspects, and "management" rather for high-level administrative tasks, the multi-layer-model makes it possible to build both kinds of tools only using MIMO[1].
- *Integration of heterogeneous middleware platforms:* MIMO is designed to enable monitoring of different middleware platforms simultaneously. This is done by introducing a generic interface for middleware platforms to MIMO, such that heterogeneous systems can be easily integrated. As interoperable applications are getting more and more popular (see e.g. CORBA-COM bridges), the ability to observe heterogeneous systems simultaneously is one of the key features for future monitoring systems.

As requirements for monitoring and management tools are very diverse, especially when allowing to observe heterogeneous components simultaneously, there is no common definition of tool functionality or behavior. Therefore, the MIMO approach is based on defining a general infrastructure, i.e. a *framework* [5] for tools. This framework consists of basic monitoring system components which offer defined and generic interfaces both for tools and middleware platforms. Furthermore, access patterns defining how to make use of the components and interfaces are defined. This combination of components and patterns makes it possible to keep MIMO very generic and configurable, and thus allows to use it for very diverse purposes without altering the core MIMO implementation.

This paper is organized as follows: Section 2 introduces the multi-layer-monitoring approach on which MIMO is based. Section 3 presents the core MIMO infrastructure, section 4 explains how information can be gathered from various middleware platforms, and section 5 presents the MIVIS visualization tool, an example for a fundamental tool making use of MIMO. Section 6 outlines an example scenario showing the usage of MIMO and MIVIS, and section 7 finally summarizes and concludes the paper.

[1] In the following, we will not distinguish between the term "monitoring" and "management" anymore; we will mostly use the term "monitoring", which shall cover both classes of systems and tools.

Related Work

Basic work on monitoring systems has be done in the OMIS project [6]. However, OMIS is mainly aimed at lower-level monitoring of parallel applications using a message-passing communication paradigm.

For CORBA, there are several management systems, which are mostly commercial products tied to specific CORBA implementations. These include e.g. ObjectObserver by Black&White Software, CORBA-Assistant by Fraunhofer Institute, or IONA's OrbixManager. An overview of these systems can be found in [4]. The main drawback of all these products is the lacking genericity which allows to deal with heterogeneity in a way MIMO does, because they are mostly tailored to one specific middleware product. Moreover, all of them only allow to monitor the server-side of CORBA applications, client-side activity cannot be observed explicitly.

2 Multi-Layer-Monitoring

This section describes the system model and multi-layer-monitoring approach, on which the MIMO system is based.

2.1 Distributed Middleware Environment Model

Figure 1 shows an illustration of a typical distributed middleware environment that we consider. The system to be monitored consists of six abstraction layers, from which the monitor collects information and provides it to tools only by means of the tool-monitor interface.

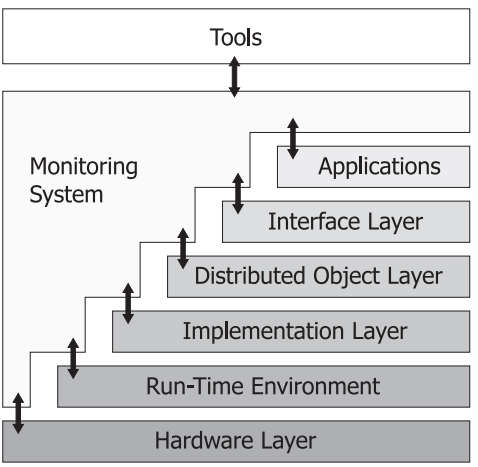

Fig. 1. Layer Model of the Distributed Environment

The highest abstraction level within the system is the application level. Here, only complete applications are of interest for the monitoring system. Within an application, the whole functionality exported by the components is described by interfaces. These interfaces are defined in an abstract way in the interface layer. The implementation of the behavior described by these interfaces is done by objects within the distributed object layer. These objects may still be considered as abstract entities residing in a global object space. In order to enable communication between the distributed objects, some type of middleware is required. Especially, a mechanism to define and uniquely identify objects within the object space is needed. All commonly used middleware standards use some kind of globally unique object references. For example, CORBA uses Interoperable Object References (IORs) to identify CORBA objects [7], Sun's Java Remote Method Invocation RMI uses Uniform Resource Locators (URLs), and Microsoft DCOM [8] generates so-called Globally Unique Identifiers (GUIDs) or monikers. As objects on the distributed object level are still abstract entities, they need to be implemented in a concrete programming language. This implementation of the objects is considered in the subsequent implementation layer. Obviously, objects may be implemented using some O-O language, but also non-O-O languages may be used, e.g. for integrating legacy code. Finally, the implementation objects are executed within a run-time environment which can be an operating system or a virtual machine on top of an operating system that is being executed by the underlying hardware nodes.

For various middleware platforms, this abstract model can be mapped to concrete entity types related to the respective middleware environments like e.g. CORBA or DCOM; Figure 2 shows the CORBA mapping of the MLM, where the main entities of interest are CORBA objects; see [9] for details.

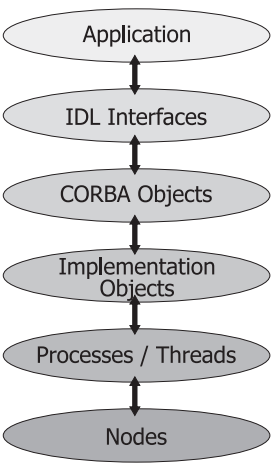

Fig. 2. CORBA Mapping

2.2 Multi-Layer Monitoring

For the monitoring system, two aspects are important: First, it has to be possible to gather data on all abstraction levels in order to serve as an information source for all kinds of online tools. And secondly, the mappings between the different layers are of great importance. As all entities within a specific layer are mapped onto appropriate entities within the layer on the next lower level until the hardware layer is reached, keeping track of these mappings is essential because the relationships between entities in two adjacent layers are not necessarily one-to-one relationships.

Tools making use of the monitoring system may be very diverse and therefore operate only on specific abstraction levels (e.g. a visualizer might be interested in interfaces and CORBA objects). For other tools, mappings between layers can be of special interest (e.g. for performance analysis, the process distribution on the nodes can be decisive).

As a consequence, a *multi-layer monitoring* (MLM) approach [9] which closely reflects the structure of distributed object-environment is well suited for a large class of online tools. For obtaining information from all abstraction layers, specialized modules adapted to requirements of the layer to be observed can be inserted into the monitoring system. Thus, the monitor is kept very modular and flexible and can easily be adjusted to changes of the distributed environment.

3 MIMO

This section introduces the principle design of MIMO's components and interfaces. An important issue for the overall design of MIMO was the genericity of the approach; this means that MIMO is kept open to integrate various types of middleware platforms, and to make it suitable for building any kind of tool. As requirements for different middleware and tools can be very diverse, MIMO itself is designed to depend as little as possible on concrete implementations and semantics of events. Hence, only little information about common entities within the environment is stored by MIMO, and flexibility is gained by tools and intruders being adjusted to each other.

MIMO itself is completely implemented in Java (Java 2 platform), making use of the ORBacus 3.2 [10] CORBA implementation.

3.1 Monitoring and Management Scenario

The MIMO MIddleware MOnitor provides a framework for online monitoring and management tools which is compliant to the multi-layer-monitoring approach. The fundamental architecture relies on the separation of the tools from the monitoring system and the observed applications [6]. Figure 3 illustrates the resulting 3-tier model, which shows tools making use of MIMO by means of a tool-monitor-interface, while MIMO collects information from the monitored applications by means of intruders or adapters which communicate with

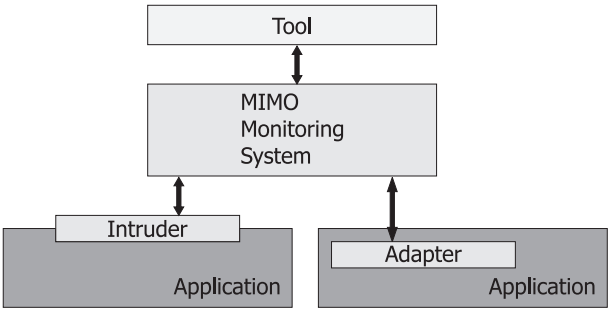

Fig. 3. 3-Tier Model of the Monitoring Architecture

MIMO through a intruder-monitor-interface. The difference between intruders and adapters is that intruders are transparently integrated into the application (without rebuilding the application), while adapters might be built by inserting code into the application (and rebuilding it).

An important aspect in this context is that MIMO makes it possible to monitor both the client- and server-side of distributed applications. Most of the existing management tools are limited to server-side monitoring and administration; MIMO's approach in contrast is layed out for client-side instrumentation too, which in most cases is implemented by proxy-instrumentation techniques.

Finding and Accessing MIMO Communication with MIMO is exclusively handled by CORBA communication. So, when tools or intruders/adapters are being started, they first need to get an IOR for the respective MIMO interfaces in order to be able to communicate with MIMO. Therefore, every running MIMO instance publishes its IOR at a CORBA naming service whose IOR is being stored at a fixed URL which the clients need to access via http; this URL is kept constant, so that clients can even find MIMO and the appropriate naming service when they get restarted with different IORs over time. As multiple instances of MIMO might be running at the same time, every registration at the naming service includes the hostname on which the MIMO instance is running; thus, client (tools, intruders/adapters) can easily choose a local MIMO instance, if it exists.

3.2 MIMO Architecture

An illustration of the basic MIMO architecture is shown in Figure 4. Every instance of MIMO keeps information about the current system state, i.e. data about the applications currently attached to this instance via the intruders/adapters. Furthermore, information about currently attached tools and their active requests is stored. Information can also be exchanged between various MIMO instances, but it is only stored once at the MIMO instance whose intruder/adapter provided the data.

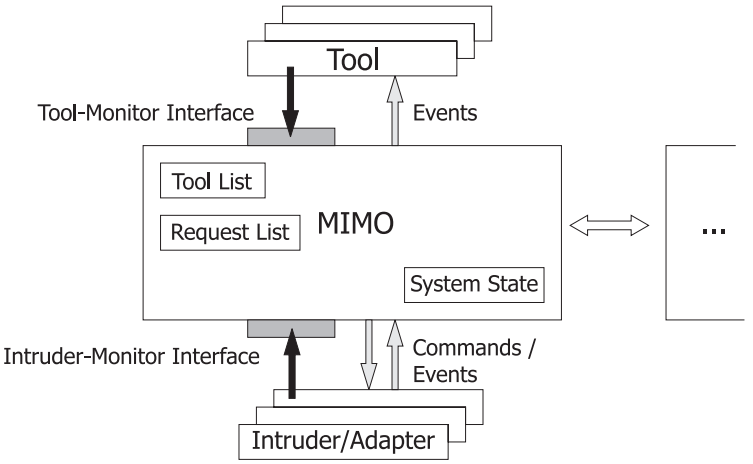

Fig. 4. MIMO Communication: Interfaces and Event Channels

Tool-Monitor-Interaction The only entrance point for tools to MIMO is the tool-monitor-interface, which is a CORBA IDL interface that basically provides methods for attaching to and detaching from MIMO, and for starting and stopping requests.

When requests are started, the result can either be returned synchronously if this is possible (e.g. for system state queries), or in an event-based manner, which is necessary for passing results of asynchronously occurring events (e.g. interactions between entities). Events are passed from MIMO to the tool through a CORBA event channel which is set up during the attachment of the tool. For example, a tool might issue the simple command

$$request(\text{"}get\_objects\text{"}, < appl1, appl2 >, objList),$$

which provides a list of all objects belonging to applications `appl1` and `appl2` as an out parameter in `objList`. Or, as an asynchronous request, the tool might get notified whenever `obj2` makes use of the interface `ifc1` with the request

$$start\_request(tid, \text{"}get\_interactions\text{"}, < ifc1, obj2 >),$$

where `tid` is the tool-identifier which is needed to select the corresponding event channel for passing the interaction-events from MIMO to the tool.

Intruder/Adapter-Monitor-Interaction The entrance point for adapters and intruders is the intruder-monitor-interface, which provides methods for attaching and detaching intruders/adapters. After initialization, communication between MIMO and the clients is only handled via two CORBA event channels which are set up at startup. Event channels are mandatory because an asynchronous way of interaction is needed in order to influence the observed system

as little as possible. Whenever an "interesting" event occurs within the monitored application, the intruder builds a CORBA event and passes it to MIMO via the monitor-intruder event channel. Similarly, whenever MIMO needs to pass information to the intruder, e.g. for configuring the intruder, it passes a CORBA event to the intruder via its intruder-monitor event channel. This way of communication results in a decoupled intruder/adapter-MIMO interaction scheme. Moreover, it is very flexible due to the standardized protocol which allows for easy integration of different types of middleware platforms which need to be attached by very different intruder/adapter implementations.

4 Instrumentation of CORBA and DCOM Platforms

Enabling different middleware platforms to be observed by MIMO needs some kind of instrumentation to get information out of the respective applications. Instrumentation techniques can be very diverse, and consequently no general approach that is suitable for all kinds of middleware products can be given, but some basic mechanisms are shown here. Nevertheless, MIMO is kept open and allows for using other instrumentation techniques, whenever they provide information in the standardized way through the interfaces and event channels.

This section outlines two exemplary approaches to connect CORBA or DCOM applications to MIMO; however, the general techniques can easily be transferred to other similar middleware environments. Fundamental data to be collected from CORBA and DCOM applications contain information about all existing instances of distributed objects (i.e. CORBA or DCOM objects) within the system, and their interactions (i.e. method calls to such objects). The following examples concentrate on gathering these data.

4.1 Instrumenting CORBA Applications

As mentioned above, data collection in MIMO can either be done by an adapter or an intruder. Here, we outline both approaches for instrumenting CORBA applications.

CORBA Adapter The CORBA adapter basically consists of a Java class providing a library of methods for communicating with MIMO. This includes functions for attaching and detaching to/from MIMO easily, and for sending and receiving CORBA events to and from MIMO. Events can be any kind of information sent to MIMO, but for common tasks the following predefined functions exist:

- Object creation and deletion
- Interaction between objects
- Any other calls to CORBA middleware functions

These event types can easily be generated from within the application code by calling the MIMO adapter functions. Adapters can be useful when the source code is available, and when knowledge about the application domain can be used to instrument the application manually in a way to get specifically interesting events.

CORBA Intruder When the application cannot be rebuild, or instrumentation needs to be inserted transparently, the CORBA intruder can be used. It is based on the instrumentation of the used CORBA library (in our case the ORBacus C++ library). With this technique, *wrapper functions* for the original CORBA methods are created and inserted into the library. The original functions are renamed and called by the MIMO wrappers. The approach is implemented by using symbol replacement inside the CORBA library.

The problem with this approach is to find the appropriate CORBA methods which need to be wrapped in order to get the required information. For our CORBA intruder, the idea is as follows:

- For startup purposes, CORBA initialization functions like `ORB::init` need to be wrapped for enabling the attachment to MIMO.
- To get information about newly created or deleted objects, keeping track of the reference counter functions (duplicate and release) is a convenient way; when the reference counter reaches zero, the CORBA object gets deleted. Furthermore, observing the creation of client-side proxies is also possible by looking at the `string_to_object` operation which instantiates a proxy for a given CORBA object.
- Interactions between objects finally result in a call to a CORBA request's `invoke` (or related `send_oneway` and `send_deferred`) method. Thus, wrapping this method allows us to observe all method calls to any CORBA objects.
- Any other CORBA method call can be instrumented, if given circumstances need to access it.

Hence, this proceeding enables to monitor different aspects of CORBA systems. The advantage is that information can be gathered at different *levels of detail*, depending on the granularity required by a given tool.

Performance data evaluating the overhead introduced by the CORBA intruder will be available for a final version of the paper.

4.2 Instrumenting DCOM Applications

In DCOM, no direct way exists to get information about method calls to DCOM objects. While there is no difference implementing objects for in-process, out-of-process or remote access, there is a big difference in how they are called. Out-of-process and remote calls base on the RPC protocol and a pair of proxy and stub, in-process calls are direct procedure calls without any participation of the COM library. Gathering information about all kinds of COM calls requires other mechanisms than instrumenting the COM library.

DCOM Wrapper The best way to achieve this goal is to use a wrapper for each monitored object. This provides scalability because only calls of interest are recorded. The wrapper has to provide hooks for requests orthogonal to the method call. These requests could not only be auditing requests but also security checks etc[2]. The approach applied in MIMO is based on a universal delegator object [12,13], and trace hooks [14]. To work properly, the wrapper has to be called instead of the original object, such that it can process the call first. Therefore, the registry is manipulated to set up a special class factory for the monitored object. This class factory first creates the monitored object using the original class factory, then creates and initializes the wrapper with the object and gives back a pointer to the wrapper. Once this is set up, the wrapper analyzes the call stack every time a method call is received. Then it checks whether only to preprocess the call or to pre- and postprocess it. It sends the required information to the hook and forwards the call to the original object after changing the return address to itself. The wrapper and the hook themselves are in-process objects tied to the original object's thread (Figure 5).

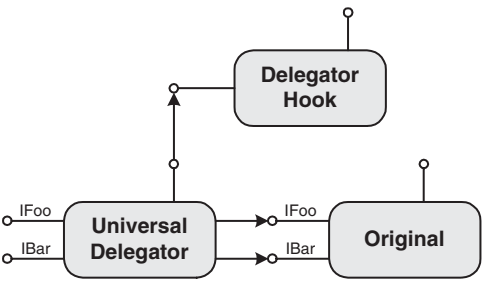

Fig. 5. DCOM Delegator

To work with MIMO, some additional work is required. Therefore, a universal framework was designed which supports any kinds of information sources and any kinds of information processors. One component is a CORBA-COM bridge-object, which provides the interface to MIMO. The main information source is the combination of the wrapper (Universal Delegator) and the hook (TraceHook). Other sources could be objects reading the event log or performance data. The overall scenario is showh in Figure 6.

Performance and Limitations Performance tests have been carried out to get an evaluation of the overhead of this solution. As the wrapper approach implies a process switch during the call, it is only applicable for out-of-process or remote calls, and not for in-process calls (as the overhead in this case would

[2] In COM+, which ships with Windows2000, such a wrapper will be integrated into the COM+ event service [11].

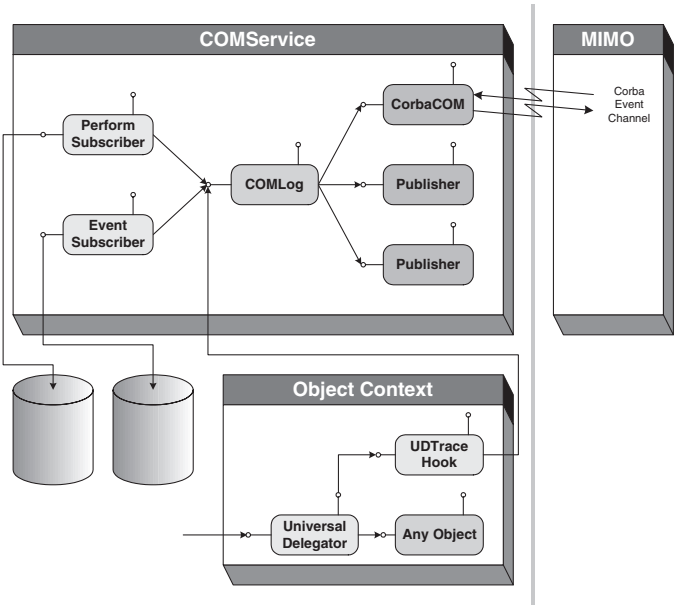

Fig. 6. Overview of the DCOM Instrumentation Approach

be tremendous). The measured overhead of the DCOM wrapper framework was 2.9 for out-of-process (but still local) calls, and 1.6 for remote calls. These values show that it might not be useful to collect all available data, but to build a more "intelligent" wrapper which only gathers information on request; this issue is an implementation problem which will be solved in a future version of the wrapper. More details can be found in [15].

5 MIVIS Visualization Tool

Here we describe the visualization tool MIVIS (MImo VISualizer). Our goal was to develop a visualization tool that is based upon the multi-layer-monitoring concept described earlier. It interacts with MIMO and presents data it receives in an advantageous way to the user. Amongst the requirements for MIVIS were scalability, uncomplicated extensibility, platform independence, an ergonomic user interface, and the possibility of having several displays at a time.

A general problem of visualization is scalability: Huge amount of data have to be presented in a way which allow the observer to keep track of the information offered. Thus, there has to be the possibility to reduce data by means of filtering mechanisms. MIVIS realizes this reduction by its selection mechanism which provides a kind of filtering based on the multi-layer-monitoring model.

5.1 MIVIS Concepts

All entities in the monitored application are shown inside the selection frame (see Figure 7). Each layer of the multi-layer-monitoring model is represented in one

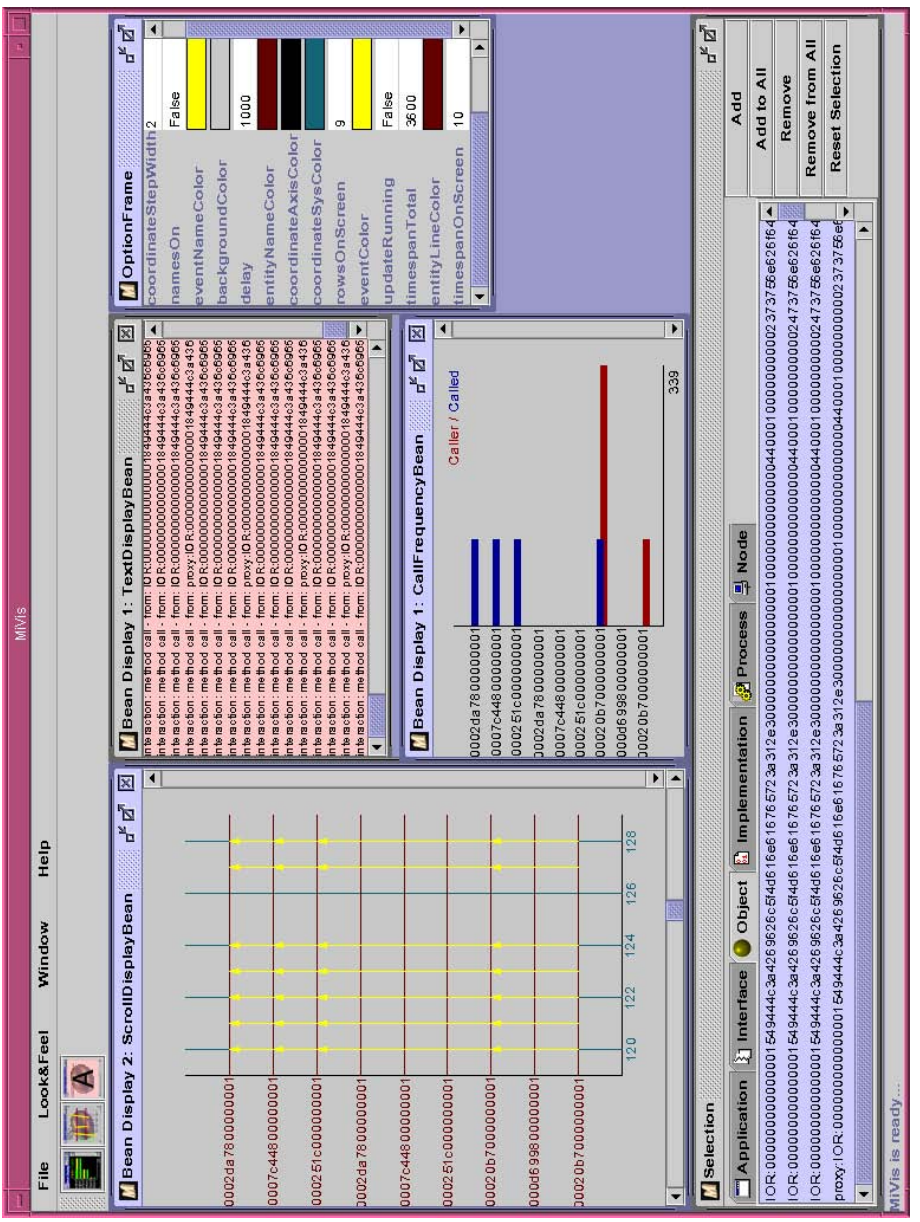

Fig. 7. MIVIS Displays

tab of a tabbed pane. The user can select entities within the different layers and thus control the granularity of his visualization. To gain an overall survey she can monitor the system on the application layer or the hardware layer without being bothered by details. To get more insight in the internals of the application she can pick out a few interesting entities and go up or down to adjacent layers to get more detailed information;

5.2 Implementation

To fulfill the requirement of uncomplicated extensibility of the visualization tool, it is split into a main program and several JavaBeans software components. The main program takes care of the communication with MIMO and the processing of the data, and the JavaBeans do the graphical display.

All JavaBeans are discovered by MIVIS at startup time, and get dynamically integrated into the GUI. If a different type of display is needed, a user can program that display type using Java and turn it into a JavaBean. This component is placed into a specific directory so that MIVIS can find and use it. The main program does not have to be changed at all, the only requirement is that the JavaBean implements a minimal interface that enables the main program to communicate with the bean.

The bean-specific properties can be set by the user. MIVIS knows about these properties by means of the introspection mechanism and provides editors to change the settings of these properties. Additional editors for properties of a special data type can be placed inside the JavaBean and used instead of the standard editors. All properties together with their editors are shown inside the Option Frame (see Figure 7). Hence, MIVIS allows the user to edit properties which the tool itself does not know from the beginning. This approach offers a very dynamic and flexible way to configure the behavior of various display types; the concept of separating the display types from the main program makes it very easy to generate new display types for MIVIS without the need of changing the original code.

5.3 MIVIS Displays

So far, three display types have been implemented: text display, scroll display and call frequency display. These three displays types can be seen in Figure 7.

- The text display prints out the events that are monitored in plain text, what can basically be used for logging purposes; details that might not be visible in a graphical display can be looked up here at a later time.
- The scroll display visualizes communication between entities. The selected entities are displayed on the y-axis in a coordinate system. The x-axis shows the time. When an entity communicates with another entity, an arrow between the two is shown in the coordinate system.
- The call frequency display visualizes communication in a different way: Only cumulative data containing the number of calls are of interest are shown as a vertical beam for each caller and for each called entity.

These displays are only fundamental aspects of an application that might be of interest, but others can easily be added by programming new JavaBeans.

In this sense, MIVIS can be seen as a general *framework for GUI-based MIMO-tools* which provides the basic monitoring functionality, and can be extended with additional JavaBeans to fulfill any further monitoring requirements.

More details about MIVIS can be found in [16].

6 Example Scenario

To test MIVIS in a real-world scenario we picked a simple library application, which represents a 3-tier client-server application with distributed data.

- The first tier consist of clients that can do various operations, such as searching for books, inserting new book into the library etc; these clients can be located on different machines.
- The second tier keeps the client interfaces which provide the business logic of the application; they basically process the client requests, make database queries to the third layer, assemble the results, and pass them back to the clients. Client interfaces can also be located on several machines.
- The third tier contains library managers, which contain the actual library databases. Different managers store information about different books. Again, the library managers can be distributed over a set of nodes.

When a client starts a request, for example a search for a certain book, the following things occur: The client selects a free client interface at random to process the request. The client interface contacts all library managers that are known to it and requests the information about the book. It waits for the answers of the library managers and combines all answers to the final result which is sent back to the client.

In our test scenario, clients periodically start search requests. Each call from a client to a client interface is followed by several calls from the selected client manager to all library managers. In our example we have three library managers, so there are three calls from the client interface for each client request. In Figure 7, every call is displayed as an arrow from the caller to the called entity. As the time between the calls is very short, all arrows for one request seem to be one line in this coarse illustration. In Figure 8, we can see that the vertical beam for the calls from the client interface is three times the size of the one for the calls from the client and for each library manager (because for each client call, the client interface invokes every library manager).

Hence, his example briefly demonstrates how to use MIVIS to visualize the behavior of distributed middleware applications. Clearly, in practice more sophisticated scenarios have to be analyzed, but the general approach to investigate request sequences in n-tier client-server applications is a very important and helpful feature, either for debugging, performance analysis, or management purposes.

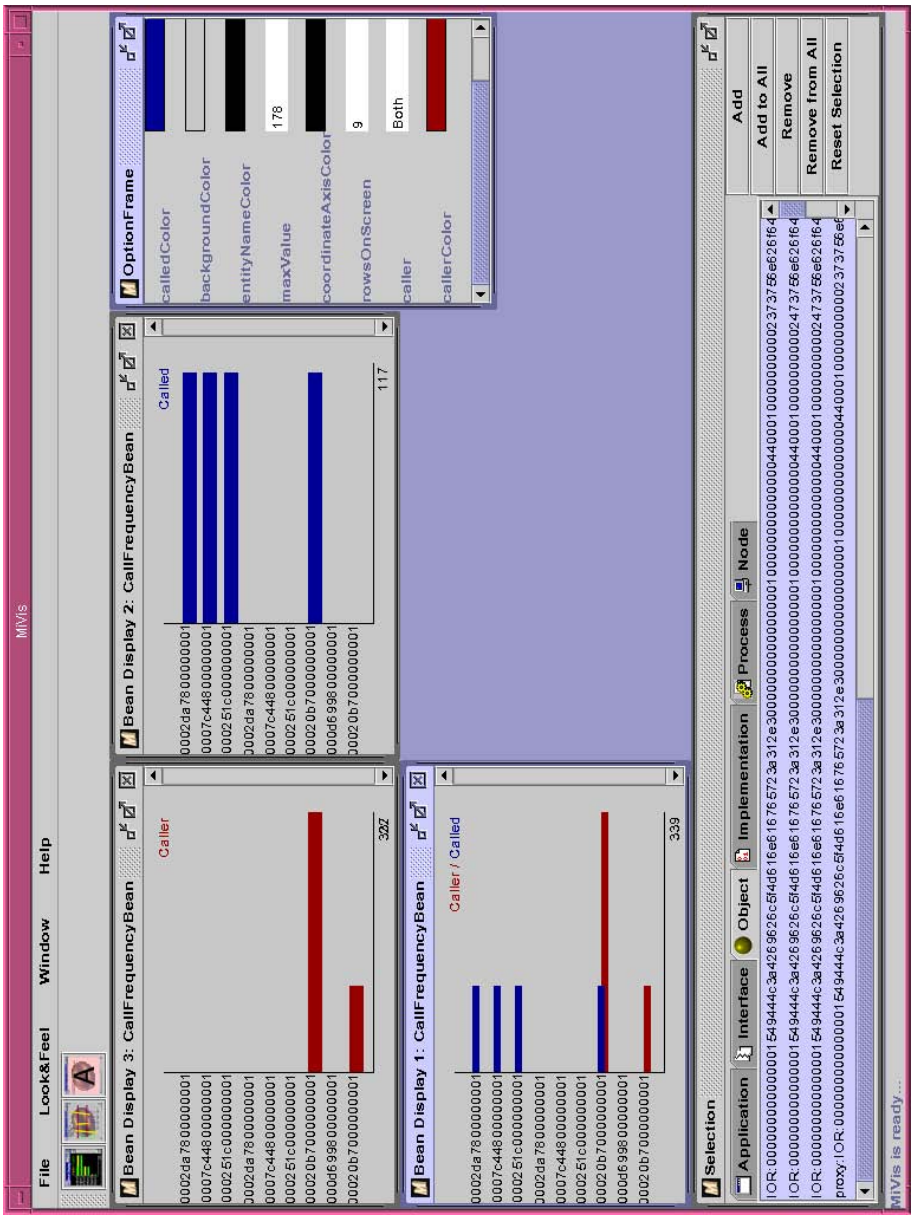

Fig. 8. MIVIS 3-tier Library Example

7 Conclusion

In this paper, we have presented the MIMO infrastructure for monitoring and managing distributed middleware environments, and the MIVIS visualization

tool demonstrating basic MIMO functionality, and which serves as a framework for further tool extensions.

MIMO is based on a new multi-layer-monitoring approach for middleware systems which allows us to handle complex middleware systems on several abstraction levels. This provides the possibility to build online tools supporting the complete software lifecycle while integrating monitoring and management functionality. The integration of different middleware platforms is reached by introducing a standardized intruder-monitor-interface. To our knowledge, no other monitoring infrastructure reaching this high degree of flexibility over several dimensions has been developed up to now.

What still needs to be completed is the distribution of MIMO itself (including synchronization and event ordering issues, similar to OCM project [17]), and the capability to dynamically insert code into intruders in order to reach an even higher flexibility.

The main research contribution is motivated by the fact that common middleware environments and tool requirements are too diverse to be handled by a single, static monitoring system. Instead, we propose and implement an open and flexible monitoring and management infrastructure, which only provides basic monitoring services, but is open to be extended easily.

References

1. Günther Rackl, Ivan Zoraja, and Arndt Bode. Distributed Object Computing: Principles and Trends. In *International Conference on Software in Telecommunications and Computer Networks – SoftCOM '99*, pages 121–132, Oct 1999. 71
2. I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18. IEEE Computer Society Press, 1998. 71
3. Ivan Zoraja, Günther Rackl, and Thomas Ludwig. Towards Monitoring in Parallel and Distributed Environments. In *International Conference on Software in Telecommunications and Computer Networks – SoftCOM '99*, pages 133–141, Oct 1999. 71
4. Bernfried Widmer and Wolfgang Lugmayr. A Comparison of three CORBA Management Tools. In Wolfgang Emmerich and Volker Gruhn, editors, *Engineering Distributed Objects (EDO'99)*, pages 12–21, Los Angeles, May 1999. 71, 73
5. Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, Oct 1997. 72
6. Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, and Arndt Bode. *OMIS — On-Line Monitoring Interface Specification (Version 2.0)*, volume 9 of *Research Report Series, Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM), Technische Universität München*. Shaker, Aachen, 1997. 73, 75
7. OMG (Object Management Group). The Common Object Request Broker: Architecture and Specification — Revision 2.2. Technical report, February 1998. 74
8. Microsoft Corporation. DCOM Architecture. Technical report, 1998. 74
9. Günther Rackl. Multi-Layer Monitoring in Distributed Object-Environments. In Lea Kutvonen, Hartmut König, and Martti Tienari, editors, *Distributed Applications and Interoperable Systems II — IFIP TC 6 WG 6.1 Second International Working Conference on Distributed Applications and Interoperable Systems*

(DAIS'99), pages 265–270, Helsinki, June 1999. Kluwer Academic Publishers. 74, 75
10. Object Oriented Concepts Inc. ORBacus, Nov 1999. http://www.ooc.com/ob/. 75
11. David S. Platt. *Understanding COM+*. Microsoft Press, 1999. 80
12. Keith Brown. Building a Lightweight COM Interception Framework, Part 1: The Universal Delegator. *Microsoft Systems Journal*, Jan 1999. 80
13. Keith Brown. Building a Lightweight COM Interception Framework Part 2: The Guts of the UD. *Microsoft Systems Journal*, Feb 1999. 80
14. Simon Fell. Activation tricks. WWW, July 1999. http://www.zaks.demon.co.uk/com/activation.htm. 80
15. Bernd Süss. Konzepte und Mechanismen zum on-line Monitoring von DCOM-Anwendungen. Diploma thesis, Technische Universität München, 1999. In german. 81
16. Michael Rudorfer. Visualisierung des dynamischen Verhaltens verteilter objektorientierter Anwendungen. Diploma thesis, Technische Universität München, 1999. In german. 84
17. Roland Wismüller, Jörg Trinitis, and Thomas Ludwig. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*. ACM Press, 1998. 86