An Open Monitoring System for Parallel and Distributed Programs

Thomas Ludwig, Michael Oberhuber, Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR–TUM)
Institut für Informatik der Technischen Universität München
D-80290 München, Germany
email:{ludwig|oberhube|wismuell}@informatik.tu-muenchen.de

Abstract. The on-line monitoring interface specification OMIS provides means for developing more powerful tool environments for parallel and distributed systems. It specifies the interaction between any tool and a monitoring system which is responsible for observing and manipulating the programs' execution. By having this well defined interface it is now possible to concurrently use several tools of possibly different developers with the same program run. The research group at LRR-TUM currently designs an OMIS compliant monitoring system for PVM running on workstation clusters. Tool developers can use this implementation to attach their own on-line tools to the system.

1 Motivation

The available support for parallel programming in environments with distributed memory varies considerably in quality and quantity. Results of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments [1] show that main problems are a lack of sophisticated tools and the almost complete absence of any interoperability of tools of different origin. Also there are no uniform environments at all, where the developer could use the same tools for different types of hardware or parallel programming libraries.

Tools which support an investigation of the running parallel program can be divided into on-line and off-line tools. In case of off-line tools, the task of a monitoring system is restricted to only collecting information about the system behavior and forwarding it to a file for storage. Since we do not know in advance which information the user wants to evaluate, enormous amounts of data have to be recorded, thus increasing the probability for a *probe effect*. With on-line tools the user evaluates this data immediately, which requires additional capabilities to be added to the monitor: programmability and adaptability.

Caused by its intermediate position, a monitoring system has two interfaces: one to the tools and one to the running program. Up to now, there are no approaches undertaken to standardize at least the tool/monitor-interface. Without a reasonable standard, however, new tools must also have new monitoring systems even if their functionality is not completely disjunct to already existing tools. Finally, the missing standard makes the integral use of a set of tools impossible as they are currently always based on differently implemented monitoring concepts which are incompatible to each other.

Having such an agreed on on-line monitoring interface, the amount of effort for having n tools on m systems (being composed of hardware, operating system, and runtime library of the programming environment) would be reduced from $n \times m$ to $n + m$, thus bringing more tools onto more parallel systems which finally would ease software development on these architectures. A further implication of having OMIS compliant monitoring systems is that finally we will reach the goal of having uniform environments, i.e. tools which are identical for a variety of target architectures.

The paper will describe the approach followed by OMIS. It will give an outline of the concepts which are incorporated into the specification and its influences on the design of a compliant monitoring system for PVM.

2 Related Work

Although methods and strategies for monitoring parallel and distributed systems seem to converge, no standard interface has yet been developed. This is especially true for the area of heterogeneous on-line monitoring. The reasons are additional demands like extendibility, programmability and asynchronous behavior. Several articles about monitoring methodologies have been published ([2], [3], [4]), but no comprehensive approach summing up state-of-the-art techniques to a common on-line monitoring system has been proposed. Comparable efforts are undertaken by the Ptools Consortium related to other areas of tool development.

Already in 1991 B. Bruegge was proposing a heterogeneous on-line monitor, called BEE (basis for distributed event environments)[2], albeit its capabilities are restricted to observe events. This constraint is also true for Xab [5]. Independence of the program environment and heterogeneity is proposed in the monitoring systems of the parallel debugger p2d2 [6], which is supposed to work with MPI as well as with PVM. But the integral use of different tools that are served by a common on-line monitor was realized a few times only [3, 4, 7] with the restriction to debuggers and performance analyzers.

The event–action paradigm is a common way to monitor parallel programs. It convinces through its intuitive use and simplicity, as shown in [8], but it lacks efficiency. There are two feasible ways to reduce the overhead without auxiliary hardware, filtering and programmability. But for an all purpose monitoring system it is not enough to recognize events and trigger actions. For example, Performance analysis requires a time driven monitoring to sample the current position of an application. Therefore, Petrenko is talking about *passive* and *active* monitoring systems [9], i.e. the monitor not does not only react, but it is acting itself.

Regarding event based monitoring systems and attempts to deal with heterogeneity the client/server approach has proven to be a feasible model for the architecture of a monitoring system [2, 10, 6]. To decouple tools and monitoring systems results in two major advantages, high portability and reusability.

The standard for on-line monitoring systems we want to propose in this article should involve the essence of the discussion above. OMIS is specified with respect to the event–action paradigm and the effort to minimize the overhead of event processing. Since a monitoring system is embedded between the tools and the object of observation it should have both, properties of a client and a server.

3 Requirements to a Monitoring Interface

The design of the monitoring interface imposes several requirements which the specification will have to meet. These requirements can be divided into three categories: (1) functional requirements, (2) conceptional requirements, and (3) efficiency requirements. We will give a short overview on major issues within these categories.

Functional requirements can be summarized as follows: the monitoring interface should be versatile enough to allow all possible tools to observe and manipulate all objects of the running program (e.g. processes, messages, variables etc.) and to watch hardware objects like the amount of available memory. In order to achieve the desired degree of versatility the monitoring interface should support the composition of more complex service requests from the basic ones.

As we neither know the complete functionality of the tools in advance nor the types of tools themselves we have to require that the monitoring interface satisfies the conceptional requirement of extendibility for future developments.

Finally, we have efficiency requirements. In order to keep the communication overhead minimal it is necessary to have powerful basic services and a possibility of composing service requests into a single request. Furthermore, the monitoring system should also be able to handle certain kinds of events occurring in the application program without interacting with the tool.

4 The Interface Structure

4.1 The System Model

An abstract view of the system's architecture on one node is presented in figure 1. The application programs consist of a certain number of tasks which communicate by mechanisms provided by the parallel programming environment. The part which joins the tools to the running program is just the monitoring system. Its role is to establish the tool/program-interaction.

In contrast to usual interactive tools, like tool A, there may be some tools that are spread all over the target nodes, like e.g. a load balancer. We will call them distributed tools, like tool B. Regarding the middle layer we see two types of extensions (ME and DTE) that will be described completely in section 4.3.

How do the individual layers of figure 1 inter-operate? The monitoring system has two interfaces: one for interaction with the different tools (tool/monitor-interface), a second one for interaction with the program and all underlying layers which keep the program running. For simplicity reasons we will call the latter the monitor/program-interface.

The subject of OMIS is the specification of semantics for the on-line monitoring interface and the design of means to extend it. The tool/monitor-interface is the therefore the union of the on-line monitoring interface and possible extensions.

4.2 Services

Conceptually the tool/monitor-interface consists of only a single procedure that can be invoked by both centralized tools and distributed tools. This procedure receives a string

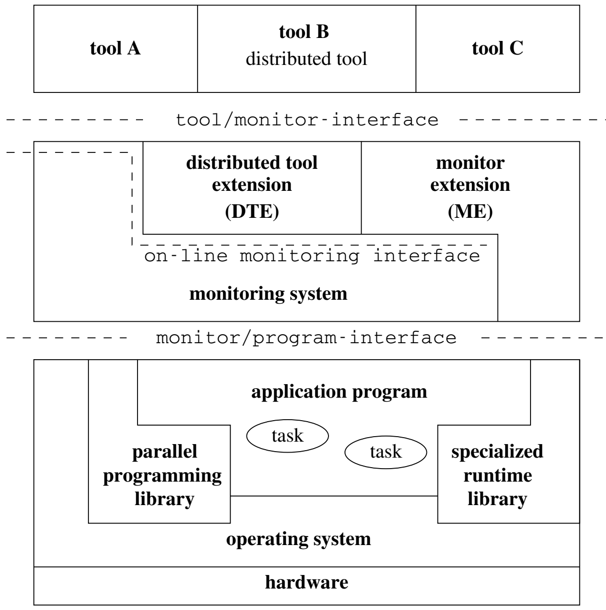

Fig. 1. System model: embedding an OMIS compliant monitoring system into an environment with tools and a parallel programming library

as an input parameter, interprets this string, and returns a result. The tool may either wait for this result from the monitoring system or may specify a call-back function that is invoked whenever the monitoring system returns a reply. The individual monitoring functions available by invoking the interface are called *services*; the string that is passed to the procedure (requesting the activation of a service) is called *service request*.

In order to meet the efficiency requirements the structure of a service request follows the event-action-paradigm, allowing the monitoring system to quickly react on state changes in the monitored system without having to communicate with the tool. If no event definition is present, the service request is unconditional and all actions are invoked immediately. The monitoring system automatically takes care of forwarding the requests to the proper nodes. In addition, we allow services that are global, i.e. that involve more than one monitor.

We classify services according to their input/output behavior:

1. **Manipulation services**. These services receive some input parameters from the caller, but will not have a result, e.g " stop processes".
2. **Synchronous services**. These services receive some input parameters and immediately return a result exactly once, e.g. "show all tasks on node 0".
3. **Asynchronous services**. These services can have an arbitrary number of replies. Usually, neither the number of replies nor the time of the replies is known. This happens in case of an action that is triggered by an event.

4.3 Extensions

It is of utmost importance to provide a means of extending the interface. The additional services can be implemented as a library of C or C++ functions. There are three situations where linking additional code to the basic monitoring system is profitable :

Monitor Extension (ME): a tool may want to observe new objects within an application, e.g. those of a new runtime library.

Distributed Tool Extension (DTE): Usually, different tools use very different methods to process events. Therefore, it is desirable to define new services for this kind of tool-specific data processing.

Distributed Tool Component (DTC): Finally, there are also fully distributed tools without a central user interface component, e.g. a load balancer. These tools should be linked to the monitoring system for efficiency reasons.

4.4 Examples

In this section we will present a short example that will show how to use the monitoring interface in case of a performance analysis system.

Assume that a performance analysis tool wants to measure the time spent by task 4 (located on node 1) in the **pvm_send** call. In addition, the tool may want to know the total amount of data sent by this task, and it may want to store a trace of all barrier events. Then it may send the following service requests in form of strings to the monitoring system:

```
10 [1]StartLibcall([4],"pvm_send"): \
                    11 [1]StartIntegrator(1),\
                    12 [1]AddCounter(2,$4)
13 [1] EndLibcall([4],"pvm_send"):\
                    14 [1]stop_integrator(1)
15 [] StartLibcall([],"pvm_barrier"):\
                    16 [$0]Trace("barrier0",$0,$1,$2,$3)
17 [] EndLibcall([],"pvm_barrier"):\
                    18 [$0]Trace("barrier1",$0,$1,$2)
19 [1] Enable(10),20 [1]Enable(13),\
                    21 []Enable(15),22 []Enable(17)
```

Each request (except the last one) is composed of an event specification followed by a colon and an action specification. Several actions can be specified in one request, but only one event is allowed. Each individual specification is composed of an identifier (provided by the tool), a list of node numbers in square brackets to which the request refers, and the actual name of the service to be invoked followed by a list of parameters. Event specifications imply a list of well defined output parameters which can be referred to in the action list by using a \$-notation.

The numbers 10 ... 22 are the request identifiers that will be included in the monitoring system's replies. However, in this example there will be no replies, except on error conditions, because only manipulation services are used as actions.

5 Current and Future Work

OMIS version 1.0 [11]is now available for the research community since February 1996. We are in the progress of discussing specific aspects of OMIS with groups being interested in using and extending OMIS.

We expect to have a detailed specification of the compliant monitoring system by middle of this year and will then start with an implementation. Currently with the implementation of the monitoring system we will adapt existing tools of LRR-TUM to PVM.

References

1. T. Sterling, P. Messina, and J. Pool. Findings of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments. Technical report, Center of Excellence in Space Data and Information Sciences, NASA Goddard Space Flight Center, Greenbelt, Maryland, 1995.
2. Bernd Bruegge. A Portable Platform for Distributed Event Environments. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 26 of *ACM SIGPLAN Notices*, pages 184–193, December 1991.
3. James E. Lumpp, Howard Jay Siegel, and Dan C. Marinescu. Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483, Paris, Mai 1990.
4. C. Clemencon, J. Fritscher, and R. Rühl. Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Tool. Technical Report CSCS TR-94-09, CSCS, Manno, 1994.
5. Adam Louis Beguelin. Xab: A Tool for Monitoring PVM Programs. In *Workshop on Heterogeneous Processing*, pages 92–97, Los Alamos, April 1993.
6. Doreen Cheng and Robert Hood. A Portable Debugger for Parallel and Distributed Programs. In *Proc. of Supercomputing'94*, pages 723–732. IEEE, November 1994.
7. O. Hansen, J. Krammer, M. Oberhuber, and R. Wismüller. A Scalable Tool Environment for Observing the Runtime Behavior of Massively Parallel Applications. *To appear in Parallel Computing*, 1996.
8. Dan C. Marinescu, James E. Lumpp, Thomas L. Casavant, and Howard J. Siegel. Models for Monitoring and Debugging Tools for Parallel and Distributed Software. *Journal of Parallel and Distributed Computing*, 9:171–182, 1990.
9. A. K. Petrenko. Methods for debugging and monitoring parallel programs: A survey. *Programming and Computer Software*, 20(3):113–129, may–june 1994.
10. Suresh K. Damodaran-Kamal and Jeffrey S. Brown. Towards Heterogeneous Distributed Debugging. Technical Report LA-UR-95-906, Los Alamos National Laboratory, 1995.
11. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification (Version 1.0). Technical Report TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Munich, Germany, February 1996.

This article was processed using the LaTeX macro package with LLNCS style