OCM — A Monitoring System for PVM

Thomas Ludwig, Roland Wismüller, and Michael Oberhuber

Technische Universität München (TUM), Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Arcisstr. 21, D-80333 München
email: {ludwig|wismuell|oberhube}@informatik.tu-muenchen.de
http://wwwbode.informatik.tu-muenchen.de/

Abstract

The OMIS project aims at defining a standard interface between tools for parallel systems and monitoring systems. A standardized interface will allow different research groups to develop tools which can be used concurrently with the same program. OCM, an OMIS compliant monitoring system, is the first realization of such an environment. It is designed for PVM programs running on workstation clusters. The paper will give an outline of the goals of this project and describe important details of the monitoring system's design.

1 Introduction

The situation in the field of parallel and distributed computing is still characterized by the fact that productivity with implementing programs is much lower than with sequential programs. A reason can be seen in additional complexities like nondeterminism, race conditions, deadlocks etc. Furthermore, there are only few adequate tools available to handle these complexities. Even the most basic problem of parallel code debugging is not yet well supported.

If we take a closer look at available development tools, we find that many of them belong to the category of trace based off-line tools. They provide means to inspect a programs behavior after its completion. Thus, no manipulation of the running code is possible nor can the tool user interact in an adaptive way with his program. For e.g. debugging or program steering however, it is inevitable to have powerful on-line tools at hand. In more detail, tools should be capable of doing both to observe as well as to manipulate the running program.

This requires to have powerful monitoring systems which act as mediators between the tool and the software complex of application program, programming library, and operating system components. The monitoring system is responsible for accepting commands from the tool and providing it with corresponding results. In addition, it controls the program by instrumenting parts of the above mentioned software components.

Up to now, most of the approaches for monitoring systems are custom built, i.e. they are designed for dedicated purposes and tied to specific architectures or programming environments. Several works about monitoring methodologies were published ([1], [10], [11], [13], [2]), but no comprehensive approach summing up state-of-the-art techniques to a common on-line monitoring system has been proposed.

In order to avoid the complexity of on-line monitoring many tools are based on traces. This refers to debugging tools [5] as well as to performance analysis [6]. A main reason for the popularity of traces is the existence of several standard formats, like SDDF [12] or PICL [4].

The goal of the OMIS (on-line monitoring interface specification) project is to separate tool and monitor development by providing a standard interface between them. It will also provide tool developers with on-line capabilities instead of only trace data to be processed. The project is a joint effort between groups at Technische Universität München and at Emory University (see [9]).

The remainder of the paper will give some insight into basic ideas of OMIS and present the design of OCM, an OMIS compliant monitoring system for PVM running on heterogeneous workstation clusters.

2 OMIS and OCM

The OMIS project started in summer 1995. Beginning of 1996 we published the OMIS document version 1.0 as a first definition of what we specified during that time. Version 1.0 serves as a basis for the design and implementation of an OMIS compliant monitoring system (OCM).

One of the main goals of the OMIS project is to decouple tool and monitor development. Groups being involved in one or the other research topic can now fully concentrate on their proper research. Furthermore, having a specified standard, allows to broaden the capabilities of parallel tools. Eventually we will come to

- **interoperable tools** where tools from different sources can be used concurrently and
- **unified tool environments** which can be used in identical form on different target architectures.

OMIS defines an on-line mechanism of tool/program-interaction. We consider interactive tools like e.g. debuggers and performance analyzers and automatic tools like e.g. load balancing facilities. Off-line tools performing program trace analysis are supported by connecting an on-line trace generator tool to OMIS. After program completion trace data recorded can be fed into any appropriate off-line tool. This is identical to working principles of trace analysis tools. However, with OMIS manipulation facilities of an OCM will remain inactive.

The goal of the research group at LRR-TUM is to design and implement OCM for a specific target architecture and to integrate it with a sophisticated tool environment. This is covered with THE TOOL-SET approach (see [8]). OMIS and OCM will be freely available to the research community and may form the basis for advanced tool technology. For the first step we will design and implement an OMIS compliant monitoring system for PVM running on workstation clusters.

The following sections of the paper will outline general concepts for the realization of such a system. Concrete design decisions will be discussed as they form the basis for an implementation.

3 Design Concepts of OCM

Concepts for the design of an OMIS compliant monitoring system can be divided into two categories: First, many design concepts refer to aspects which can be found in the specification document itself. Methods of interaction with the tools belong to this category. Second, it is inevitable to design concepts which guarantee an efficiently working implementation of OCM on a certain target architecture.

The first category mainly influences the logical behavior and provides a frame for the software structure of the the implementation whereas the second category already fixes which software mechanisms provided by the system have to be used.

The OMIS document tries to set up a monitoring interface which is as far as possible independent from the programming library used for applications. However, any instance of an OMIS compliant monitoring system must be implemented for a fixed programming library or parallel programming language and a dedicated target architecture. As the interface specification tries to be abstract, our goal is to identify and implement a generic monitoring system which can easily be ported among different combinations of software and hardware architecture. The design of the individual modules of the monitoring system shall always identify whether or not the modules belong to the core system. This also implies an extensive definition of interfaces between parts belonging to the core system and architecture specific parts.

Cooperation between tools and the monitoring system is specified to be composed of service requests and replies to them. A service request is sent to the monitoring system as a coded string which describes which activity has to be invoked on the occurrence of a specified event. As a matter of fact, service requests program the monitoring system to listen to event occurrences, perform actions, and transfer results back to the tool. The monitoring system inherently follows an event/action working model. Cooperation with tools is also covered by this model as the receipt of service requests represents an event to which an appropriate action belongs.

As the application program to be manipulated is distributed the monitoring system has to adopt this architectural principle. The monitoring system belonging to one application program consists of a set of communicating single-node monitors. They are functionally fully equipped and additionally perform the task of coordinating their activities in order to guarantee detection of global events and/or execution of global actions. Global here means non-local, i.e. any activity that involves more than one node of the application's node set.

One of the main questions refers to the type and number of new components which have to be added to the already available software and hardware environment. Technically the question is of how to start up the monitoring system, how many components belong to it, how should they interoperate, how to manage dynamic changes, and finally, when will they be deleted from the system?

Currently, the monitoring system is designed to consist of one monitor process per node of the virtual machine of a user. Each monitor process runs under the same user identification as the application program. No special rights are necessary to install this software. In case of multiuser tools like e.g. load balancers we need an additional component on every node which can determine the node status. This so-called node status reporter is designed as a process which has to be integrated into the system by the super user. It is persistent in the sense that it runs also on nodes where PVM tasks are not yet or no longer existing. The node status reporter can be accessed by any single-node monitor in order to get up-to-date load values.

In addition to having monitor and node status reporter processes we also need to instrument libraries and daemons of all runtime environment software. In our systems these are the PVM library and daemon and PFSLib library and daemon. The latter is a parallel file system for workstation clusters developed at LRR-TUM [7]. Instrumentation is inevitable to efficiently observe the dynamic program behavior and to be able to manipulate it.

Appropriate communication mechanisms have to be chosen for component interoperation on a single node and between nodes.

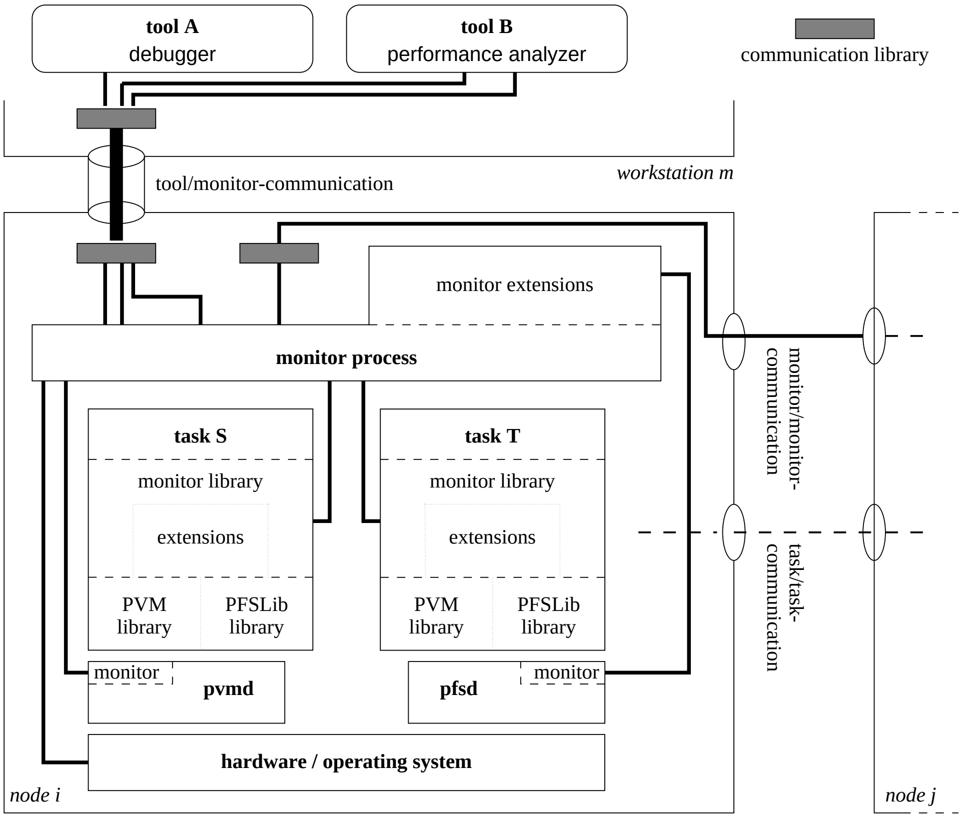

Figure 1: The software module structure of the monitoring system

4 Design of the Software Module Structure

Let us now have a closer look at the software module structure of the complete system. On the top of Figure 1 we can see various tools running e.g. on a remote workstation. Via a communication library they send requests to the monitor process on any node of the system. The monitor handles request interpretation and starts appropriate activities. A more detailed description of its tasks will follow in the next section.

The monitor process is linked together with extensions which are necessary for controlling specialized software like the PFSLib daemon. A discussion of this functionality is beyond the scope of this paper. Please refer to the OMIS document for details [9].

Every monitor cooperates with tasks on its node, with daemons of the programming environment, with the operating system, and with monitors on other nodes. Task code is composed of the code performing the program's calculation, library code linked to it, and a monitoring library added to this complex. The monitoring library implements for instance monitor objects like timers and counters which are directly handled in the application context. Usually, manipulation of these objects is triggered by the execution of programming library calls. E.g. with performance analysis a send message call might add the call completion time to a clock counter. Potentially all calls of all programming libraries used can be instrumented by adding wrappers to the original function calls which can then perform the necessary activities. Technically, communication with the monitor process is designed (but not restricted) to be handled via shared memory segments.

Instrumentation of the PVM daemon is needed for special purposes only like e.g. getting a signal when a message is received from another node.

Finally, with the operating system there is no active instrumentation employed. Here all cooperation is handled via well defined system calls (e.g. the `ptrace()` interface) and devices (e.g. `/dev/kmem`).

The following section will discuss the processing of requests in the monitor process itself. We will not describe the module structure of the monitor but explain which stages a request passes until an answer gets back to the tool.

5 Request Processing in the Monitor Process

From the tool's point of view a request to the monitoring system is a function call which gets two input parameters: the service request and a callback function. The service request is represented as a string while an address denotes the callback function for passing result parameters back to the tool. There is no dedicated master monitor, thus a service request can be sent to any node.

We distinguish synchronous and asynchronous service requests. Synchronous requests are followed by an immediate reply whereas with asynchronous requests we will have a reply every time the event defined in the request happens.

Let us now have a look at the node local request processing (see Figure 2). The receiving monitor first scans and parses the service request string and converts it into appropriate data structures. OMIS defines a request format consisting of an optional event specification and a list of actions to be invoked, which must not be empty. In addition all event and action specifications carry with them node numbers which indicate on which nodes they have to be executed.

The request distribution compiler analyzes these node numbers and forwards local requests to the next stage, the node request receiver. Requests which are not only or not at all meant to be handled locally will be forwarded to the appropriate monitors on remote nodes. At the same time other nodes might send requests to this node.

Local requests and incoming requests will both be received by the node-request receiver which will forward them to the instrumentation manager.

The instrumentation manager splits event specifications into lists of corresponding basic instrumentations. These basic instrumentations finally are applied to the appropriate code segments of all software components involved (i.e. task codes and daemon codes). Measuring for example the duration of sending routines is handled by two events: the first starts an integrating timer on the event of entering a send routine; the second stops the timer if the process exits the routine. Activation of instrumentations is done by means of basic instrumentation services (in our example both instrumentation points are located in the communication library of PVM, i.e. in the task's address space).

The instrumentation manager not only activates instrumentation but also triggers the event/action management. The latter stores all defined events together with lists of actions to be invoked on their occurrence. Note that with tool manipulation getting more complex new actions will possibly be appended to already existing events. If actions are to be deleted the event/action manager checks whether or not the event has to be purged, too. In this case, all instrumentations belonging to this event will be deactivated.

Event/action processing might also be triggered by remote events (e.g. with requests like "stop task on node A if something happens on node B"). In this case the local monitor receives the event occurence information from a remote node and starts appropriate activities. The feature of involving several nodes in one service

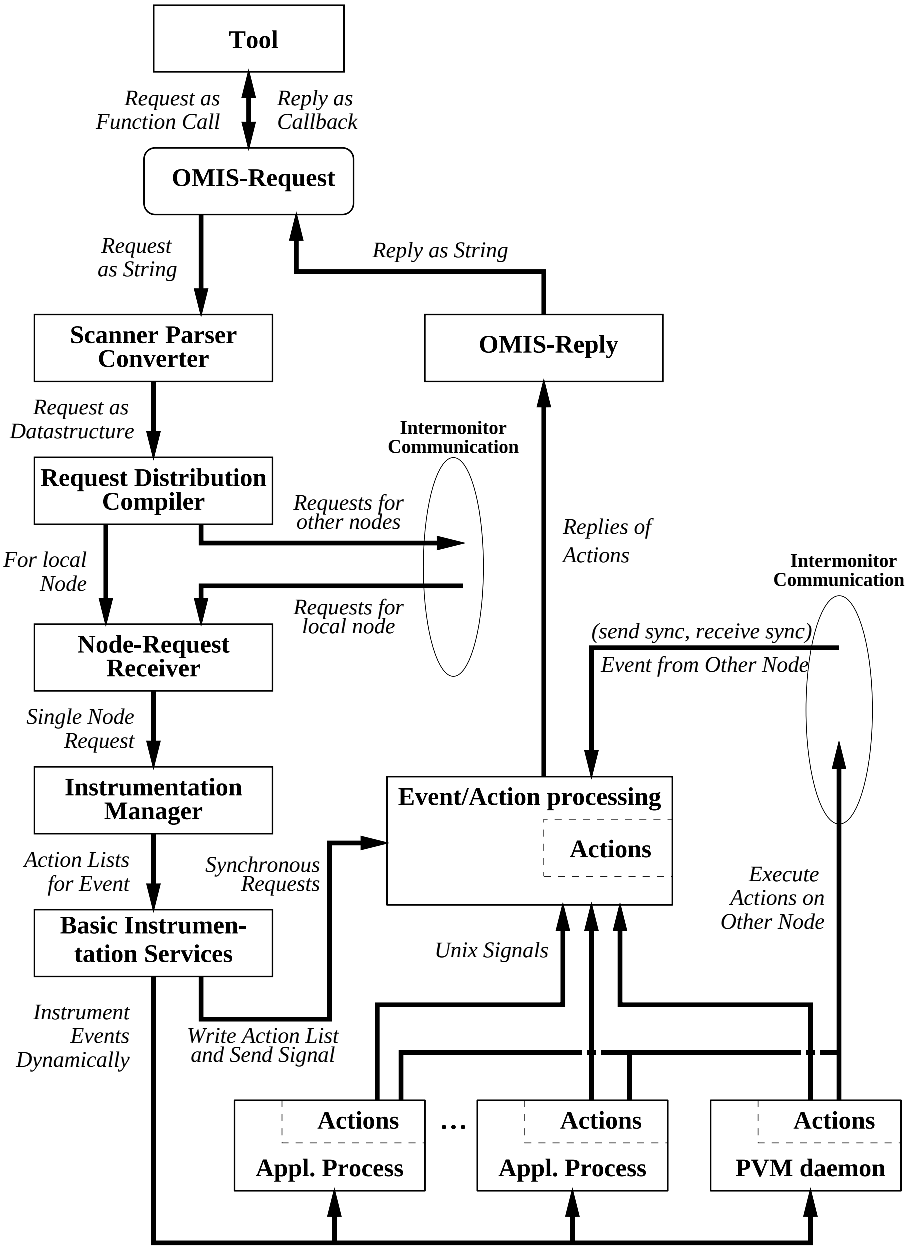

Figure 2: Proceeding of requests from tools

request is mainly used with debuggers and load balancers. It allows to specify breakpoint conditions which spawn several nodes.

In case of a synchronous service request sent by a tool the action list will be activated immediately by the instrumentation manager as there is no event we have to wait for.

With asynchronous service requests the actions are triggered as soon as the event occurs. The event/action manager searches the appropriate action lists and invokes all specified actions. Their results are collected and passed back to the requesting tool by calling the specified callback function.

The design documents which are currently developed for the OCM implementation comprise a concise description of all modules and data structures involved in this cycle. Please refer to [3] for details.

6 Project Status and Future Work

Currently, the OMIS document is un-modified. We received helpful comments and criticism that will be integrated in a new version which is due to appear in late fall 96. With the OCM project – the realization of what OMIS describes – we started beginning of 96. The design phase will be finished in September and will result in publicly available design documents [3]. Implementation will start at that time. Currently, we already design and implement the node status reporter. This work is carried out at Emory University in the group of Vaidy Sunderam.

References

[1] 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.

[2] 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. http://www.cscs.ch/pub/CSCS/techreports/1995/CSCS-TR-94-09.ps.gz.

[3] M. Geischeder, M. Uemminghaus, H. Zeller, T. Ludwig, Michael Oberhuber, R. Wismüller, and A. Bode. OCM: OMIS Compliant Monitoring System — Design Documents. Technical report, to appear at Technische Universität München, Munich, Germany, October 1996.

[4] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. A User's Guide to PICL, a portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, October 1990.

[5] S. Grabner, D. Kranzlmüller, and J. Volkert. Debugging parallel programs using atempt. In B. Hertzberger and G. Serazzi, editors, *High Performance Computing and Networking 1995*, number 919 in Lecture Notes in Computer Science, pages 235–240. Springer, May 1995.

[6] M. T. Helth and J.E. Finger. Paragraph: A toll for visualizing performance of parallel programs. Technical report, Oak Ridge National Laboratory, 1993.

[7] S. Lamberts, T. Ludwig, C. Röder, and A. Bode. PFSLib — a file system for parallel programming environments. Technical Report TUM-I9619, SFB-Bericht Nr. 342/10/96 A, Technische Universität München, Munich, Germany,

May 1996. http:/~ludwig/papers/tl9602/WWW/report.html
ftp:/~ludwig/papers/tl9602/PS/article.ps.gz.

[8] T. Ludwig and R. Wismüller. THE TOOL-SET environment. In A. Bode, T. Ludwig, V. Sunderam, and R. Wismüller, editors, *Workshop on PVM, MPI, Tools, and Applications*, pages 28–32. Technische Universität München, November 1995. http:/~ludwig/papers/tl9505/WWW/article.html
ftp:/~ludwig/papers/tl9505/PS/article.ps.gz.

[9] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification. Technical Report TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Munich, Germany, February 1996. http:/~ludwig/papers/tl9601/WWW/article.html
ftp:/~ludwig/papers/tl9601/PS/article.ps.gz.

[10] 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, May 1990.

[11] A. K. Petrenko. Methods for debugging and monitoring parallel programs: A survey. *Programming and Computer Software*, 20(3):113–129, may–june 1994.

[12] Daniel A. Reed. Experimental analysis of parallel systems: Techniques and open problems. In Günter Haring and Gabriele Kotsis, editors, *Computer Performance Evaluation*, volume 794 of *LNCS*, pages 25–51. Springer, 1994.

[13] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships on distributed computations: In search of the holy grail. Technical Report SFB 124-15, University of Kaiserslautern, December 1992.