OCM — A Monitoring System for Interoperable Tools*

Roland Wismüller, Jörg Trinitis, Thomas Ludwig
Technische Universität München (TUM), Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Arcisstr. 21, D-80333 München
email: {wismuell|trinitis|ludwig}@informatik.tu-muenchen.de

Abstract

On-line tools for parallel and distributed programs require a facility to observe and possibly manipulate the programs' run-time behavior, a so called monitoring system. Currently, most tools use proprietary monitoring techniques that are incompatible to each other and usually apply only to specific target plattforms. The On-line Monitoring Interface Specification (OMIS) is the first specification of a universal interface between on-line tools and a monitoring system, which is not targeted towards a single specific class of tools (e.g. performance analyzers). Instead, it can serve very different kinds of tools at the same time, thus enabling interoperable, portable and uniform tool environments. The paper gives an introduction into the basic concepts of OMIS and presents the design and implementation of an OMIS compliant monitoring system (OCM). Special attention is given to the techniques needed to deal with the distributed target system.

1 Introduction

The development of parallel programs is inherently more difficult and expensive than that of sequential programs. This is partly due to the lack of widely available, powerful tools supporting test and production phases of those programs. The support required here is essentially the same as for sequential programs and among the most important tools are visualizers, performance analyzers, and — of course — debuggers. Other tools apply specifically to parallel and distributed programming, e.g. load balancers or deterministic execution controllers.

The need for tools was recognized by many different research groups all over the world, and a variety of tools supporting parallel programming was developed [11]. These tools differ widely in terms of aims and quality, ranging from early research prototypes to industrial strength tools. What these tools have in common is their need for a module that enables them to observe and possibly manipulate the parallel, distributed application, a so called *monitoring* module. Usually every tool includes a monitoring module that is specifically adapted to its needs. Since these modules typically use special low level interfaces to the operating system and possibly the underlying hardware, these layers tend to be incompatible to those required by other tools. As a result of this, it is impossible for a developer to connect two or more tools to a running application at the same time. Even worse, if he wants to use two different tools, one after the other, he usually has to re-compile, re-link, or otherwise re-process his application.

In this paper we will describe the design and implementation of a universally usable on-line monitoring infrastructure that is general enough to support a wide variety of tools and their interoperable use. We will first describe existing approaches, analyze the requirements for a general monitoring interface, then introduce our approach and finally describe our implementation.

*This work is partly funded by *Deutsche Forschungsgemeinschaft*, Special Research Grant SFB 342, Subproject A1.

To appear in Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98, August 1998, Welches, OR, USA

2 State of the Art

Existing tools can be classified into interactive vs. automatic, observing vs. manipulating, and on-line vs. off-line tools. Interactive tools (e.g. debuggers) usually offer a graphical user interface whereas automatic tools are hidden in layers close to the operating system (e.g. load balancers). Observing tools are tools that merely read and present data, e.g. program and data flow visualizers or performance analyzers, and are to be distinguished from tools that allow to manipulate a running application, e.g. computational steering and debugging tools. A crucial distinction is between on-line tools that operate while the program is running and off-line tools that run after the program has finished. Off-line tools usually operate on traces generated at runtime. It should be clear that an off-line tool can never manipulate the running application and thus falls into the category of observing tools. Taking a look at existing tools one can state that:

- There are more off-line tools than on-line tools.
- Most tools are only available on one or very few platforms.
- Different on-line tools (and their monitoring systems) can typically not be applied concurrently to the same program.
- There are no generally available integrated development environments for parallel and distributed programming.

There are several reasons for this. First, developing on-line monitoring systems is complex and expensive. Second, on-line monitoring systems are difficult to port from one platform to another. And third, on-line monitoring systems are usually specifically adapted and restricted to the tools that sit on top of them and require incompatible modifications to the application. Nevertheless, very powerful on-line tools have been developed in the later past. Two that are especially interesting concerning their monitoring parts will be discussed here.

In the area of debugging tools, p2d2 by Cheng and Hood [4] is an important representative. Different from most earlier approaches, p2d2 clearly separates the platform specific on-line monitoring system from portable parts of the tool itself. To accomplish this, a client-server approach is used [5]. The p2d2 server contains all platform specific parts, whereas the p2d2 client consists of the portable user interface, program model, etc. Client and server communicate via a specified protocol. The server offers objects such as processes and variable values. The client can then operate on these objects. Cheng and Hood encourage platform vendors to implement their own p2d2 debugger servers that comply with the p2d2 protocol. Users could then simply use their existing p2d2 debugger clients to debug applications using these servers in a portable way on different machines. In the meantime, Cheng and Hood supply their own debugging servers based on gdb, the GNU debugger [10].

Taking a look at performance analysis tools, the ambitious Paradyn project by Miller and Hollingsworth [8] clearly deserves a closer look. On the low-level side, Paradyn utilizes the exceptional technique of dynamic instrumentation [3]. With this technique, code snippets that perform tasks necessary for performance analysis are dynamically inserted into and removed from the running process' code. The interface between the tool and the dynamic instrumentation library *dyninst* is clearly defined and published [1].

On the higher level, Paradyn uses the so called W^3-model to automatically determine performance bottlenecks in the running application [2, 8]. The W^3-model in particular profits from dynamic instrumentation because intrusion can be kept low by removing instrumentation that is no longer necessary. In summary, Paradyn clearly represents the state-of-the-art of performance analysis.

In contrast to earlier developments, the above two tools are two of the very few that define an explicit interface between the monitoring system and the tool itself. By clearly defining this interface it becomes possible to separate the development of tools from the development of the on-line monitoring system. And, since the interface is the same on all platforms, porting the tool becomes much easier, improving the availability of tools and thus overall propagating the use of tools for parallel programming.

Unfortunately, p2d2 as well as Paradyn both target only a single class of tools: p2d2 is limited to debugging, whereas Paradyn/dyninst is concentrated on performance analysis. It is very difficult or even impossible to build new tools of a different kind on top of either of them, for example debuggers on top of dyninst or performance analyzers on top of p2d2. Also it is not possible to build integrated and interoperable tools with either of these environments. OMIS, on the other hand, is designed for exactly that.

3 The OMIS Approach

One key idea in OMIS (*On-line Monitoring Interface Specification*) [7] is to define a standardized interface between the tools and the on-line monitoring system. This is comparable to p2d2 and Paradyn/dyninst. However, our approach is more general: we do not concentrate on a single class of tools like performance analyzers (as is the case for Paradyn) or debuggers (like p2d2). OMIS supports a wide variety of on-line tools, including centralized, decentralized, automatic, interactive, observing, and manipulating tools (off-line tools can be implemented by a tiny on-line module that writes the necessary trace files).

The tools we initially had in mind were the tools from The Tool-set [12], developed at LRR-TUM: VISTOP (program flow visualization), PATOP (performance analysis) [13], DETOP (debugging) [13], CoCheck (checkpointing and process migration), LoMan (load balancing), Codex (controlled deterministic execution), and Viper (computational steering). But we do not only want to support classical stand-alone tools. OMIS is intended to support new kinds and combinations of tools as well. For example **interoperable tools** (two or more tools at the same time), e.g. a performance analyzer covering and measuring the dynamic load balancing and process migration as well. Another topic are **integrated tools**, e.g. a debugger integrated and cooperating with a checkpointing system and a deterministic execution unit, enabling the user to start debugging sessions from checkpoints of long lasting program runs.

By using OMIS as a standard interface between tools and monitoring systems we achieve two major goals: First, development of new tools is easier because they can now be based on a well defined infrastructure that observes and manipulates parallel programs. Second, porting tool environments to new target architectures is easier because it is sufficient to port the monitoring system only. Existing tools can then be used as before. Thus, the effort of bringing m tools to n target systems is reduced from $m \times n$ with a monolithic approach to $m + n$ with an OMIS-based approach.

In order to achieve a high degree of versatility of the monitoring interface with respect to different types of tools we use the concept of a programmable monitoring system which is able to handle services of various types. The main concept of its programming language is the event/action-paradigm: The behavior of the monitoring system is completely defined by a set of event/action-relations which are sent from the tools to the monitor in form of simple text strings (Later they are compiled into an internal representation.) Each such relation defines a condition that the monitor observes. Whenever the condition comes true, the monitor performs the corresponding actions. Thus, we define for example a breakpoint by sending a relation where the event defines an address in the program's code and the action defines to stop a process. The whole string is called a request. We distinguish conditional requests (with event definition) from unconditional requests. The latter carry an empty event definition and the defined actions are performed immediately (e.g. to stop a process).

The power of this approach comes from the fact that events as well as actions may refer to lists of objects. Thus we can easily monitor system-wide conditions and perform global actions like e.g. a manipulation of all processes belonging to a program. The notion of an object in the context of OMIS will be discussed in detail later.

An analysis of the functionality required by different tools shows that the set of events they use is rather identical, although the actions differ. E.g. both a debugger, a visualization tool, and a performance analyzer will be interested in an event "a process sends a message". However, the debugger may want to stop the application when this event happens, the visualizer may want to store a trace entry, while the performance analyzer may want to increment a counter containing the number of send operations. Therefore, the functionality offered by OMIS concentrates on the detection of common events and on mechanisms to associate actions with the occurrence of these events. The specific functionality needed by a tool is achieved by a proper combination of actions with events. The OMIS core already provides a rich set of events (including user-defined events) and actions. In addition, tools can add their own events and actions via a well-defined extension interface (see below).

The main interface between the tool and the monitoring system consists of only a single function, which accepts requests, i.e. event/action-relations from the tool in a machine independent string representation.

The following shows the syntax of the main interface procedure being used by the tools:

```
Omis_reply omis_request(char * request,
                void (* callback)(Omis_reply reply,
                                  void * param),
                void * param,
                Omis_flags flags);
```

The string `request` representing the event/action-relation is the most important parameter to this function. Additional information has to be supplied depending on the type of request. With unconditional requests the `flags` can be used to select whether the `omis_request`-call shall work in a blocking or a non-blocking manner. Conditional requests however enforce a non-blocking behavior as we cannot wait for the defined event to come true. The tool specifies a callback function which is activated by the monitoring system whenever the actions belonging to that request have been executed. Note that we cannot predict when and how often this will be the case.

In order to minimize intrusion, requests can be quickly enabled and disabled when, for example, a measurement has to be temporarily disabled. When it is no longer necessary, the whole request can be deleted from the monitoring system.

Every tool that connects to an OMIS compliant monitor is able to define its own view of the observed system. This is done by explicitly attaching to the nodes and processes of interest. Different tools can thus have different views if desired, and are only informed about events taking place within their observed system. This is especially useful when attaching tools to non-exclusively used target environments.

While not being object oriented in the in the usual sense, the notion of objects forms an important part of the interface. A major conceptual issue of OMIS is the set of object types that are supported. The current version of the specification is oriented towards the message passing programming paradigm. As a consequence, all software object types relevant in this field are offered. A tool is provided access to node objects, process and thread objects, message queue and message objects. For each object type a set of corresponding services is defined. They can be further subdivided into information services (for inspection of an object's state), manipulation services (to modify the state), and event services (to trigger both former services on state changes). In addition to software objects we define special types for internal objects of the monitoring system and for conditional requests. Both are subject to manipulation services.

Every object is represented by an abstract identifier called a token. The interface defines two methods for token conversion following the concept of a hierarchy relation defined on the individual object types. At the top of the hierarchy, there is the object type "system" which at the next level includes the node objects. Nodes include processes, processes include threads. Messages are part of message queues which themselves belong to threads, processes, or nodes. (The hierarchy may vary for different programming models.)

The two conversion functions are called localization and expansion. They are automatically applied to every event or action service definition that has tokens as parameters that refer to an object type different from that for which the service is defined. In this case the token is converted to a single token or a list of tokens of the object type the service works on. To make this clearer: a node token specified in a service that works on processes is expanded to the list of all processes under observation on this node. Vice versa, a process token in a service that works on nodes is localized, i.e. replaced by the corresponding token of the node this process is located on.

A special role is given to the empty token list. It means: "the complete system we are currently attached to". Thus `thread_stop([])` means to stop all threads in all observed processes on all observed nodes. A problem arising from this semantics is the question of when to expand to the actual list of thread tokens. A reasonable semantics is: "all threads at the moment when the event takes place that triggers the action". This implies a) that the token list has to be expanded only when the event takes place and b) that it has to be re-calculated for every further event occurrence. A complex semantics like this is inevitable to cover dynamic systems that exhibit a varying number of nodes and processes. Section 5 will give an example how the concept of token conversion is realized in the implementation of a monitoring system.

Although the above concepts are very powerful if wisely applied, there are always situations where they do not suffice to support some specific environment. To overcome this, OMIS employs another concept: extendibility. Two types of extensions are defined: tool extensions and monitor extensions. Tool extensions (e.g. for distributed tools) introduce new actions into the monitoring system. As stated above, OMIS concentrates on the detection of events within the system under observation. The processing of these events can be very tool-dependent, so the actions offered by the OMIS core may not suffice. E.g. for performance analysis, different tools need various kinds of timers with differing semantics, statistical processing, various trace formats etc. Tool extensions allow each tool to add actions performing exactly the type of processing required by that tool. In contrast, monitor extensions are used when there are additional system components to be observed (e.g. programming libraries or special hardware components). Monitor extensions thus can introduce new events, actions and objects. One monitor extension is for example the PVM extension. It introduces message buffers, barrier objects and services (events and actions) related to the PVM programming model. Other extensions might introduce mailbox objects, shared memory objects and such to the system.

A detailed description of the current version of OMIS can be found in [7].

4 Design of an OMIS Compliant Monitoring System

Starting from the OMIS specification, we have done a first implementation of an OMIS compliant monitoring system, called OCM. In contrast to OMIS itself, which is not oriented towards specific programming libraries or hardware platforms, the OCM currently targets PVM applications on networks of UNIX workstations. However, we also plan to extend this implementation to MPI and to Windows NT workstations.

Goals of the OCM project. The implementation of the OCM serves three major goals. First, it will be used as an implementation platform for the tool developments at LRR-TUM [12] and at other institutes. Currently, there are cooperations with the MTA SZTAKI Research Institute (formerly KFKI-MSZKI) in Budapest, Hungary[1], where the OCM will be used to support debugging and performance analysis of programs designed with the GRADE graphical programming environment [6], the University of Vienna, Austria (debugging of Vienna-FORTRAN programs), and the Institute of Computer Science, AGH in Krakow, Poland (performance analysis of MPI applications).

The second goal is to validate the usability of the concepts and interfaces specified by OMIS and to gain experiences guiding further refinements of the specification. This also includes closer investigation of the possibilities and benefits of interoperable, coexisting, and cooperating tools.

Finally, the OCM should also serve as a reference for other implementors who want to develop an OMIS compliant monitoring system. Despite our efforts to avoid this, there are undoubtly cases where the OMIS document still leaves room for interpretation or where we wanted to explore possible implementations before specifying the exact behavior. The reference implementation will help us to identify many of these cases and will serve as an addition to the specification by clarifying the intended behavior. Therefore, the OCM will be made available to the public under the terms of the GNU license.

Global structure of OCM. Since the current target platform for the OCM is a network of workstations, i.e. a distributed system where virtually all of the relevant information can only be retrieved locally on each node, the monitoring system must itself be distributed. Thus, we need one or more local monitoring components on each node. We decided to have one additional process per node, which is automatically started when a tool issues the first service request concerning that node. An alternative implementation would be to completely integrate the monitoring system into the programming libraries used by the monitored application, as it is done with many trace collectors. However, we need an independent process for various reasons. First, the OMIS compliant monitoring system must be able to autonomously react on events, independently of the application's state. Second, some necessary operating system support (especially the `ptrace` system call, the `/proc` file system and possibly access to the `/dev/kmem` device) requires the use of a separate process for technical reasons. Finally, it is very questionable to try to include code that is — among other things — intended to support debugging into the address space of the suspected erroneous process.

A principal decision has been to design these monitor processes as independent local servers that do not need any knowledge of global states or the other monitor processes. Thus, each monitor process offers a server interface that is very similar to the OMIS interface, with the exception that it only accepts requests that can be handled completely locally. From the viewpoint of software engineering, this decision offers the great advantage that the design of the local monitor processes need not pay any attention to the fact that we have a distributed monitoring system. This greatly simplifies the implementation and helps us to improve the robustness of the resulting system. In addition to that, the fact that local monitors only act on local objects eases the modeling of the system as a whole when applying formal methods supporting proofs of correctness, deadlock avoidance, etc.

However, since OMIS itself allows a tool to issue requests that may relate to multiple nodes, we need an additional component in the OCM. This component, called Node Distribution Unit (NDU), has to analyze each request issued by a tool and must split it into pieces that can be processed locally by the monitor processes on the affected nodes. E.g. a tool may issue the request `:thread_stop[p_1,p_2]`[2] in order to stop all threads in the processes identified by the tokens *p_1* and *p_2*. In this case, the NDU must determine the node executing process *p_1* and the node executing *p_2*. If the processes are located on different nodes, the request must be split into two separate requests, i.e. `:thread_stop[p_1]` and `:thread_stop[p_2]`, which are sent to the proper nodes. Addition and removal of nodes, as it can occur in environments like e.g. PVM, is also detected and handled by the NDU. Section 5 gives a more detailed and general description of this distribution process. In addition, the NDU must also assemble the partial answers it receives from the local monitor processes into a global reply sent to the tool.

In order to keep our first implementation of the OCM manageable, the NDU is currently implemented as a single central process. Thus, in a typical scenario, we may have several tool processes, which are arbitrarily distributed in the network and which connect to the central NDU process. The NDU is in turn connected to the local monitor processes on the monitored nodes of the target system, as shown in Figure 1. The figure also shows that the NDU contains a request parser that transforms the event/action-relations specified by the tools into a preprocessed binary data structure that can be handled efficiently by all other parts of the monitoring system[3]. Our current results make us confident that the decision to have a central NDU component does not cause any problems in terms of scalability or performance. However, in future versions we might replace the current implementation of the NDU with a distributed implementation for the sake of fault tolerance or improved performance, if this becomes desirable.

The communication between all of these components is handled by a small communication layer, which offers non-blocking send and interrupt-driven receive operations. We need interrupt-driven receives — or comparable techniques, such as Active Messages or the use of multiple threads — to avoid blocking of the monitor processes. It was one of our main goals to have most of the functionality inside the monitor processes work asynchronously. Blocking would result in the inability to timely react on occurrences of other important events. A polling scheme using non-blocking receive

[1]Project GRADE-OMIS funded by German and Hungarian ministries, contract UNG-053-96

[2]The leading colon in the request is the separator between event definition and action list. Since the event definition is empty, the action will be executed immediately

[3]For readability, we will also represent these transformed requests as strings throughout the paper

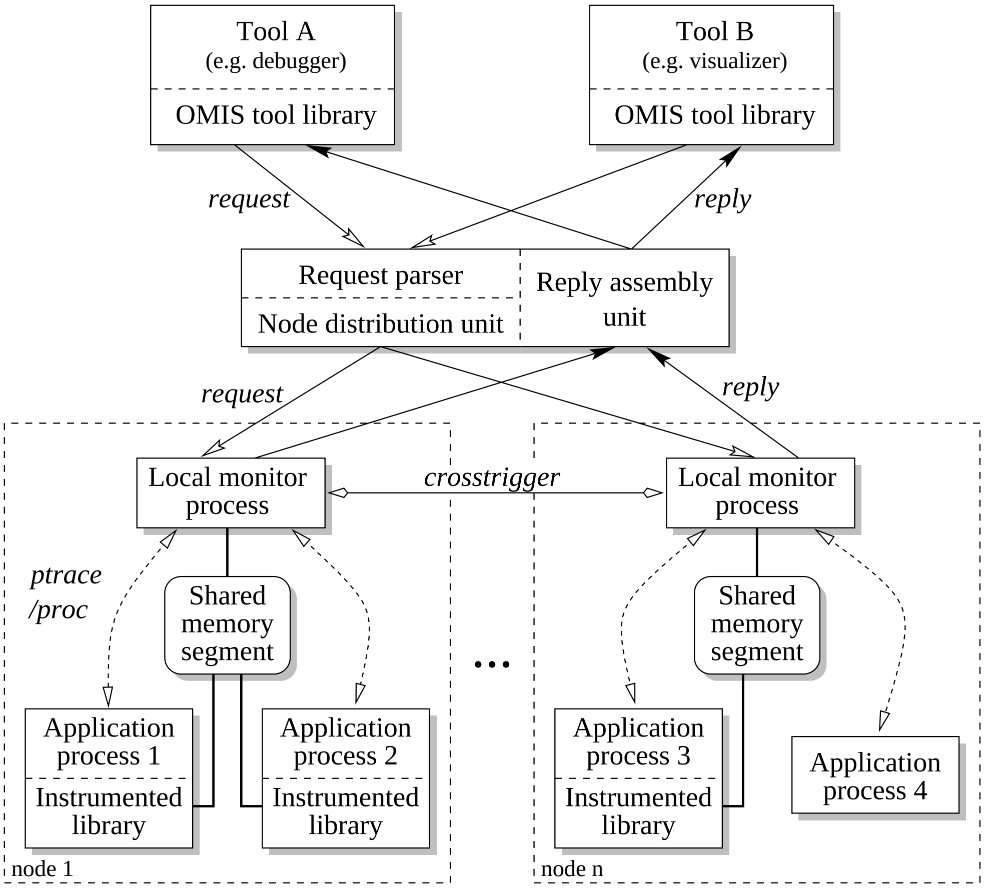

Figure 1: Coarse Structure of the OCM

operations is not feasible, because it consumes too much CPU time and thus leads to an unacceptable disturbance of the monitored application.

As an additional feature, our communication layer can also be used for intra-process communication to allow for different partitioning of the system into various sets of processes. In this very special case, the asynchronous receive function is called directly out of the send operation, which results in a local procedure call.

For the current version of the OCM, the communication layer is based on PVM. However, we are working on an independent version based on BSD-style sockets, which is a necessary preliminary for the MPI port.

As is indicated in Figure 1, tools are linked against a library that implements the procedural interface defined by OMIS and transparently handles the connection to the NDU and all the necessary message passing and asynchronous reply handling.

Structure of the local monitor. The local monitor process operates in an event driven fashion. This means that the process is normally blocked in a system call (either `wait` or `poll`, depending on whether we use `ptrace` or `/proc` for process control), which returns when an event in one of the monitored processes has been detected by the operating system. In addition, the call is interrupted when a message arrives. After the monitor process has been unblocked, it analyzes the exact kind of event that has occurred and executes the list of actions that are associated with that event[4]. If the event happens to be the arrival of a service request, the associated action causes the request to be received and analyzed. In the case of an unconditional request (i.e. a request with an empty event definition), it is executed and the results are sent back. When a conditional request is received, the monitor process dynamically inserts or activates the necessary instrumentation of the target system and stores the request's action list, so it can be executed later when the event occurs.

All the lower level functionality that is needed to instrument the target system is encapsulated inside a single module named the *target interface*. This module separates all other parts of the local monitor process from platform and operating system specific interfaces like `wait`, `poll`, `ptrace`, or `/proc`. Also knowledge about stack layout, registers etc. is only present inside this module. Concentrating these parts into a single module greatly enhances portability. Currently, implementations for `ptrace` or SVR4 `/proc` exist. Future versions might include a port to the corresponding interfaces for Windows NT, which is expected to be straightforward. We are also taking a look into using upcoming versions of dyninst [1] to base our target interface on top of it, as soon as this becomes feasible.

[4] The exact algorithm and event classes used to determine which exact event occurred are beyond the scope of this paper.

Unfortunately, this relatively easy execution scheme of having a monitor process observing and controlling the monitored processes is insufficient due to efficiency constraints. For example, assume that a performance analysis tool wants to measure the time spent in `pvm_send` calls for the process identified by token *p_1*. The tool would then issue e.g. the following requests:

```
thread_has_started_lib_call([p_1],"pvm_send") :
    pa_timer_start(ti_1)
thread_has_ended_lib_call([p_1],"pvm_send") :
    pa_timer_stop(ti_1)
```

Thus, a timer is started when `pvm_send` is called by *p_1*[5] and is stopped again when the call returns. However, if event detection and action execution are only performed in the monitor process, we get an overhead of at least four context switches per library call, which renders performance analysis useless.

Thus, the OCM must be able both to detect certain events within the context of an application process and to execute simple actions directly in that process, thereby removing the need for context switching. To achieve this, the monitored processes must be linked against instrumented libraries, i.e. extended versions of standard programming libraries that include additional parts of the monitoring system. In order to tell the monitoring parts in these libraries what they shall do, we use a shared memory segment that stores information on which events to monitor and which actions to execute once an event occurs. The shared memory segment is also used to store data needed by the actions (e.g. the timer value in the above example) and for returning results. Note that the actions that are executed within the application processes should not return results every time they are executed (except in the case of errors), since this would again result in context switches. Rather they update data stored in the shared memory segment which are read by actions executed in the monitor process upon request from a tool.

With this extension, the final structure as shown in Figure 1 results. Note that the OCM does not require that each monitored application process is linked with instrumented versions of programming libraries. When the OCM attaches to an application process, it automatically detects which services are supported by the library versions linked to that process. To support this, each instrumented library contains a global data structure defining the services offered by the library. When the OCM attaches to the process, it reads this information from the process's memory. Thus, when a service is requested, the monitor can decide whether it can be executed in the context of the application process. If not, it may either return an error message, or use other implementation techniques for this service, e.g. dynamic instrumentation [3] or trap instructions.

Modularization of the OCM The design of the OCM implementation is chosen such that the code exhibits a high degree of portability. For this to be guaranteed the use of standard interfaces was favored wherever possible. Services that deal with process management are either based on the ptrace-interface or on the `/proc` file system. Others employ standard subroutines provided by Unix and its variants. Where further modularization was hindered by the fact of missing interfaces we decided to conceive new solutions. This is for example the case for the information service `node_get_info`. Its task is to provide the requesting tool with a comprehensive set of information on a chosen set of nodes. Static information (e.g. CPU type, main memory installed) can be collected as well as dynamic information (e.g. current number of processes on a node).

Using standard system calls for information retrieval suffers from two decisive drawbacks:

- Available interfaces vary between operating systems. Furthermore, there is no single procedural interface to satisfy the requirement of quick and easy information inquiry.
- The use of multiple tools would multiply the overhead caused by information gathering although there is an identical set of data for all tools. Especially when orienting towards performance analysis or load balancing tool intrusion must be kept minimal.

The approach chosen to deal with these problems consists in specifying a new general purpose interface for node-related information and to provide an efficient implementation of this interface for a set of target architectures. The approach is called node status reporter (NSR) and is part of the OCM project. However, it can be used as a stand-alone solution for tools that need access to above mentioned information only.

The following major objectives are pursued in the design of the NSR with respect to above listed requirements:

- *Portability:* The target architecture of NSR is a cluster of heterogeneous workstations. The same set of status information has to be accessible in the same way on every architecture for which an implementation exists. New architectures should be easily integrated with existing implementations.
- *Uniformity:* For each target architecture, the measured values have to be unified with respect to their units of measure (i.e., bytes, bytes/second). Hence, status information becomes comparable also for heterogeneous workstations.
- *Flexibility:* The NSR provides a wide variety of status information. Measurement applications can determine which subset of information has to be measured on which workstations. Current values can be requested once or regularly based on a specified time interval. Additionally, the frequency of measurements has to be freely configurable.
- *Efficiency:* The NSR can simultaneously serve as a measurement component for several applications. If possible, concurrent measurements of the same data should be reduced to a single measurement.

It is likely to happen in time-shared systems that several measurement applications execute concurrently, e.g. several on-line tools for several parallel applications execute on an overlapping set of workstations in the cluster. A severe measurement overhead on multiple observed workstations can result from an increasing frequency of periodically initiated measurements. Multiple measurements of the same status information are converted to a single measurement, thereby reducing the measurement overhead. However, synchronized data measurement demands for a mechanism to control the synchronization. We use interrupts to synchronize

[5] More exactly: when `pvm_send` is called by *any thread in* process *p_1*. However, since PVM is not multi-threaded, there is only one thread in the process.

data measurements. In case of periodic measurements, the period of time between any two measurements is at least as long as the period of time between two interrupts. However, this limits the frequency of measurement initiations and can delay the measurement application. We implemented two concepts to reduce the delays. Firstly, if a measurement application immediately requires status information, then measurements can be performed independently from the interrupt period. Secondly, we use asynchronous operations for data measurement and data access.

The architecture of the NSR follows the client-server paradigm. On each node that participates in the cluster that is used by the application programs, an NSR daemon process is installed and acts as a server for node-related measures. Any OCM monitor process belonging to the tool session of a user, can contact the local daemon and thus react to corresponding service request from the individual tools.

On order to achieve a higher degree of versatility, NSR daemons can also exchange load values between each other. Thus, in scenarios where NSR is used without OCM, a client request can collect information from a set of nodes without taking care of how to contact the individual nodes. For further details on the concepts of the NSR refer to [9].

5 Request Distribution

We will now have a closer look at the concepts used to distribute requests to individual nodes. By means of an example we will demonstrate the role of the NDU. Its task is to transfer a tool's service request into sets of local sub-requests that are handled by the individual nodes. Obviously, after having split a request into parts, the NDU is responsible for coordinating activities of these parts.

In the example we consider the situation where all processes on all nodes under observation shall be stopped when process *p_3* hits address 0x568. In this case, a tool will send the following event/action-relation as a request to the monitoring system:

```
thread_reached_addr([p_3],0x568) :
    thread_stop([])
```

Note that in the example we refer to a programming model without threads. Thus, all services starting with `thread_` refer to processes (We consider them to contain exactly one thread). As already mentioned, token lists are possibly converted before they are used. By localization and expansion, tokens are transformed until the list contains only tokens of the object type the service works on.

For the design and activities of the NDU this has several implications. It can not expand the empty list to a process list as this has to be deferred until the event actually takes place. However, it has to program the monitors of all the currently observed nodes. A solution is to partially expand the empty list to a list of node tokens and to program these nodes in an appropriate manner. In addition, as the number of observed nodes may change over time, the NDU must react to this fact. In case of new nodes it must activate appropriate observations on these nodes. In case of node deletion it must re-arrange its list of objects being involved in the execution of the above tool request[6]. We will not deal with the dynamic aspects of the NDU's activities but will discuss the composition and distribution of sub-requests.

The NDU will now program the individual monitors in an appropriate way. Its task is to activate an event detection on the node where `p_3` is located and to trigger the action list on each node when the event takes place. As in our current implementation local monitors can exclusively communicate with the NDU it also performs the task of forwarding important information between nodes.

Assume that process *p_3* is located on node *n_0* and that we altogether have three nodes: *n_0*, *n_1*, and *n_2*. Table 1 shows a list of sub-requests each of which consists of an event definition and an action list. Requests are numbered and the nodes are listed to which they refer.

The observation of *p_3*'s activities is covered by a local sub-request (1) which is sent to node *n_0*. It comprises the appropriate event detection combined with an action that does not exist at the monitoring interface's user visible abstraction level. The internal service `ct_send(`*nodelist*`,` *identifier*`)` is called cross-trigger and serves for inter-node synchronization purposes. It sends a signal[7] tagged with *identifier* to all nodes in *nodelist*. Cross-trigger signals are consumed by their corresponding `ct_recv(`*num*`,` *identifier*`)` events, where *num* defines the number of individual signals to be waited for until the event is triggered. The variable *identifier* defines the tag of the cross-trigger signal. Hence, sub-request (1) results in a signal to the NDU.

Note that in order to simplify the following description, we will say that "'we wait for a cross-trigger signal"', although the local monitor will not actually block until the signal arrives. Instead, the arrival of a cross-trigger signal triggers an event that is handled by the monitor's event detection mechanism, just like any other event that occurs in the monitored application. Thus, we don't run into problems with deadlock or delayed detection of other events.

With sub-request (2) we wait (in the above sense) for the signal generated by sub-request (1) to arrive and forward a new cross-trigger signal to a set of nodes. The semantics of this signal means that the event specified in the tool's service request took place and that appropriate activities have to be started on selected nodes. The list of nodes specified in `ct_send` comprises all nodes referred to in the action list of the tool's service request.

On all these nodes we did already install corresponding sub-requests that consume the trigger signal (3.1 - 3.3). Their action lists are composed by actions derived from the tool's request action list. As the latter specifies to stop processes so does the local action list. Note that with the tool's request we had an empty token list referring to all processes on all nodes. In the sub-request this gets substituted by the identifier of the local node. Actually, this means to stop all processes on this node. The execution of this action can be performed locally after expansion of the local node token to a process token list. The action list concludes with a cross-trigger signal to the NDU telling it that the list was executed.

With request (4) we wait for a number of cross-trigger signals to arrive. This number is defined by the number of nodes where sub-request (3._) was installed. Keep in mind that the monitoring system must be able to adapt itself to a varying number of nodes. Thus, the NDU always has to know on how many nodes sub-request (3._) is currently active. After having received the correct number of acknowledgment triggers, the NDU signals node *n_0* that it can now perform a clean-up for this event occurrence. Results, if any, are sent back to the tool (5).

Finally, the NDU also performs optimization of the sub-request composition. This refers to sub-requests 1 and 3.1.

[6] Note that with static systems like MPI-1 the above considerations are irrelevant.

[7] This is a special message and must not be confused with signals from the Unix operating system.

Table 1: Example of request distribution

No.	Nodes	Request
1	n_0	`thread_reached_addr([p_3],0x568) : ct_send([NDU],1)`
2	NDU	`ct_recv(1,1) : ct_send([],2)`
3.1	n_0	`ct_recv(1,2) : thread_stop([n_0]) ct_send([NDU],3)`
3.2	n_1	`ct_recv(1,2) : thread_stop([n_1]) ct_send([NDU],3)`
3.3	n_2	`ct_recv(1,2) : thread_stop([n_2]) ct_send([NDU],3)`
4	NDU	`ct_recv(`*num[]*`,3) : ct_send([n_0],4)`
5	n_0	`ct_recv(1,4) : event_finish()`

The latter gets activated via a cross-trigger signal from the NDU. As it is located on the same node where the event happens, it will be more efficient to append its action list directly to sub-request 1. It might be even interesting to prepend it in order to stop the process as quickly as possible. In consequence sub-request 2 should no longer send a signal to node *n_0*. Sub-request 3.1 can then be deleted. In general, the NDU must take care not to introduce unnecessary cross-trigger signals.

From this example, which in fact is simple but nevertheless exhibits local and global aspects, we can derive a list of tasks to be performed by the NDU; its title might be "divide-and-conquer":

- The NDU analyzes the tool's service request and extracts sub-requests for any node where an event occurrence has to be observed.
- By means of low-level actions for synchronization it forces the local monitors to signal event occurrences to the NDU. Thus it gains the necessary control over the request execution.
- The action list defined in the tool's service request is also distributed onto the appropriate nodes. They get activated as soon as the NDU knows about the event occurrence. Note that sub-requests never refer to objects outside the local node. Thus, all activities of a local monitor can be handled locally.
- Sub-requests must be optimized with respect to performance. Introduction of cross-trigger signals must be avoided wherever possible.

The event-action paradigm of the monitoring system gets reflected in the local monitors and the NDU. This facilitates concepts for their implementation.

6 Interoperable Tools Based on OMIS/OCM

One of the major goals of OMIS together with its implementation OCM is to support interoperable tools. This means that more than one tool can be applied to a program run at the same time and that tools are aware of each other. The degree of awareness could be subdivided into not disturbing each other and cooperating with one another. Both cases shall be covered by OMIS although a support for explicit tool cooperation will only be integrated in the next release.

What are the benefits of the concept of interoperability? First, when considering individual tools from different vendors, we find that we would sometimes like to use more than one tools at a time. E.g. if we find a performance flaw we would like to switch to a debugger to inspect appropriate variables in the source code. Second, we find automatic tools that are incompatible with interactive tools. E.g. with load balancing (performed by process migration) being active, debugging and performance analysis can no longer be applied.

If we consider the example of a combination of automatic and interactive tools in more detail we find several interesting scenarios where we would like to apply this. The combination of performance analysis and load balancing by process migration is one such combination. Interoperability would imply that both tools are aware of each other. If they just coexist without direct cooperation we can already identify new functionality for the performance analyzer: it could visualize the activity of the load balancer and could either hide migrations from the user or make them explicit to him. The user could thus evaluate the program performance on this target architecture (having load balancing as a basic mechanism) or could evaluate the influence of load balancing on the performance of the program. According to the overall optimization goals a switch in the interactive tool can select one or the other functionality

By applying OMIS/OCM for tool construction, tool developers can now provide these types of functionality. It results from the synergy of the individual tools' functionality being coordinated via the interface of the monitoring system.

7 Project Status and Future Work

A first implementation of the OCM, as it is described in Section 4, has already been finished. It is currently supporting PVM 3.3.x and PVM 3.4 on networks of workstations running Solaris, Linux, and Irix. However, some of the services defined by OMIS still need to be implemented. In order to test the OCM, the Tool-set debugger has been adapted to the OMIS interface and is already operational.

Our current work includes finishing a machine-independent interface between the OCM and the monitoring support offered by the nodes' operating systems, which will increase the OCM's portability. In addition, we are working on the specification of an interface allowing extensions of OMIS compliant monitoring systems independent of the individual implementation. This interface will be supported by a stub generator that eases the programming of new events and actions. A first version of the interface and the corresponding stub generator has already been finished.

Future work will focus on adapting the OCM to the MPI implementation `mpich`, and on porting it to the operating system Windows NT. In addition, we will use the OCM as the basis for providing interoperable tools within The Tool-set project[12]. Finally, as a more long term goal, we will implement the OCM also for distributed shared memory systems, both based on hardware support and pure software implementations.

8 Acknowledgments

We would like to thank Manfred Geischeder, Michael Uemminghaus, and Hans-Günter Zeller for their participation in the design phase of the OCM implementation. Alexander Fichtner, Carsten Isert, and Andreas Schmidt are still busy with implementing the OCM. We appreciate their support. Special thanks to Michael Oberhuber and Vaidy Sunderam who have been heavily involved in discussing the OMIS specification in its first phase, and to all other people that provided feedback on OMIS. We also like to thank the anonymous referees who pointed out several unclear issues in the draft paper.

References

[1] J. K. Hollingsworth and B. Buck. DynInstAPI Programmer's Guide Release 1.0, Sept. 1997.

[2] J. K. Hollingsworth and B. P. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. In *International Conference on Supercomputing*, Tokio, July 1993.

[3] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *1994 Scalable High-Performance Computing Conference*, pages 841–850, Knoxville, TN, Mai 1994.

[4] R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127–136, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.

[5] R. Hood and D. Cheng. Accomodating Heterogeneity in a Debugger - A Client-Server Approach. In *Proc. of the 28th Annual Hawaii International Conference on System Sciences, Volume II*, pages 252–253. IEEE, Jan. 1995.

[6] P. Kacsuk, J. Cunha, G. Dzsa, J. Lourenco, T. Antao, and T. Fadgyas. GRADE: A Graphical Development and Debugging Environment for Parallel Programs. *Parallel Computing Journal*, 22(13):1747–1770, Feb. 1997.

[7] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

[8] B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithap, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 11(28), November 1995.

[9] C. Röder, T. Ludwig, and A. Bode. Configurable load measurement in heterogeneous workstation clusters. In *Proceedings of the Europar98*, 1998. (Accepted as distinguished paper; to appear).

[10] R. M. Stallman and R. H. Pesch. *Using GDB, A Guide to the GNU Source Level Debugger, GDB Version 4.0.* Free Software Foundation, Cygnus Support, Cambridge, Massachusetts, July 1991.

[11] 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 95-162, Center of Excellence in Space Data and Information Sciences, NASA Goddard Space Flight Center, Greenbelt, Maryland, 1995.

[12] R. Wismüller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Röder, and G. Stellner. THE TOOL-SET Project: Towards an Integrated Tool Environment for Parallel Programming. In *Proc. Second Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97*, pages 9–16, Koblenz, Germany, Sept. 1997.

[13] R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415–442, Mar. 1996.