

OMIS v2.0

A Universal Interface for Monitoring Systems

Thomas Ludwig, Roland Wismüller

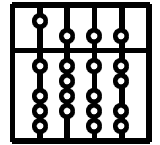
LRR-TUM

Institut für Informatik

Technische Universität München, Germany

e-mail: ludwig@informatik.tu-muenchen.de

<http://wwwbode.informatik.tu-muenchen.de/~omis>



Motivation

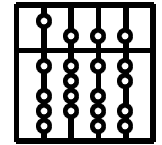
Powerful tools for parallel and distributed programming need on-line monitoring facilities for **observation** and **manipulation** of programs during runtime.

E.g. debugging, performance analysis, load balancing, computational steering ...

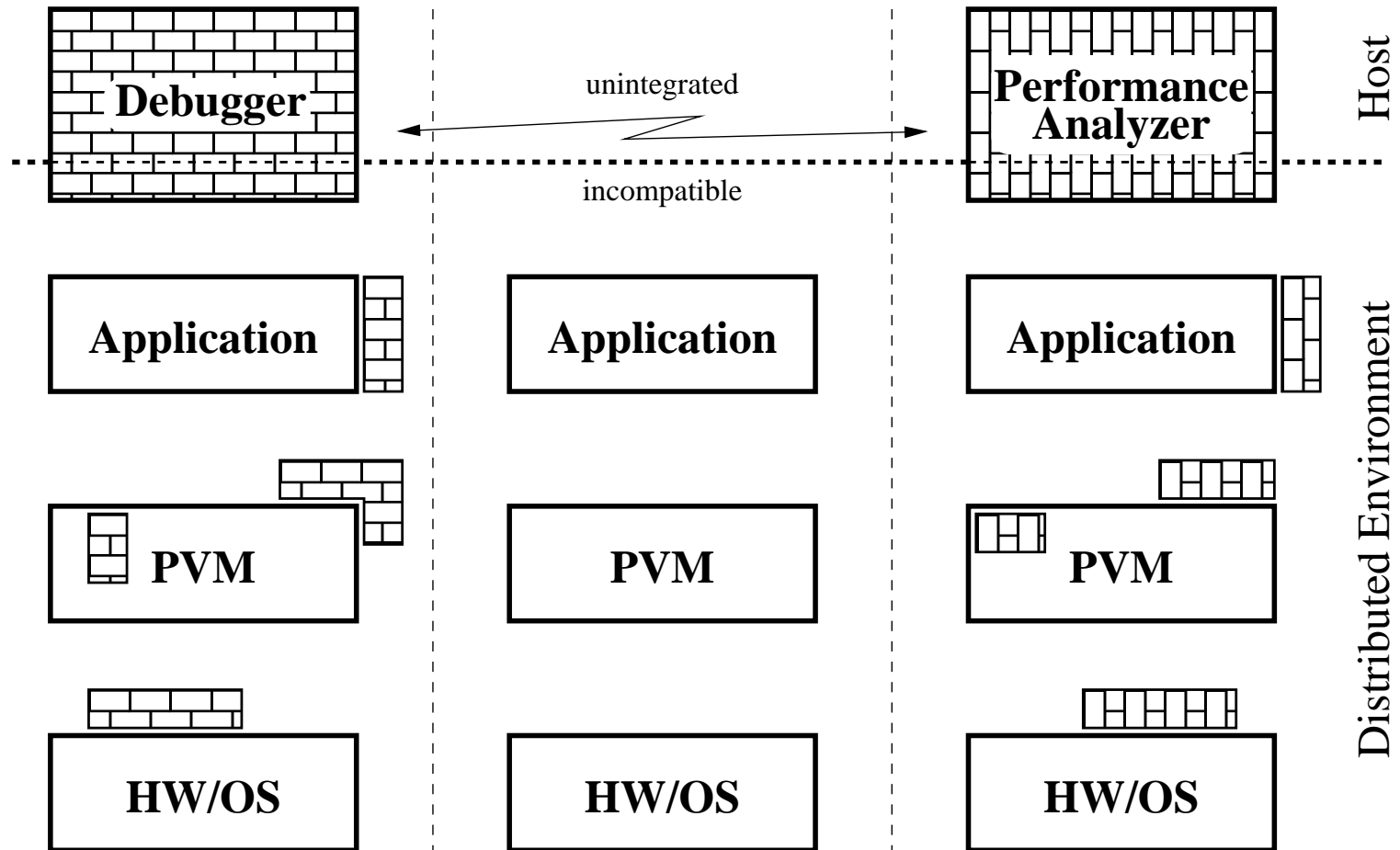
Sophisticated tools already exist, **but**:
all use proprietary monitoring systems which are incompatible to each other!

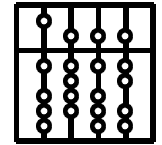
No standard infrastructure exists to connect tools to running systems

- ⇒ every new tool requires to implement a new monitoring system
- ⇒ no interoperable tools
- ⇒ no uniform tool environments



Conventional Tool Environments

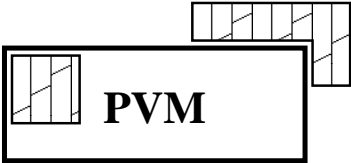


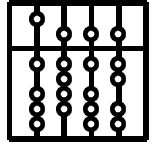


OMIS-Based Tool Environments



OMIS-On-line Monitoring Interface Specification





OMIS v2.0 — On-line Monitoring Interface Specification

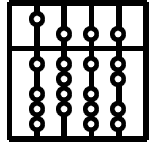
OMIS has been started in Summer 1996 as a joint project between Emory University and LRR-TUM.

Project Goals:

- define a universally usable on-line monitoring interface
- separate tool development from monitoring system development
 - ⇒ R&D groups involved in tool design and implementation
 - ⇒ R&D groups involved in monitor design and implementation
- provide basis for interoperable tools and uniform tool environments

OMIS v1.0 finished in Feb. 1996

Design of an OMIS implementation and feedback from other research groups lead to revised version 2.0



Requirements on a Universal Monitoring Interface

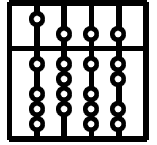
The interface should ideally be

- usable for any kind of run-time tools
(interactive/automatic, centralized/distributed)
- independent of the target platform
(hardware, operating system, programming library)
- implemented efficiently with minimal intrusion

~> To meet all of these requirements at the same time is hard!

Approach:

- provide a very flexible interface
- provide means to extend the interface in an easy and well-defined way



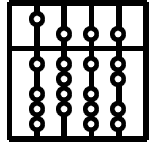
Flexibility and Extendibility

Flexibility:

- interface for centralized and distributed tools
- event-action paradigm:
 - request consists of event definition and action list
 - events needed by different tools are fairly the same
 - actions may differ \Rightarrow extensions for specific tools

Extendibility:

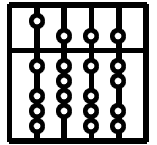
- distributed tool extensions
define derived events and new actions, platform independent
- monitor extensions
define new objects and new events, platform dependent



Flexibility

Support for different kinds of tools

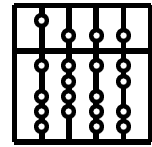
- ⇒ Interface is suitable for centralized and distributed tools based on location independent (possibly remote) procedure calls
- ⇒ Event-action paradigm: each request consist of two parts:
 - **event definition** specifies *when* something has to be done
 - **action list** specifies *what* has to be done
- event definition may be empty ⇒ Action list is executed immediately
- ↪ events needed by different tools are fairly the same, e.g. communication, process creation, ...
- ↪ actions may be different ⇒ OMIS provides a standard set of actions that can be extended by each tool
- ↪ parameters describing an event occurrence may be passed to the actions (e.g. node, process where event occurred, time stamp)



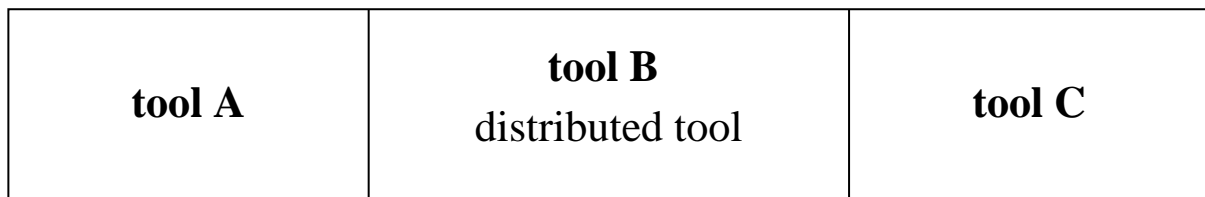
Extendibility

Two types of extensions with different qualities:

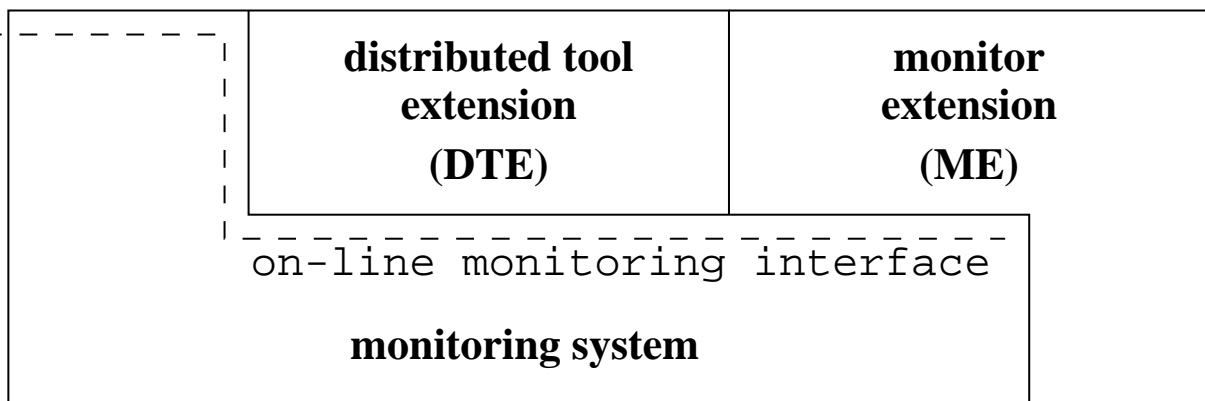
- Distributed tool extensions
 - parts of a tool integrated into monitoring system
 - new events derived from existing ones (filters)
 - tool-specific actions (e.g. writing a trace)
 - independent of target system and OMIS implementation
 - ↪ OMIS defines both external interface of extension and internal interfaces usable for extension
- Monitor extension
 - new kinds of objects and services for these objects (e.g. PVM groups)
 - may depend on target system and OMIS implementation
 - ↪ OMIS defines only external interface of extension



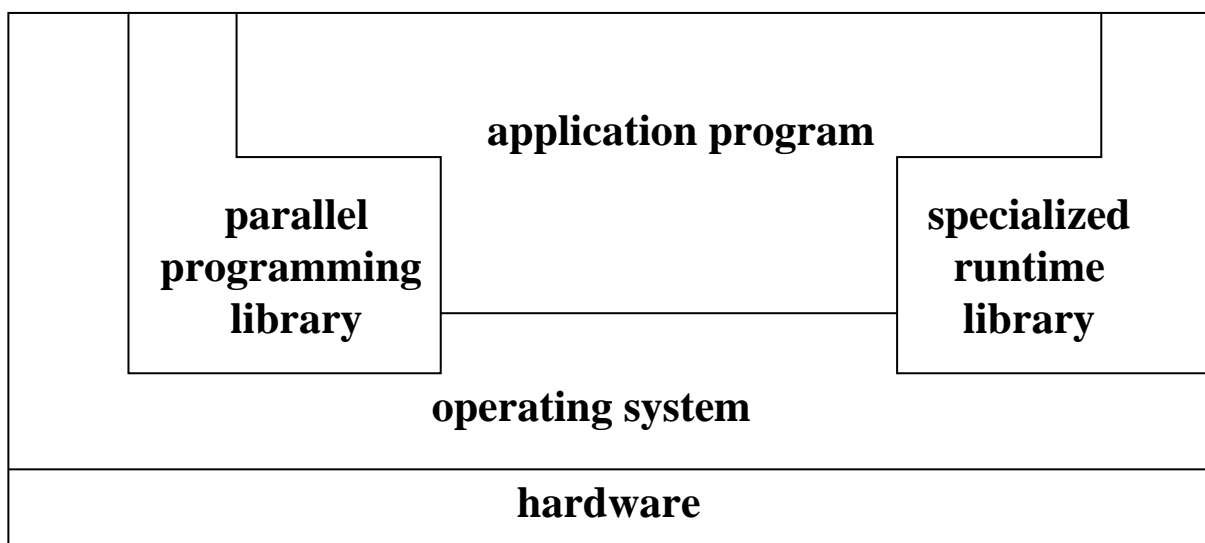
System Model

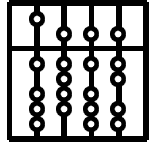


----- tool/monitor-interface -----



----- monitor/program-interface -----





The Monitoring Interface

Interface procedure (usable by tools and extensions):

```

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

```

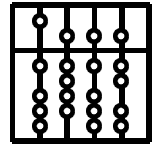
General syntax of requests:

```

request          ::= [ event_definition ] : action_list
event_definition ::= service_name ( parameters )
action_list      ::= action | action [ ; ] action_list
action           ::= service_name ( parameters )

```

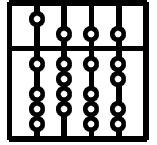
An action list may execute concurrently; a semicolon acts as a global synchronization (barrier).



Enhancements in OMIS v2.0

Feedback on OMIS:

- OMIS v1.0 is very dependent on PVM and NOWs
 - ⇒ new object based model of the observed system (threads, SMPs)
 - ⇒ each tool defines the observed system explicitly
 - ⇒ all PVM specific details are now in an extension
- Explicit specification of nodes is often cumbersome
 - ⇒ location transparency
- Reply string is a source of inefficiency
 - ⇒ reply is more structured now
- Services are not completely specified
 - ⇒ OMIS v2.0 contains full specifications (optional and required parts)



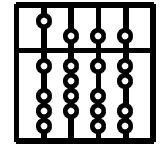
Support for different target platforms

Object based approach

- Abstract objects: nodes, processes, threads, messages, and message queues
- Hardware and process model suitable for DMPs, NOWs, SMPs, and clusters of SMPs
- Identification by object handles, automatic conversions (localization, expansion)
- OMIS core only defines platform independent services (others: platform specific extensions)

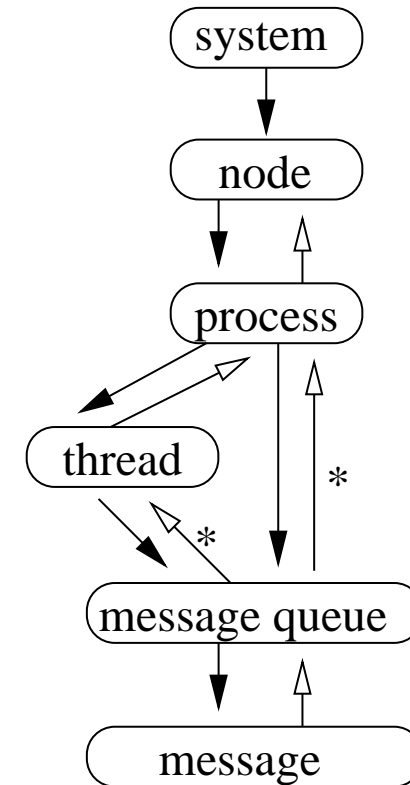
Location transparency

- Events and actions work on any sets of objects
- Tools *need* not care about the location of these objects

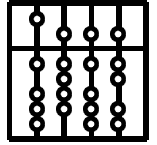


Model of Observed System

- System consists of a hierarchy of five object classes
- Each tool has its private view
 - nodes and processes must be attached explicitly (extensions may provide necessary information)
 - other objects are attached automatically
- ⇒ enhanced tool interoperability
- ⇒ independent of programming library
- Object expansion results in a list of all contained objects the tool is attached to.



↓ expansion ↑ localization
 * depends on target platform



Services Defined by the OMIS Core

1. Nodes

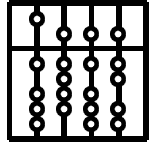
- addition/deletion of nodes, ...
- attach, detach, static information (e.g. number, names), dynamic information (e.g. load) ...

2. Processes

- creation, termination, receipt of signals, ...
- create, attach, detach, send signal, write memory, static and dynamic process information, ...

3. Threads

- creation, termination, stopped, continued, execution of statements, execution of library calls, ...
- detach, stop, continue, goto address, static and dynamic thread information, stack backtrace, read registers, ...



Services Defined by the OMIS Core

4. Messages and message queues

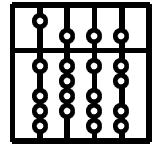
- insertion into message queue, receipt of tagged message, ...
- insert/delete message, tag message, queue information, ...

5. Monitor objects

- user defined events, ...
- define/raise user events, enable/disable requests, extensions, ...

PVM extension

- get nodes/tasks in VM, pack/unpack messages, group information, ...



Example: Debugging

```
thread_has_started_lib_call([], "pvm_send") :
  proc_get_info([$proc], 12) thread_get_backtrace($thread, 1)
```

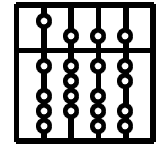
Whenever a process calls `pvm_send`, return its `tid`, its argument vector, and the return address of the call

```
thread_reached_addr([p_12, p_34, p_44], 0xfe08) : thread_stop([])
```

When one of the specified processes reaches the breakpoint, stop the whole application

```
thread_creates_proc([p_34]) : proc_attach($new_proc) thread_stop($new_proc)
```

Debug (i.e. attach and stop) new processes dynamically created by process `p_34`



Example: Performance Analysis

```
thread_has_started_lib_call([p_21],"MPI_Send") :
  pt_integrator_start(pt_i_1) pt_counter_add(pt_c_1,$par5)
```

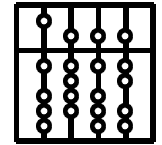
When process p_21 starts a send call, start an integrating timer and add the message size (event context parameter par5) to a counter

```
thread_has_ended_lib_call([p_21],"MPI_Send") : pt_integrator_stop(pt_i_1)
```

When the process has finished the call, stop the timer again

Result: time spent in send calls and amount of data being sent.

Note: the actions in this example come from a distributed tool extension. Later versions of OMIS may include similar services in the core.

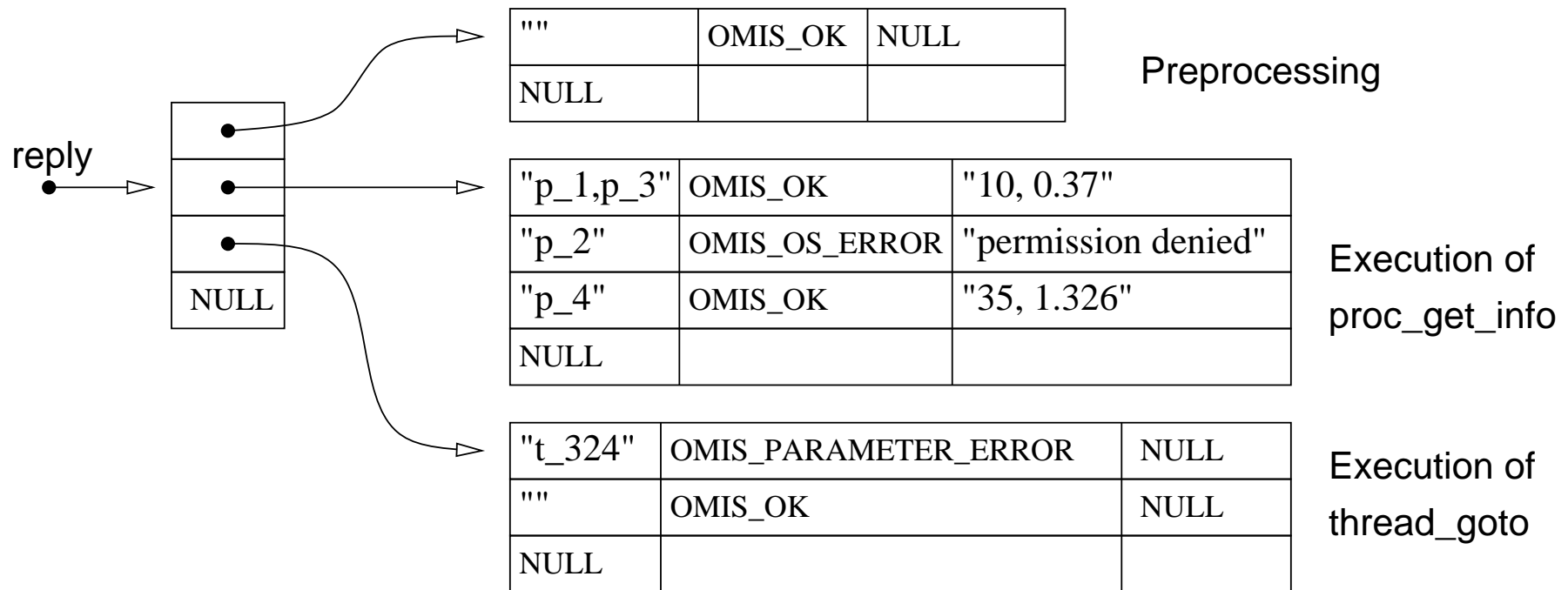


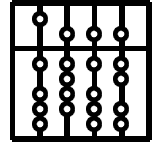
Service Replies

Replies are structured according to the request's syntactical structure, e.g. for a request

: `proc_get_info([],12288) thread_goto([],0x4304)`

the reply could be:





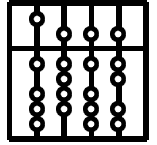
Error Handling

Three alternatives:

1. Abandon execution when first error is detected, return single status value
 - + easy to implement
 - state of system may be inconsistent and is unknown

 2. Requests behave as transactions
 - + system state is always consistent
 - inefficient (or even impossible) to implement

 3. Single services on single objects behave as transactions, if possible. A status value is provided for each service and object
 - + relatively easy to implement
 - + system state may be inconsistent, but is known (except for fatal errors)
- ⇒ choice of OMIS



OCM: An Implementation of OMIS

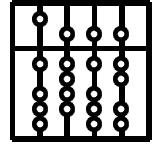
OCM = OMIS compliant monitoring system

Implementation of OMIS for PVM on clusters of workstations

Project goals:

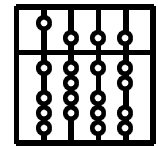
- Verify the practicability of OMIS \Rightarrow feedback
- Provide an implementation platform for THE TOOL-SET
- Provide an implementation platform for other tool environments
- Prepare a reference for other OMIS implementations

OCM will be released under the GNU license conditions!

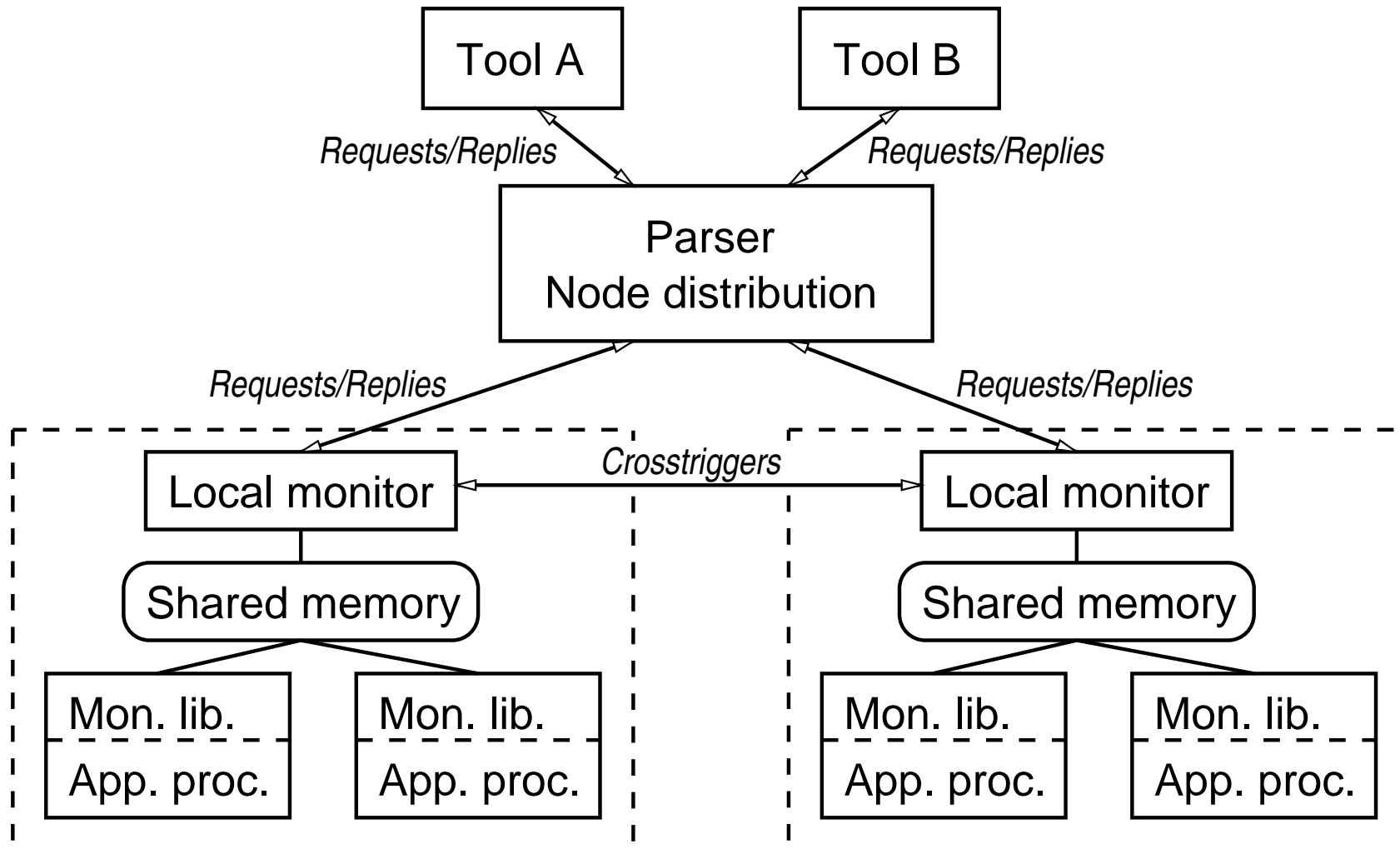


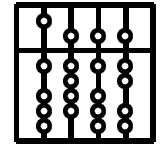
Design Concepts

- The monitoring system is a set of communicating monitors, one on each node attached by some tool
- Global requests are distributed to target nodes at definition time; automatic insertion of crosstrigger events and actions
- Internode communication is based on PVM
- For the first implementation: centralized parsing and distribution unit
- Single node monitors don't see the distributed environment
- Event based implementation of single node monitors
- Event detection and action execution both in the monitor process and in the application processes
- Intranode communication uses shared memory segments

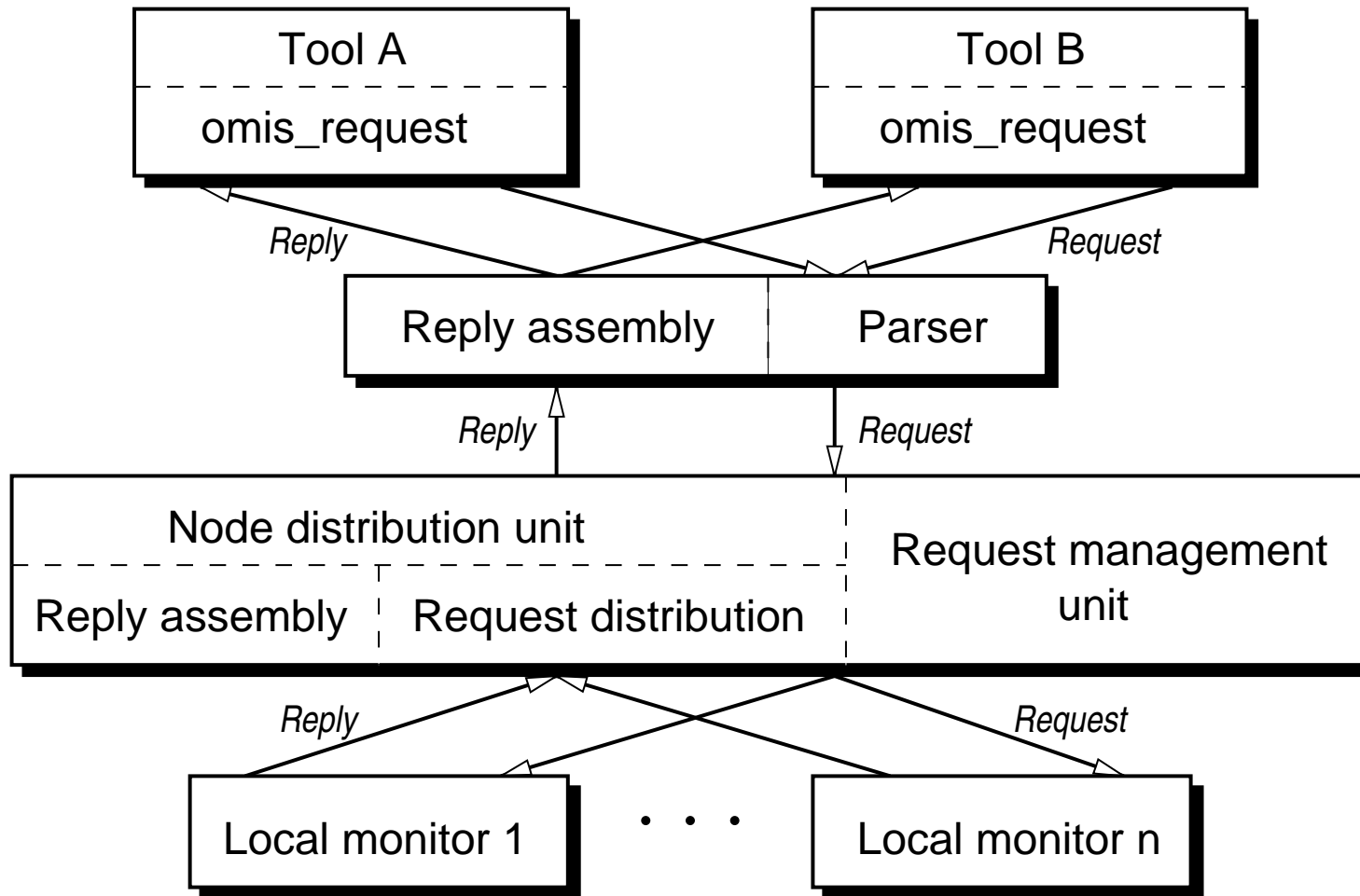


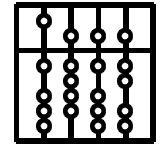
Coarse Grain Structure of OCM





Coarse Grain Structure of OCM



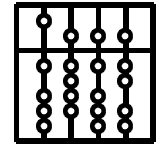


Request Distribution

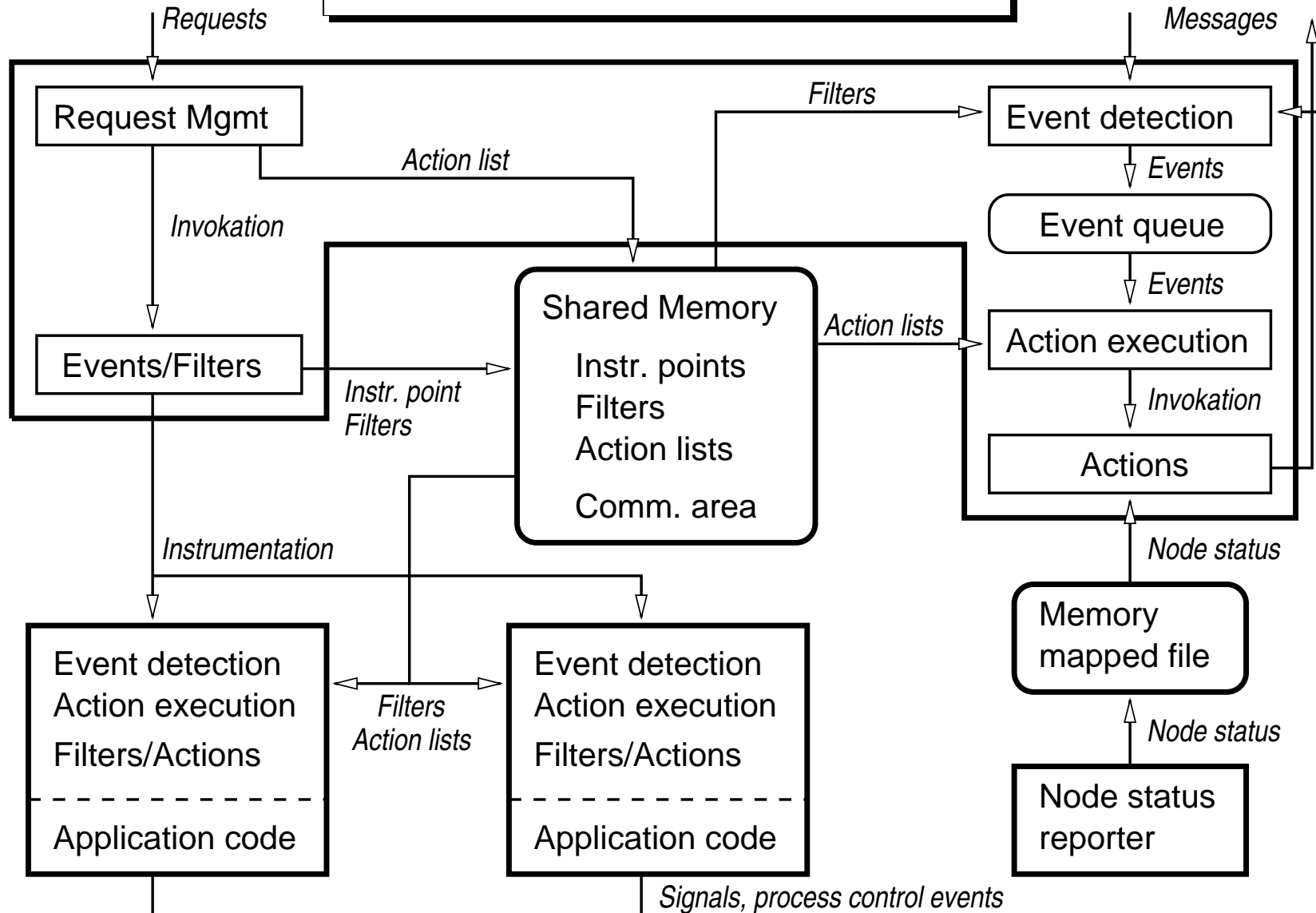
Example: global breakpoint

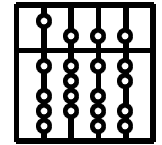
```
thread_reached_addr([p_3],0x568) :
    thread_get_backtrace($proc, 0) thread_stop([])
```

Node 0	Node 1,2,3	NDU
<pre>thread_reached_addr([p_3],0x568) : ct_send([NDU],1) ct_rcv(1,2) : thread_get_backtrace(\$proc, 0) thread_stop([n_0]) ct_send([NDU],3) ct_rcv(1,4) : event_finish()</pre>	<pre>ct_rcv(1,2) : thread_stop([locnode]) ct_send([NDU],3)</pre>	<pre>ct_rcv(1,1) : ct_send([],2) ct_rcv(num[],3) : ct_send([n_0],4)</pre>



Structure of Local Monitor

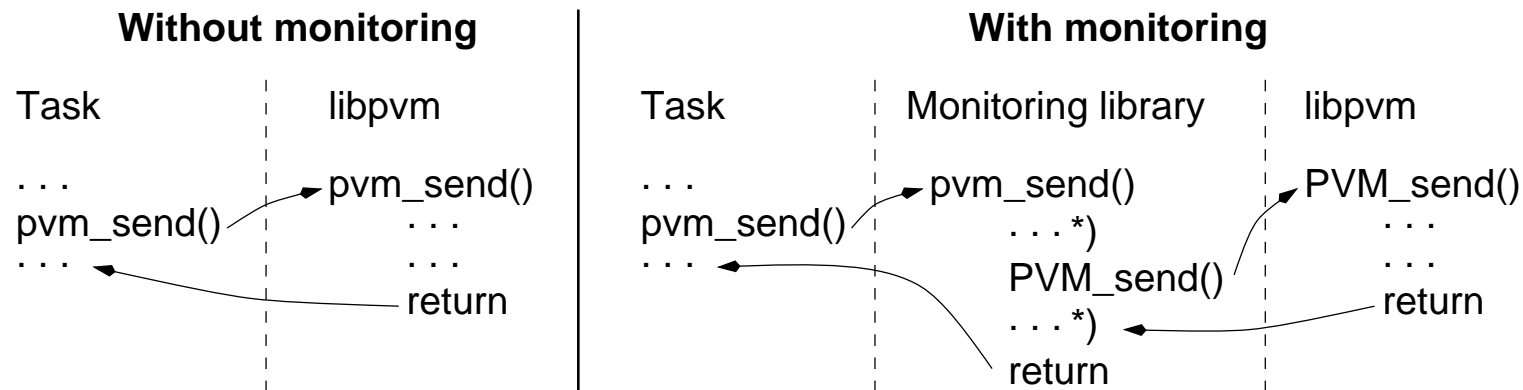




Instrumentation Techniques

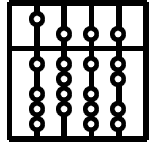
Events related to programming library:

- Detected in application processes
- Binary wrapping
rename some library functions, link new code in between these functions and their calls



*) = if (hook_active[<hook no>]) ...

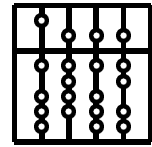
- Dynamic instrumentation ?
patch actions directly into binary program, no need to relink



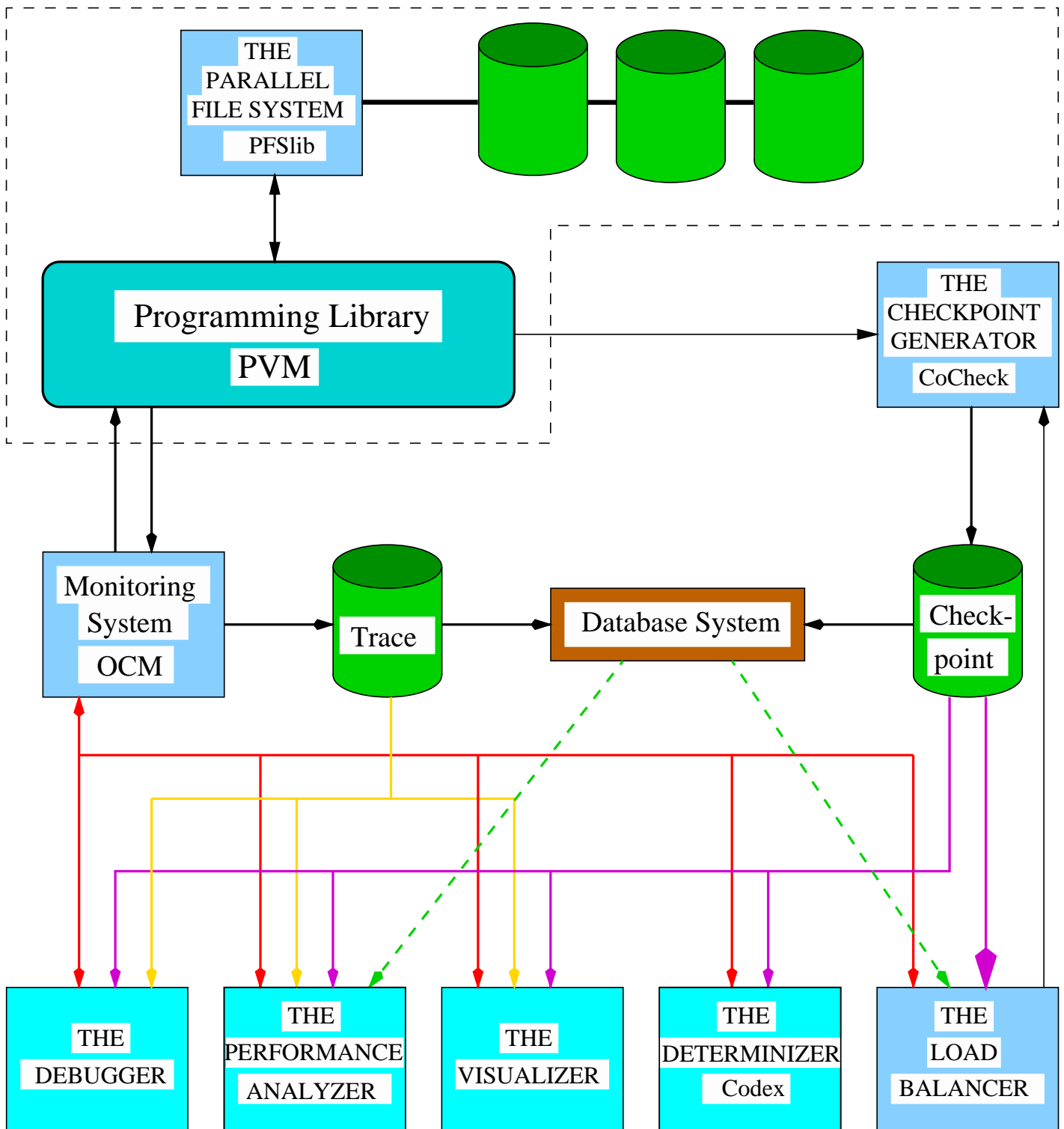
Instrumentation Techniques

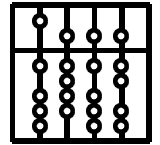
Other events:

- Detected in monitor process
- Process control events:
 - `ptrace` and `wait` system calls
 - `/proc` file system currently not used:
 - * very different on different platforms (e.g. Solaris, Linux)
 - * problem: handling of multiple processes
- Other events: signals and traps
- Message receipts: non-blocking via signals



Application of OCM: THE TOOL-SET for PVM





Project Status

OMIS

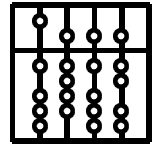
- Version 1.0 released February 1996
- Major update: Version 2.0 released July 1997

OCM

- Implementation started January 1997 (3 researchers and 4 students)
- First Implementation: July 1997
- First tools of THE TOOL-SET: October 1997

Cooperations

- ENS Lyon, France (Dosmos)
- KFKI-MSZKI Research Institute, Budapest, Hungary (GRADE)
- GUP, Univ. of Linz, Austria (MAD)



Future Work

- Finish implementation of OCM and adaptation of tools
- Optimize OCM
- Integrate new tools: checkpointing, load balancing, deterministic execution
- Integrate feedback into OMIS document
- OMIS for distributed shared memory systems
- Further discussions with other research groups

Any feedback is welcome, please look at:

<http://www.bode.informatik.tu-muenchen.de/~omis>