Configurable Load Measurement in Heterogeneous Workstation Clusters

Christian Röder, Thomas Ludwig, Arndt Bode

LRR-TUM — Lehrstuhl für Rechnertechnik und Rechnerorganisation
Technische Universität München, Institut für Informatik, D-80290 München
{roeder|ludwig|bode}@informatik.tu-muenchen.de

Abstract. This paper presents the design and implementation of NSR– the Node Status Reporter. The NSR provides a standard mechanism for measurement and access to status information in clusters of heterogeneous workstations. It can be used by any application that relies on static and dynamic information about this execution environment. A key feature of NSR is its flexibility with respect to the measurement requirements of various applications. Configurability aims at reducing the measurement overhead and the influence on the observed cluster.

1 Motivation

During the last years, clusters of time-shared heterogeneous workstations became a popular execution platform for parallel applications. Parallel programming environments like PVM [5] and MPI [13] have been developed to implement message passing parallel applications. Late development stages and efficient production runs of parallel applications typically require sophisticated on-line tools, e.g. performance analysis tools, program visualization tools, or load management systems. A major drawback of existing tools like e.g. [1, 7, 9, 12] is that all of them implement their own monitoring technique to observe and/or manipulate the execution of a parallel application. Proprietary monitoring solutions have to be redesigned and newly implemented when shifting to a new hardware platform and/or a new programming environment.

In 1995, researchers at LRR-TUM and Emory University started the OMIS[1] project. It aims at defining a standard interface between tools and monitoring systems for parallel and distributed systems. Using an OMIS compliant monitoring system, tools can efficiently observe and manipulate the execution of a message passing based parallel application. Meanwhile, OMIS has been redesigned and is available in version 2 [8]. At the same time, the LoMan project [11] has been started at LRR-TUM. It aims at developing a cost-sensitive system-integrated load management mechanism for parallel applications that execute on clusters of heterogeneous workstations. Combining the goals of these two projects, we designed the NSR [6] that is a separate module to observe the status of this execution environment.

[1] OMIS: On-line Monitoring Interface Specification.

The goal of NSR is to provide a uniform interface for *data measurement* and *data access* in clusters of heterogeneous workstations. Heterogeneity covers two aspects: different performance indices of the execution nodes and different data representations due to different architectures. Although primarily influenced by the measurement requirements of on-line tools, the NSR serves as a powerful and flexible measurement component for any type of application that requires static and/or dynamic information about workstations in a cluster. However, the NSR does not post-process and interpret status information because the semantics of measured values is known by measurement applications only. In the following, we will use *measurement application* to denote any application that requires status information.

The document is organized as follows. The next section presents an overview of related work. In § 3 we propose the reference model of NSR and discuss the steps towards its design. Implementation aspects are presented in § 4. The concept of NSR is evaluated in § 5. Finally, we summarize in § 6 and discuss future development steps.

2 Related Work

Available results taken into consideration when designing and implementing the NSR can be grouped into three different categories:

- Standard commands and public domain tools (e.g. `top`[2] and `sysinfo`[3]) to retrieve status information in UNIX environments.
- Load modeling and load measurement by load management systems for parallel applications that execute in clusters of heterogeneous workstations (e.g. load monitoring processes [4, 7]; classification of load models [10]).
- The definition of standard interfaces to separately implement components of tools for parallel systems (e.g. OMIS [8], UMA[4] [14], SNMP[5] [2], and PSCHED [3]).

Typically, developers implement measurement routines for their own purposes on dedicated architectures or filter status information from standard commands or public domain tools. An overview on commands and existing tools is presented in [6]. The implementation of public domain tools is typically divided into three parts. A textual display presents the front-end of the tools. A controlling component connects the front-end with measurement routines that are connected by a tool-defined interface. Hence, porting a tool to a new architecture is limited to implement a new measurement routine. However, most tools only provide status information from the local workstation. Routines for transferring measured data have to be implemented if information is also needed from remote workstations.

[2] `top`: available at ftp://ftp.groupsys.com/pub/top.
[3] `sysinfo`: available at ftp://ftp.usc.edu/pub/sysinfo.
[4] UMA: Universal Measurement Architecture.
[5] SNMP: Simple Network Management Protocol.

The second category taken into considerations covers load modeling requirements of load management systems. A load management system can be modeled as a control loop that tries to manage the load distribution in a parallel system. Developers of load management techniques often adopt measurement routines to retrieve load values from the above mentioned public domain tools. However, it is still not yet clear which status information is best suited for load modeling in heterogeneous environments. A flexible load measurement component can serve as a basis to implement an adaptive load model [10]. Furthermore, different combinations of status information could be tested to find the values that can adequately represent the load of heterogeneous workstations.

Standard interfaces are defined to allow separate development of tool components. Tool components can subsequently cooperate on the basis of well-defined communication protocols. The exchange of one component by another component (even from different developers) is simplified if both are designed taking into account the definitions of such interfaces. We already mentioned the OMIS project. In § 4.2 we will see that OMIS serves as the basis to define the status information provided by NSR.

The UMA project addresses data collection and data representation issues for monitoring in heterogeneous environments. Status information is collected independently of vendor defined operating system implementations. The design of UMA includes data collection from various sources like e.g. operating systems, user processes, or even databases with historical profiling data. Although most measurement problems are addressed, the resulting complexity of UMA might currently hamper the development and widespread usage of UMA compliant measurement systems. Furthermore, the concept does not strictly distinguish between data collection and subsequent data processing.

The PSCHED project aims at defining standard interfaces between various components of a resource management system. The major goal is to separate resource management functionality, message passing functionality of parallel programming environments, and scheduling techniques. Monitoring techniques are not considered by PSCHED at all. The PSCHED API is subdivided into two parts: 1) a task management interface defines a standard mechanism for any message passing library to communicate with any resource management system; 2) a parallel scheduler interface defines a standard mechanism for a scheduler to manage execution nodes of a computing system regardless of the resource manager actually used. By using a configurable measurement tool, the implementation of a general-purpose scheduler can be simplified.

3 The Design of NSR

The design of a flexible measurement component has to consider the measurement requirements of various applications. A measurement component serves as an *auxiliary module* in the context of an application's global task. Hence, the NSR is designed as a *reactive system*, i.e., it performs activities on behalf of the supported applications. According to the main goal of NSR to provide a uniform interface for *data measurement* and *data access*, we consider three questions:

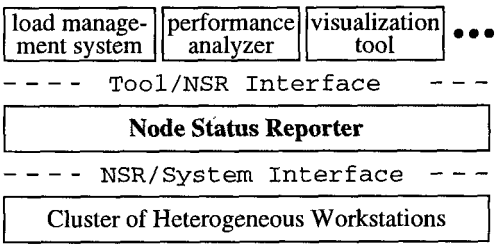

Fig. 1. The Reference Model of NSR.

1. *Which data is needed?* Different applications need different types of status information. Status information can be classified into *static* and *dynamic* depending on the time-dependent variability of its values. Static and dynamic information is provided for all major resource classes, i.e., operating system, CPU, main memory, secondary storage, and communication interfaces. Interactive usage of workstations in a time-shared environment gains increasing importance when parallel applications should be executed. Hence, the number of users and their workstation utilization are measured as well.
2. *When is the data needed?* On-line tools like e.g. load management systems, often have a cyclic execution behavior and measurements are *periodically* initiated. In this case, the period time between two successive measurements depends on the intent of measurement applications. However, measurement applications can have other execution behaviors as well. Now, status information can be measured *immediately and exactly once* as soon as it is needed.
3. *From which set of nodes is the data needed?* On-line tools can be classified regarding their implementation as distributed or centralized. In case of a central tool component (e.g. the graphical user interface of a program visualization tool), status information of multiple workstations is typically needed at once. Status information of any workstation is accessible on any other workstation.

Different applications might need the same type of status information, although their global tasks are different. For example, a load management system can consider I/O statistics of parallel applications and maps application processes to workstations with high I/O bandwidth. The same status information can be used by a visualization tool to observe the I/O usage statistics in a workstation cluster. The following section presents the reference model of NSR before we list the design goals in detail.

3.1 The Reference Model

The reference model of NSR is illustrated in Fig. 1. The NSR builds a layer between measurement applications and a cluster of heterogeneous workstations. It supports on-line tools for parallel applications like e.g. load management systems, performance analysis tools, or visualization tools. However, any type of

measurement application can utilize the NSR. Applications communicate with the NSR via the `Tool/NSR Interface`. Using this interface, they can manage and control the behavior of NSR. Recall that the NSR is designed as a reactive system on behalf of the measurement applications. The applications are responsible to configure the NSR with respect to their measurement requirements, i.e., the type of status information and the time at which measurements have to be initiated. The interface provides uniform access to information of any workstation that is observed by NSR. If necessary, the NSR distributes measured values between the workstations with respect to the applications' location.

The `NSR/System Interface` connects the NSR to a cluster of heterogeneous workstations and hides the details of workstation heterogeneity. The main purpose of this interface is to connect the NSR and measurement routines for various architectures. Furthermore, the interface provides a uniform data representation with unity scale. Developers are free to implement their own data measurement and data collection routines.

3.2 Design Goals

The following major objectives have to be pursued during the design of NSR with respect to the above listed measurement 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, etc.). Hence, status information becomes comparable also for heterogeneous workstations.

Scalability: Tools for parallel system typically need status information from a set of workstations. An increasing number of observed workstations must not slow down these applications.

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. 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.

3.3 Design Aspects

According to the goals and the requirements we consider the following aspects:

Data measurement: The measurement of a workstation's status information issues system calls to the operating system. It is one of the main tasks of the operating systems to update these values and to keep them in internal management tables. System calls are time consuming and slow down the calling application. Additionally, different architectures often require different measurement

routines to access the same values. For portability reasons, data measurement is decoupled from an application's additional work and performed asynchronously to it. The set of data to be measured is freely configurable by a measurement application.

Synchronization: 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 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.

Asynchronous Operations: Asynchronous operations are used to hide latency and overlap data measurement and computation. An identifier is immediately returned for these operations. A measurement application can wait for the completion of an operation using this identifier. Measurements are carried out by NSR, while applications can continue their execution.

Data Distribution: Apart from data measurement, the status information is transparently transferred to the location of a measurement application. A data transfer mechanism is implemented within the NSR. Again, data transfer is executed asynchronously while applications can continue their execution.

4 Implementation Aspects

4.1 Client-Server Architecture

The architecture of NSR follows the client-server paradigm. The client functionality is implemented by the *NSR library*. A process that has linked the NSR library and invokes library functions is called *NSR client* (or simply *client*). An NSR client can request service from the *NSR server* (*server*) by using these library functions. The NSR server can be seen as a global administration instance that performs data measurement and data distribution on behalf of the clients. It is implemented by *a set of replicated NSR server processes* that cooperate to serve the client requests. The server processes are intended to execute in the background as other UNIX processes. To clarify our notation we call them *NSR daemon processes* (or simply *daemons*). A single NSR daemon process resides on each workstation in the cluster for which an implementation exists.

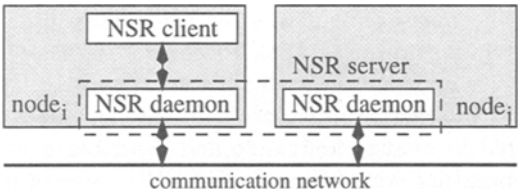

Fig. 2. The Client-Server Architecture of NSR.

An exemplary scenario of this architecture is illustrated in Fig. 2. One NSR daemon process resides on workstation $node_i$, while a second daemon resides on workstation $node_j$. The client on $node_i$ can only communicate with the local NSR daemon process. Applications can not utilize the NSR server if a local NSR daemon process is not available. The `Tool/NSR Interface` is implemented on top of this interconnection.

The local NSR daemon process controls the execution of a client request. From the viewpoint of a client, its local NSR daemon process is called *NSR master daemon*. If necessary, the NSR master daemon forwards a request to all daemons that participate in serving a client's request. The latter daemons are also called *NSR slave daemon processes*. A daemon is identified by the name of the workstation on which it resides. Note, the attributes *master* and *slave* are used to distinguish the roles each daemon plays to serve a client request. However, the attributes are not reflected in the implementation. An NSR daemon process can be both master and slave because several clients can request service from the NSR server concurrently. The main difference between master and slave daemons is that an NSR master daemon keeps additional information about a requesting client. Slave daemons do not know anything about the client that performed the request. The `NSR/System Interface` connects the workstation specific measurement routines with the NSR daemon processes. Daemons exchange messages via the communication network.

All information needed by the NSR server to serve the client requests are stored by the NSR daemon processes. Contrarily, a client is stateless with respect to data measurement. The result of a previously initiated library call does not influence the execution of future library calls. Hence, no information has to be stored between any two calls.

4.2 Status Information

The data structure `nsr_info` is the container for the status information of workstations and is communicated between the NSR components.

```
struct nsr_info { long                mtag, tick;
                  static_hostinfo         stthi;
                  dynamic_hostinfo        dynhi; };
```

The first entry `mtag` is a sequence number for the status information. Applications can identify old and new values by comparing their `mtag` values. The second entry `tick` is a timestamp for the information. It indicates the interrupt

number at which the measurement was carried out. In case of periodic measurements, a master daemon uses `tick` to control data assignment if several clients request status information with different periods of time. Information about resources and interactive usage are split into two data structures. Static information is stored in `static_hostinfo` and describes a workstations e.g. in terms of type of operating system, number of CPUs, size of main memory, etc. Benchmark routines are used to determine performance indices of workstation resources. Dynamic information is stored in `dynamic_hostinfo` and represents the utilization values of workstation resources. For detailed information, we refer to the information service **node_get_info** in the OMIS document[6] [8, pp. 69 ff.]. The definition of this service matches the definition of these two data structures.

4.3 NSR Library Functions

The NSR library functions are split into two major groups. The first group covers *administrative functions* to manage the execution behavior of the NSR server. The second group includes *data access routines* to retrieve status information that has been measured by the daemons. Additional functions primarily ease the usage of the NSR.

Administrative functions are needed to configure the NSR server. They are available as *blocking* or *non-blocking* calls. The following example illustrates the difference between these two types. Initially, a client attaches itself to a set of NSR daemon processes with `nsr_iattach()`, that is a non-blocking call. The local daemon analyzes the parameters of the request, performs authentication checks for the client, and prepares one entry of its internal client management table. From the client request, the names of workstations are extracted from which status information should be retrieved. Subsequently, the master daemon forwards the request to all requested daemons which themselves store the location of the master daemon. As a result, the master daemon returns the identifier `waitID` for the request. The client can continue with its work. Later on, the client invokes `nsr_wait()` with `waitID` to finish the non-blocking call. This call checks whether all requested daemons participate to serve the client's request. The return value to the client is the identifier `nsrID`, that has to be passed with every subsequent library call. The master daemon uses `nsrID` to identify the requesting client. In any case, a non-blocking call has to be finished by `nsr_wait()` before a new call can be invoked by a client. A blocking library call is implemented by its respective non-blocking counterpart immediately followed by `nsr_wait()`.

Three additional administrative functions are provided by the NSR library. Firstly, a client can program each daemon by sending a measurement configuration message. A measurement flag determines the set of status information that should be measured by each daemon. In case of periodic measurements, a time value determines the period of time between any two measurements. A new configuration message reconfigures the execution behavior of all attached daemons. Secondly, the set of observed workstations can be dynamically changed during

[6] `http://wwwbode.informatik.tu-muenchen.de/~omis`

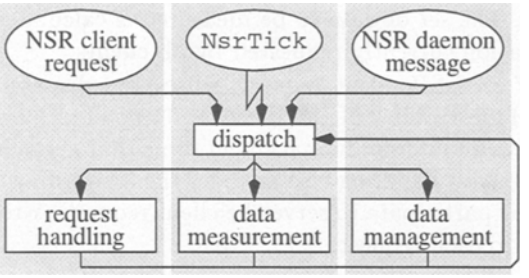

Fig. 3. Functionality of an NSR Daemon

the client's execution time. Finally, a client detaches itself from the NSR server as soon as it does not need service any more.

The second group of NSR library functions provides access to the measured data. A request for immediate measurement blocks the caller until the status information of the requested workstation is available. In case of periodic measurements, the local daemon stores the status information of every node into a local memory segment. A library call reads the memory location to access the information. A client can read the status information of all observed workstations at once. Additionally, it can request the status information of one specific workstation. The NSR master daemon stores status information independently from the execution behavior of requesting clients. Again, it is the client's responsibility to periodically read the data.

4.4 NSR Daemon Functionality

At startup time, an NSR daemon process measures a default set of status data and multicasts the values to all daemons in the cluster. During this phase, it initializes an internal timer that periodically interrupts the execution of the NSR daemon process at fixed time intervals. The period of time between two successive interrupts is called `NsrTick`. Any time interval used by the NSR implementation is a multiplier of this basic `NsrTick`. As shown in Fig. 3, three main tasks can be executed by a single daemon: *client request handling*, *data measurement*, and *data management*.

A central dispatching routine determines the control flow of the NSR daemon processes. It invokes one of the tasks depending on one of the following asynchronous events: 1) the expiration of `NsrTick`, 2) the arrival of an NSR client request, or 3) the arrival of another daemon message.

In the first case, a default set of status information is updated every $n \times$ `NsrTick` with a predefined number n as long as no client request has to be handled. This concept of *alive-messages* ensures that all daemons know about all other daemons at any time. Note, a daemon tries to propagate a *breakdown-message* in case of a failure. If client requests indicate periodic measurements then `NsrTick` is used to check for the expiration of a requested period of time. If necessary, the daemon measures exactly that status information requested at

this point of time. The set of data to be measured is calculated as a superset of the status information currently requested by all clients.

The arrival of an NSR client request causes the dispatcher to invoke the request handling functionality of the daemon. Depending on the request type, the daemon updates an internal management table that keeps information about its local clients. Request handling also includes the forwarding of a client request to all daemons that participate to serve the client request. A result is sent to the client in response to each request.

A daemon can receive different types of messages from other daemons, i.e., alive-messages, breakdown-messages, reply-messages and update-messages. Reply-messages are sent by an NSR slave daemon in response to a forwarded client request. The result of a reply is stored in the client management table. Later on, the results of all slave daemons are checked if the daemon handles the previously mentioned `nsr_wait()` call. Finally, the update-messages indicate that a daemon sends updated status information in case of periodic measurements. New status information of a workstation invalidates its old values. Therefore, the daemon overwrites old workstation entries in the memory segment.

Each daemon in the cluster can perform the same functionality. Typically, several daemons cooperate to serve a client's request. Without going into further details, we state that the NSR server can handle requests from multiple clients. On the one side, applications can be implemented as a parallel program and each process can be a client that observes a subset of workstations. On the other side, several applications from different users can utilize the NSR server concurrently. Additionally, limited error recovery is provided. If necessary, a daemon that recovers from a previous failure is automatically configured to participate in serving the client requests again.

4.5 Basic Communication Mechanisms

Two types of interaction are distinguished to implement the NSR concept: the *client-server interaction* and the *server integrated interaction*, i.e., the interaction between the NSR daemons. Their implementations are based on different communication protocols.

Client-Daemon Communication:: The communication between a client and its local NSR daemon process is performed *directly* or *indirectly*. Direct communication is used for the administrative functions and involves both the NSR client and the NSR master daemon simultaneously. A TCP socket connection is established over which a client sends its requests and receives the server replies. The asynchronous communication uses a two phase protocol based on direct communication (see description above).

Indirect interaction between a client and the local daemon is used to implement data exchange in case of periodic measurements. In analogy to a mailbox data exchange scheme, the daemon writes status information to a memory mapped file independently of the clients execution state. Similarly, any client

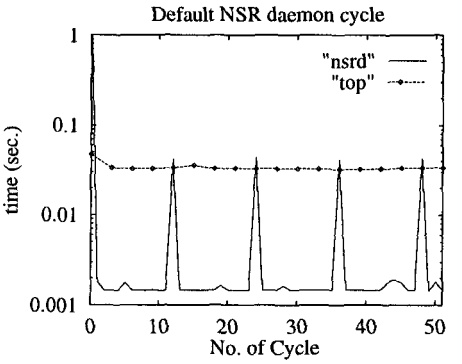

Fig. 4. Measurement duration of `top` and `nsrd`.

located on the workstation can read this data independently of the daemon's status.

Daemon-Daemon Communication: The NSR server implementation is based on the IP multicast concept. The main advantage of IP multicast over a UDP broadcast mechanism is that the dimension of a cluster is not limited to a LAN environment. A multicast environment consists of senders and receivers. Receiving processes are part of a multicast group. In our context, the NSR server is a multicast group consisting of its daemon processes. Communicated data are converted into XDR to cover the heterogeneity of different architectures. In response to a forwarded client request, an NSR slave daemon sends its reply to the master daemon via an UDP socket connection.

5 Evaluation

We will now evaluate the concept and implementation of NSR. As a point of reference, we compare the public domain software product `top` with the execution of `nsrd`, i.e., the NSR daemon process. Two experiments have been performed on a SPARC Station 10 running the SUN Solaris 2.5 operating system. Firstly, we compared the measurement overhead of `top` and `nsrd`. Secondly, we compared the average CPU utilization of both tools to determine their impact on the observed workstation.

A comparison between the measurement overhead of `top` and `nsrd` is shown in Fig. 4. It illustrates the cyclic execution behavior of `nsrd`. Both tools have been instrumented with timing routines to measure the duration of a single measurement.

The NSR daemon process measures its default set of values every $n \times$ `NsrTick`, with `NsrTick` $= 5$ (seconds) and $n = 12$, while `top` updates its values every $m \times$ `NsrTick`, with $m = 3$. The set of measured information covers CPU and memory usage statistics. The set of values is approximately the same for both tools. For $i \bmod 12 \neq 0$, the `NsrTick` just interrupts the execution of `nsrd` that subsequently performs some internal management operations. The time values of Fig. 4 indicate that the measurement routines of `top` and `nsrd` are similar with

respect to their measurement overhead. The reason for this similarity is that the measurement technique is almost identically implemented in both tools.

However, the `nsrd` outperforms `top` in terms of CPU percentage that is a measure for the CPU utilization of a process. The CPU percentage is defined as the ratio of CPU time used recently to CPU time available in the same period. It can be used to determine the influence on the observed workstation. The following lines show a shortened textual output of `top` when both tools execute concurrently.

```
  PID USERNAME ··· SIZE ···  CPU COMMAND
 3317 roeder   ··· 1908K ··· 1.10% top
 3302 root     ··· 2008K ··· 0.59% nsrd
```

The above snapshot was taken when both tools are executing on their regular level. It combines both the time each tool needs to measure its dynamic values and the time to further process these data. The `top` utility prints an list of processes that is ordered by decreasing CPU usage of the processes. The `nsrd` writes the data to the memory mapped file, converts it into the XDR format, and multicasts it to all other daemons. From this global point of view, `nsrd` outperforms `top` by a factor of 2. The overhead of `top` mainly results from its output functionality. In summary, the NSR introduces less load to the observed system than the `top` program.

6 Summary and Future Work

We presented the design and implementation of NSR that is a general-purpose measurement component to observe the status of a cluster of heterogeneous workstations. The NSR provides transparent access to static and dynamic information for any requesting measurement application. A key feature of NSR is its flexibility with respect to measurement requirements of various applications. If necessary, applications can control and observe different subsets of workstations independently. From the viewpoint of NSR, several applications can be served concurrently.

Currently, we implemented a prototype of NSR for SUN Solaris 2.5 and Hewlett Packards HP-UX 10.20 operating systems. In the future we will enhance the implementation of NSR. By that, measurement routines for various architectures like IRIX 6.2, AIX 3.x, and LINUX 2.0.x will be implemented. Additionally, we will optimize the performance of NSR to reduce the measurement overhead. Regarding performance evaluation, we will evaluate the NSR library calls and the data distribution inside the NSR server. In spring 1998, we will release the NSR software package under the GNU general public license. On some architectures, measurements are performed by reading kernel data structures. In this case, read permission to access the kernel data has to be provided for an NSR daemon process. Security aspects are not violated because the NSR does not manipulate any data. Measurement applications can be implemented without special permission flags. In production mode, a NSR daemon process can

automatically start up during the boot sequence of a workstation's operating system.

The NSR already serves as the load measurement component within the LoMan project [11]. In the future, the NSR will be integrated into an OMIS compliant monitoring system to observe the execution nodes of a parallel application.

References

1. A. Beguelin. Xab: A Tool for Monitoring PVM Programs. In *Workshop on Heterogeneous Processing*, pp. 92–97, Los Alamitos, CA, Apr. 1993.
2. J.D. Case, M. Fedor, M.L. Schoffstall, and C. Davin. Simple Network Management Protocol (SNMP). Internet Activities Board - RFC 1157, May 1990.
3. D.G. Feitelson, L. Rudolph, U. Schweigelshohn, K.C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In *IPPS'97 Workshop on Job Scheduling for Parallel Processing*, pp. 1–25, Geneva, Switzerland, Apr. 1997. University of Geneva.
4. K. Geihs and C. Gebauer. Load Monitor $\mathcal{LM}$ – Ein CORBA-basiertes Werkzeug zur Lastbestimmung in heterogenen verteilten Systemen. In *9. ITG/GI Fachtagung MMB'97*, pp. 1–17. VDE-Verlag, Sep. 1997.
5. G.A. Geist, J.A. Kohl, R.J. Manchek, and P.M. Papadopoulos. New Features of PVM 3.4 and Beyond. In *Parallel Virtual Machine — EuroPVM'95: Second European PVM User's Group Meeting, Lyon, France*, pp. 1–10, Paris, Sep. 1995. Hermes.
6. M. Hilbig. Design and Implementation of a Node Status Reporter in Networks of Heterogeneous Workstations. Fortgeschrittenenpraktikum at LRR-TUM, Munich, Dec. 1996.
7. D.J. Jackson and C.W. Humphres. A simple yet effective load balancing extension to the PVM software system. *Parallel Computing*, 22:1647–1660, Jun. 1997.
8. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification. TR. TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, Jul. 1997.
9. B.P. Miller, J.K. Hollingsworth, and M.D. Callaghan. The Paradyn Parallel Performance Tools and PVM. Tr., Department of Computer Sciences, University of Wisconsin, 1994.
10. C. Röder. Classification of Load Models. In T. Schnekenburger and G. Stellner (eds.), *Dynamic Load Distribution for Parallel Applications*, pp. 34–47. Teubner, Germany, 1997.
11. C. Röder and G. Stellner. Design of Load Management for Parallel Applications in Networks of Heterogeneous Workstations. TR. TUM-I9727, SFB 342/18/97 A, SFB 0342, Technische Universität München, 80290 München, Germany, May 1997.
12. C. Scheidler and L. Schäfers. TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications. In *Proc. PARLE'93, Parallel Architectures and Languages Europe*, vol. 694 of *LNCS*, pp. 403–413, Berlin, Jun. 1993. Springer.
13. D.W. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, 20(4):657–673, Apr. 1994.
14. X/Open Group. *Systems Management: Universal Measurement Architecture Guide*, vol. G414. X/Open Company Limited, Reading, UK, Mar. 1995.