TDP_SHELL: An Interoperability Framework for Resource Management Systems and Run-time Monitoring Tools*

Vicente Ivars, Ana Cortes, Miquel A. Senar

Departament d'Arquitectura d'Ordinadors i Sistemes Operatius
Universitat Autònoma de Barcelona
Barcelona, Spain
vicente@aomail.uab.es,{ana.cortes, miquelangel.senar}@uab.es

Abstract. Resource management systems and tool support are two important factors for efficiently developing applications in large clusters. On the one hand, management systems (in the form of batch queue systems) are responsible for all issues related to executing jobs on the existing machines. On the other hand, run-time tools (in the form of debuggers, tracers, performance analyzers, etc.) are used to guarantee the correctness and the efficiency of execution. Executing an application under the control of both a resource management system and a run-time tool is still a challenging problem in most cases. Using run-time tools might be difficult or even impossible in usual environments due to the restrictions imposed by resource managers. We propose TDP-Shell as a framework for providing the necessary mechanisms to enable and simplify using run-time tools under a specific resource management system. We have analyzed the essential interactions between common run-time tools and resource management systems and implemented a pilot TDP-Shell. The paper describes the main components of TDP-Shell and its use with some illustrative examples.

1. Introduction

Large distributed clusters are becoming a common platform for running compute-intensive applications. Developing applications that run in these environments is still a difficult task, even after several decades of intense research on methodologies and supporting tools. The intrinsic complexity of distributed systems and the continuous changes in hardware, operating systems and middleware platforms contribute to this complexity. Run-time tools play an important role in program development and we can find tools of various types to help programmers to develop, optimize and maintain code [1][2]. These tools can be used for debugging, performance analysis, program tracing, program flow visualization and computational steering.

* This work was supported by MEyC-Spain under contract TIN 2004-03388, and partially supported by NATO under contract EST.EAP.CLG 981032.

While these tools are readily available in the programmer's desktop computer, when programs move into a distributed environment their usage and availability become more difficult. On one hand, the required components of the run-time tools may not be in place and, on the other hand, the run-time tool may not be able to carry out some necessary actions to control and monitor the application. These problems generally arise due to conflicts between executing run-time tools and the existence of resource management systems that schedule access to the distributed resources. Resource management software is responsible for accessing resources and providing the resources needed to run a job, as well as monitoring the job's execution and retrieving any results produced by the job. It plays a crucial role in any distributed cluster because it guarantees that applications are executed seamlessly and securely. Common resource management systems are used in local clusters in the form of batch queuing environments (e.g. Condor [3] or PBS[4]).

Using run-time tools in a distributed environment is difficult because of complex interactions between the application program, operating system, and layers of job and resource management software. This leads to situations where executing a run-time tool to monitor an application is incompatible with the existence of a resource management system unless the run-time tool is individually ported and adapted to run under each particular resource management system. We refer to this problem as a problem of interoperability between run-time tools and resource management systems. By interoperability, we mean the ability of different tools and resource managers to co-operate in controlling user applications by using common services and communication mechanisms.

A different approach to the interoperability problem was also considered in the literature when the concept of interoperable tools was used by some authors to refer to using more than one run-time tool in a user's program [5][6][7][8]. For instance, using a debugger and a performance analyzer concurrently is an example of two inter-operable tools. In this case, the problem of interoperability applies to sets of tools that work at the same level, while in our work we consider resource managers and run-time tools, which have a different hierarchy in the system. Run-time tools have to run under the control and supervision of resource managers.

There are few references to this case of the interoperability problem in the literature on development tools for distributed systems. The TDP (Tool Daemon Protocol) [9] constitutes a recent contribution aimed at providing a standard interface that codifies the essential interactions between the run-time tool, the resource manager and the application program in order to make them interoperable. Using TDP-enabled tools and resource managers significantly simplifies the interoperability problem because it reduces the porting effort. Obviously, this requires modifications in existing tools and resource management systems in order to include the necessary TDP pieces. This limitation of TDP has motivated the continuation of this work towards TDP-Shell, a framework intended to provide interoperability in a flexible and easy way, which does not require changes in run-time tools or in resource managers. TDP-Shell provides a scripting-like language that allows programmable steps to be specified to guarantee their cooperation while executing a user application with maximum transparency and portability. It also uses agents interposed between the resource manager and the run-time tool that execute all these programmed steps.

The rest of the paper is organized as follows. Next, in Section 2, we discuss the problem background and we review the main characteristics of TDP, which has been used as a basis for TDP-Shell. Section 3 describes the general architecture of TDP-Shell and Section 4 illustrates usage examples of TDP-Shell. We present conclusions and future work in Section 5.

2. Problem Background

Analysis of run-time tools and resource management environments shows a common architecture based on the existence of two main pieces: a front-end part that runs on the user's desktop computer and a back-end part that runs on the remote host where the user's application is also running. This common configuration requires a communication channel (typically a TCP/IP connection) that is established between the front-end and the back-end processes.

There are several crucial issues that must be addressed when an application running under the control of a resource management system also has to be monitored by a run-time tool.

- *Process creation:* This operation can be in conflict with a tool such as a debugger or profiler that also expects to launch the process. While most sophisticated run-time tools have the ability to attach to a running process, they cannot handle the case when the tool wants to attach to the process before it starts execution. There might be scenarios in which an application process is created but does not start, so the run-time tool attaches to the process and performs its initial processing, and then starts the application. Tools such as gdb, Totalview, and Paradyn use this technique. The appropriate information must be provided to the run-time tool so that it can find and operate on the application program.
- *Tool creation:* Similarly to the application, the resource manager is responsible for launching the run-time tool. This action also implies that configuration and data files needed by the run-time tool are transferred to the execution nodes. The run-time tool might be launched before the application is created (as above) or launched afterwards. In this second case, the resource manager must provide the appropriate information to the run-time tool so that it can attach to and operate on the application.
- *Process monitoring and control:* In the course of normal operation, the resource manager may pause and resume or vacate the application process. All these actions should also affect the run-time tool back-end, while the tool front-end is notified somehow.
- *Front-end/back-end coordination:* The front-end and the back-end of the run-time tool need to communicate and this communication is typically done with TCP/IP sockets. This communication can generally be established by a host/port number pair that must be provided either to the front-end or to the back-end when the other party has started execution. Executing a back-end controlled by the resource manager implies that in most cases this information is not known before hand and the front-end must wait until the resource manager allocates a particular resource and starts the back-end there. Therefore, a synchronization and coordination

mechanism is required to guarantee a proper connection between the front–end and the back-end.

TDP-Shell is specifically targeted at the above mentioned issues, which are related to the problem of interoperability between a resource manager and a run-time tool. Other significant works have studied the problem of interoperability between run-time tools and the issues involved in coordinating the interactions between multiple run-time tools. While TDP-Shell is designed to allow multiple tools to be launched for a given application, the interactions between these tools must be coordinated by the tools themselves.

To the best of our knowledge, no other significant works have been proposed to deal with the interoperability problem as considered in this paper. There are isolated solutions (such as Totalview [10] running under MPICH) but only the Tool Daemon Protocol (TDP) mentioned in the introduction addresses the problem in a general way by proposing an interface that codifies the essential interactions between run-time tools and resource managers.

2.1 The Tool Daemon Protocol (TDP) library

Our work on TDP-Shell uses the basic functionality provided by the Tool Daemon Protocol library. The TDP library provides three main groups of services: process management, inter-daemon communication interface and event notification. We outline below the main features related to these three groups. The core component of TDP consists in an Attribute Space that is used as a medium for data exchanging, for process synchronization and for event notification. The Attribute Space has many similarities with a Linda tuple space [11]. The two basic Attribute Space primitives are *tdp_get* and *tdp_put*. Information in the shared environment space is kept in the form of (attribute, value) pairs, where both the attribute and value are constrained to only being null terminated strings. An attribute consists simply in a character string that names data in the shared space. In the current implementation, the attribute space is supported by a Central Attribute Space Service that runs on the front-end machine. Any process using the TDP library can access the attribute space of the CASS.

The basic functions to work with the attribute space are the following:

- *tdp_get*: obtains the value of a given attribute from the attribute space
- *tdp_put*: inserts a new (attribute, value) pair into the attribute space.
- *tdp_del*: obtains and removes a given attribute from the space.

These operations block forms of communication between a daemon and the C*ASS*. Asynchronous versions for retrieving and storing information from the shared space are also available: (*tdp_async_get, tdp_async_put and tdp_aync_del*).

Finally, the TDP library provides several process management functions that are used to create, destroy, attach to, detach from, suspend or resume processes. While similar process management functions are present in common operating systems, TDP provides its own interfaces that are OS neutral.

3. The TDP-Shell Architecture

TDP-Shell is a framework based on two process agents. One agent runs on the front-end machine (referred to as TDP-SC:TDP-Shell Console) and the other agent runs on the back-end machine (referred to as TDP-SA: TDP-Shell Agent). Communication between the TDP-SC and the TDP-SA is based on the TDP attribute space. The framework also includes a scripting-like language that is used to describe actions that must be carried out both at the front-end machine and the back-end machine in order to start the run-time tool components and the process application in the right order. Both TDP-SC and TDP-SA execute a command file written in the above mentioned scripting language. Most of the available commands consist of wrappers to the TDP library functions mentioned above. Additionally, the command file may include simple assignment and control flow statements. Figure 1 shows the main components of the framework and their connection to the resource management and the run-time tool daemons.

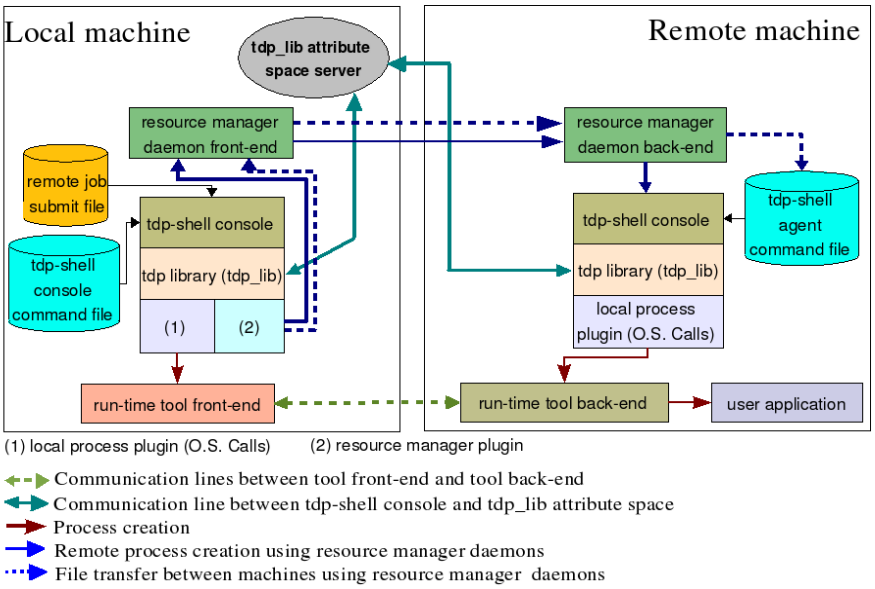

Figure 1. Architecture of the TDP-Shell framework

1. TDP-Shell Console (TDP-SC): this is the process that runs at the front-end machine. It receives a job submit command file that specifies all the actions that should be carried out in the front-end machine. Among others, the job submit command file contains a set of commands to submit the application process to the resource manager, commands to start the run-time tool front-end and commands for the subsequent steps required to synchronize the run-time tool front-end with the run-time tool back-end. TDP-SC also contains the specific logic that guarantees that all the necessary components of the run-time tool are transferred to the remote machine. It does this by generating an appropriate submit file that is

later submitted to the resource manager. A specific plug-in for different resource managers is used in TDP-SC to generate these submission files. TPD-SC is also responsible for starting the central attribute space and it may run as an interactive process or in the background.

2. TDP-Shell Agent (TDP-SA): this is the process that runs in the back-end machine. It is started by the back-end daemon of the resource manager and, similarly to the TDP-SC, it executes a command file that specifies the actions that must be carried out to start the application process and the run-time tool back-end daemon. The command file also contains communication and synchronization actions between TDP-SA and TDP-SC through the attribute space provided by the TDP library. TDP-SA sits between the resource manager daemon and the application process and therefore it is also responsible for forwarding to the latter all control actions generated by the former.
3. TDP-Shell command file: this file contains the set of commands that specify the actions that should be carried out either at the front-end machine or at the back-end machine. Commands are wrappers to the functions provided by the TDP library that are combined with flow control statements and local function definitions.

4. Operation of the TDP-Shell

Below, we briefly sketch the process of submitting a user job using the TDP-Shell.

1. In the initial state, the front-end and the back-end resource management daemons are running in the corresponding machines (see
2. Figure 2). Users supply their applications with all the necessary files, a job submission file and two TDP-Shell command files (a local one and a remote one). The job submission file is specific for each resource management system (for instance, this file could be a shell script in PBS and a description file in Condor).

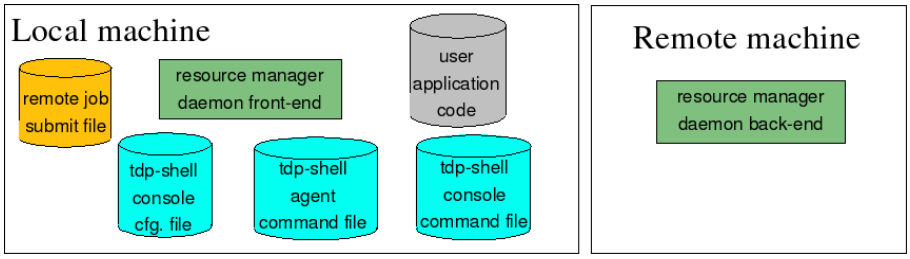

Figure 2. TDP-Shell operation: initial state

3. Users create the TDP-Shell Console (using a *tdp-sc* command). The corresponding TDP-Shell command file is provided as a complement to TDP-SC (Figure 3 shows an example of the command file that uses *gdb* as a run-time tool). The TDP-SC starts the execution of the TDP-Shell command file. Following the example in Figure 3, the attribute space is created once the **tdp_init** command is called (see Figure 4).

tdp_gdb_commands.tdp

```
$RESOURCE_MANAGER= Condor
$JOB_SUBMIT_FILE=job.cfg
$REMOTE_COMMANDS_FILE =tdp_gdbserver_commands.tdp

tdp-include file_to_gdb.sh

tdp-fun fun_error ($error_msg) {
   tdp-print $error_ms
   tdp-exit
}
tdp-fun fun_gdbserver_end () {
      (b1)
   tdp-print _$async_value
}
tdp-init
tdp-asyncget END_GDBSERVER fun_gdbserver_end
$prog_debug_name=/tdp-tool/bin/demo_prog
tdp-put PROG_DEBUG=$prog_debug_name
$ret_fun=tdp-launch
if ($ret_fun==ERROR) {
   $error_info="ERROR launching the remote gdbserver"
   fun_error $error_info
   tdp-exit
}
$port=tdp-get PORT
$host_remote=tdp-get REMOTE_HOST
file_to_gdb.sh $host_remote $port $gdb_remote
$gdb_id=tdp-create_process -interactive gdb -x $gdb_remote
                                $prog_debug_name
if ($gdb_id==ERROR) {
    $error_info="ERROR creating the gdb process"
    tdp-asyncput END_GDB=$error_info

    fun_error $error_info

}
repeat ($ret_fun!=FINISH) {
    $ret_fun=tdp-process-status $gdb_id
    if ($ret_fun==ERROR){
        $error_info="ERROR during execution of gdb"
        tdp-asyncput END_GDB=$error_info
        fun_error $error_info
    }
}
tdp-asyncput END_GDB=”gdb has finished”
tdp-exit
```

tdp_gdbserver_commands.tdp

```
tdp-include hostname.sh


tdp-fun fun_error ($error_msg) {
   tdp-asyncput ERROR_GDBSERVER=$error_msg
   tdp-exit
}
tdp-fun  fun_gdb_end () {
    tdp-exit
}

tdp-init
tdp-asyncget END_GDB fun_gdb_end

hostname.sh  &$local_host
$prog_debug=tdp-get PROG_DEBUG

$process_id=tdp-create_process -paussed $prog_dubug
if ($process_id==ERROR) {
   error_info="ERROR paussing the  program to debug "
   fun_error $error_info
}

$gdbserver_id=tdp-create_process gdbserver $local_host:5000
                                    -atach 'pidof $prog_debug'
if ($gdbserver_id==ERROR) {
   $error_info="ERROR in the gdbserver command"
   fun_error $error_info
}

tdp-put PORT=5000
tdp-put REMOTE_HOST=$local_host

repeat ($ret_fun!=FINISH) {
$ret_fun=tdp-process-status $gdbserver_id
   if ($ret_fun==ERROR){
       $error_info="ERROR during execution of gdbserver  "
       fun_error $error_info
   }
}

tdp-asyncput END_GDBSERVER=”gdbserver has finished”
tdp-exit
```

Figure 3. TDP-Shell command file example. Left: TDP-Shell Console part; Rigth: TDP-Shell Agent part

4. A job is submitted to the resource manager when the *tdp_launch* command is found. The TDP-Shell Console creates a special submission file taking into account the particular resource manager used in the system (in our example, $RESOURCE_MANAGER = Condor), the original job submission file ($JOB_SUBMIT_FILE = job.cfg) and if necessary the binaries and other additional files required by the back-end daemon of the run-time tool (in our example, we assume that these binaries are available in every remote machine, so they don’t need to be transferred). A specific plug-in for each resource management system (see Figure 1) is responsible for generating the special submission file. Actually, TDP-SC submits a job that consists in the TDP-Shell Agent. The information about the original user’s job and the run-time tool are combined in this TDP-SA job automatically and transparently. As a consequence, when the resource manager finds a suitable machine for the job, it will actually

start an instance of the TDP-Shell Agent. The TDP-SA command file (see example of TDP-SA command file on the right side of Figure 3), the user job files and the run-time tool files will also be copied in the same machine. The TDP-SA starts the execution of the command file and joins the attribute space when the **tdp_init** command is called (see Figure 5).

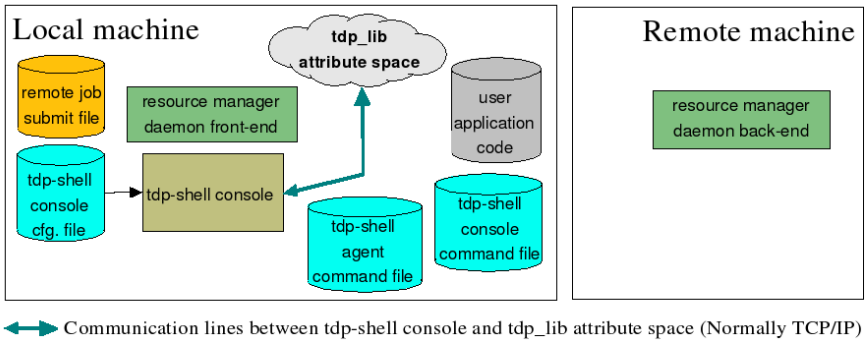

Figure 4. TDP-Shell operation: job submission

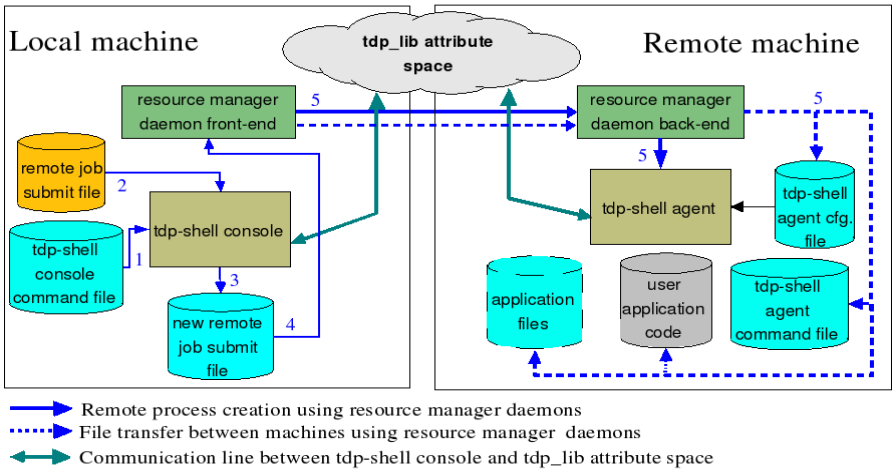

Figure 5. TDP-Shell operation: TDP-SA start up

5. The TDP-SC and the TDP-SA execute their commands concurrently and they synchronize or exchange information using the attribute space provided by the TDP library. Specifically, the TDP-SC will start the run-time tool front-end daemon and the TDP-SA will create the user job and the run-time tool-back end daemon (see Figure 6).
   The order for creating these elements depends strongly on the run-time tool's characteristics. Some tools require the front-end daemon to start before the back-end daemon (i.e. Paradyn [12]), while others require the opposite order (i.e.

Totalview [10] or gdb [13]). In order to accommodate the requirements of any particular run-time tool, users can control the order of creation by using the appropriate synchronization between the TDP-SC and the TDP-SA. In our example in Figure 3, the TDP-SA first creates the gdbserver (back-end daemon) and puts two attributes in the attribute space (PORT and REMOTE_HOST), which are needed in the TDP-SC to start the gdb front-end. The TDP-SC is blocked until these two attributes are put in the attribute space and then it starts the gdb front-end. In general, most run-time tools publish some information that is required to establish the connection between its back-end and front-end daemons properly. Unfortunately, there is no common and easy way to obtain this information. For instance, the Paradyn front-end publishes its two connection ports in an external file, the gdb front-end must know the host name of the machine where the gdbserver has been starter, and so on. An external user function provided by the user can be used to obtain this information. This external function (shell script) must publish the information in the *stdout*; it is declared with the *tdp_include* statement and invoked within the TDP-Shell command file. Finally, the application can be created either in a paused or non-paused way. The choice depends on the ability of the tool to attach to the application later and continue it.

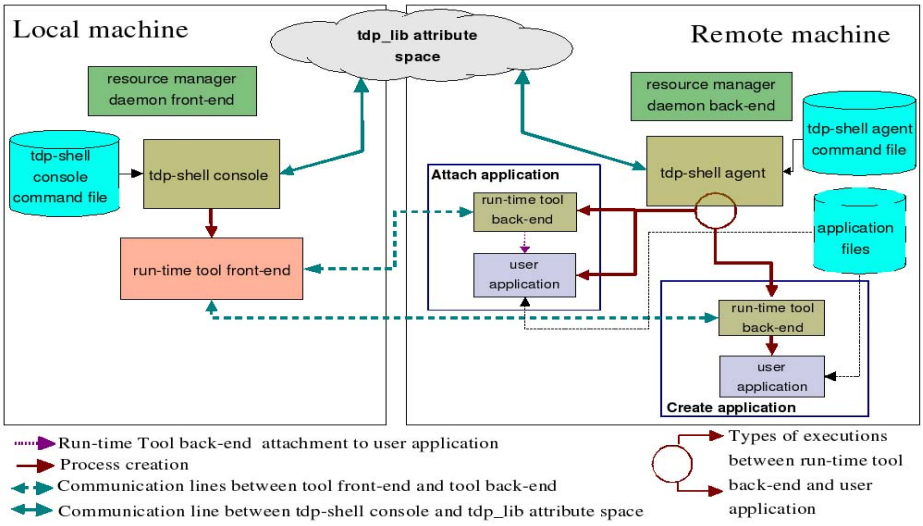

Figure 6. TDP-Shell operation: run-time tool and application start-up

Once the application and the run-time tools have been successfully created, they continue their execution in a normal way. Users may interact with the run-time tool front-end and control the execution of the application as usual. TDP-SA detects the finalization of the application or the run-time back-end daemon. It will carry out any programmed action included in the command file, which includes notifications through the attribute space (the example in Figure 3 contains some examples of error control statements that are invoked asynchronously).

6. Conclusions and Future Work

Large-scale distributed environments imply a new scenario that requires new mechanisms that enable run-time monitoring tools to be launched under the control of a resource management system. We have developed TDP-Shell as a generic framework that is able to deal with a wide range of different run-time tools and resource managers. TDP-Shell uses a simple and easy notation mechanism, similar to the one exhibited by most OS shells, to specify the interactions between the run-time tool and the user application when executed by a given resource management system. TDP-Shell is based on two agents that have little impact on the normal execution of the application and introduce minimum overhead (mostly at the application launching time). Our current prototype includes two resource manager plug-ins (one for PBS and one for Condor) and it has been successfully used to submit sequential applications to these two batch systems and monitor them using gdb and Paradyn. Currently, a user is still limited to specifying the usage of a run-time tool at the submission time and it is not possible to start a run-time tool on-the-fly if it was not specified when the application was submitted. Our future aims are to overcome this limitation and also to support parallel applications (based on MPI).

6. References

[1] T. Sterling. P. Messina and J. Pool. "Findings of the second Pasadena Workshop on system software and tools for high performance computing environments", Tech. Report 95-162, Center of Exc. in Space Data and Inform. Sciences, NASA, 1995E.

[2] S. Johnsen, O. J. Anshus, J. M. Bjørndalen, and L. A. Bongo, "Survey of execution monitoring tools for computer clusters", Tech. Report, Univ. of Tromso, Sept. 2003.

[3] M.J. Mutka, M. Livny and M. W. Litzkow, "Condor – A Hunter of Idle Workstations", 8th Int'l Conf. on Distributed Systems, San Francisco, June, 1988.

[4] http://www.openpbs.org/

[5] R. Wismuller, J. Trinitis and T. Ludwig, "OCM-A Monitoring System for Interiperable Tools", in Proc. 2nd SIGMETRICS Symposium on Parallel and Distrubuted Tools, Welches, USA, August 1998.

[6] T. Ludwig and R. Wismüller, "OMIS 2.0 -- A Universal Interface for Monitoring Systems", in Proc. 4th European PVM/MPI Users' Group Meeting, pp. 267-276, 1997.

[7] G. Rackl, M. Lindermeier, M. Rudorfer and B. Süss, "MIMO-An Infraestructure for Monitoring and Managing Distributed Middleware Environments", in Proc. Middleware 2000, pp. 71-87, 2000.

[8] R. Prodan and J. M. Kewley, "A Framework for an Interoperable Tool Environment", Proc. of EuroPar 2000, Lecture Notes in Computer Science, vol. 1900, pp. 65-69, 2000.

[9] B. Miller, A. Cortes, M. A. Senar, and M. Livny, "The Tool Daemon Protocol (TDP)", Proc. SuperComputing, November, 2003.

[10] Etnus LLC, "TotalView User's Guide", Document version 6.0.0-1, January 2003. http://www.etnus.com

[11] N. Carriero and D. Gelernter, "Linda in Context", Comm. of the ACM, 32, 4, pp. 444-458, 1989.

[12] B.P. Miller, et alter, "The Paradyn Parallel Performance Measurement Tools", IEEE Computer 28, 11, 1995.

[13] http://www.gnu.org/software/gdb/