The Tool Dæmon Protocol (TDP)

Barton Miller, Ana Cortés†, Miquel A. Senar†, and Miron Livny**

*Computer Sciences Department
University of Wisconsin
Madison, WI 53706 USA
{bart,miron}@cs.wisc.edu

†Departament d'Informàtica
Universitat Autònoma de Barcelona
08193 Bellaterra (Barcelona) Spain
{miquelangel.senar,ana.cortes}@uab.es

Abstract

Run-time tools are crucial to program development. In our desktop computer environments, we take for granted the availability of tools for operations such as debugging, profiling, tracing, checkpointing, and visualization. When programs move into distributed or Grid environments, it is difficult to find such tools. This difficulty is caused by the complex interactions necessary between application program, operating system and layers of job scheduling and process management software. As a result, each run-time tool must be individually ported to run under a particular job management system; for m tools and n environments, the problem becomes an m × n effort, rather than the hoped-for m + n effort. Variations in underlying operating systems can make this problem even worse. The consequence of this situation is a paucity of tools in distributed and Grid computing environments.

In response to the problem, we have analyzed a variety of job scheduling environments and run-time tools to better understand their interactions. From this analysis, we isolated what we believe are the essential interactions between the run-time tool, job scheduler and resource manager, and application program. We are proposing a standard interface, called the Tool Dæmon Protocol (TDP) that codifies these interactions and provides the necessary communication functions. We have implemented a pilot TDP library and experimented with Parador, a prototype using the Paradyn Parallel Performance tools profiling jobs running under the Condor batch-scheduling environment.

1 INTRODUCTION

Run-time tools are crucial to program development. In our desktop computer environments, we take for granted the availability of tools for operations such as debugging, profiling, tracing, checkpointing, and visualization. When programs move into distributed or Grid environments, it is difficult to find such tools.

Distributed computing is well-established, gaining popularity in recent years thanks to the availability of compute clusters, shared-memory multiprocessors, and Grid infrastructure. In general, such systems require software to schedule access to them, stage the resources needed to run a job, monitor the jobs's execution and retrieve any results produced by the job. This software is commonly referred as a resource manager and it has been used in local clusters in the form of batch queuing environments.

Resource management plays a crucial role in the cluster environment because it has the responsibility of carrying out the necessary steps to guarantee the execution of applications in a seamless and secure way by supporting mechanisms such as resource discovery, status monitoring, selection, allocation and job control. Systems such as Condor [13], Load Leveler [9], NQE [4], and LSF [15] are some of the most popular commercial and research batch queuing environments currently used to schedule jobs in a local area cluster.

More recently, attention has focused on Grid computing, using systems such as Globus [7] or Legion [8] to provide access to heterogeneous collections of widely distributed, dynamically configured resources. The presence of such a Grid system provides additional services for authentication, data staging, monitoring, and scheduling. While these interfaces are crucial for running programs in this complex environment, they offer additional layers of interfaces and abstractions that must be negotiated when trying to deploy a run-time tool in that environment.

The use of run-time tools in a distributed environment is difficult because of the complex interactions between application program, operating system, and layers of job scheduling and process management software. As a result, each run-time tool must be individually ported to run under a particular job management system; for m tools and n environments, the problem becomes an $m \times n$ effort, rather than the hoped-for $m + n$ effort. While there have been isolated point-solution successes (such as Totalview [5] running under MPICH), the effort needed to generally solve the problem is prohibitive. Variations in underlying operating systems can make this problem even worse. The consequence of this situation is a paucity of tools in distributed and Grid computing environments.

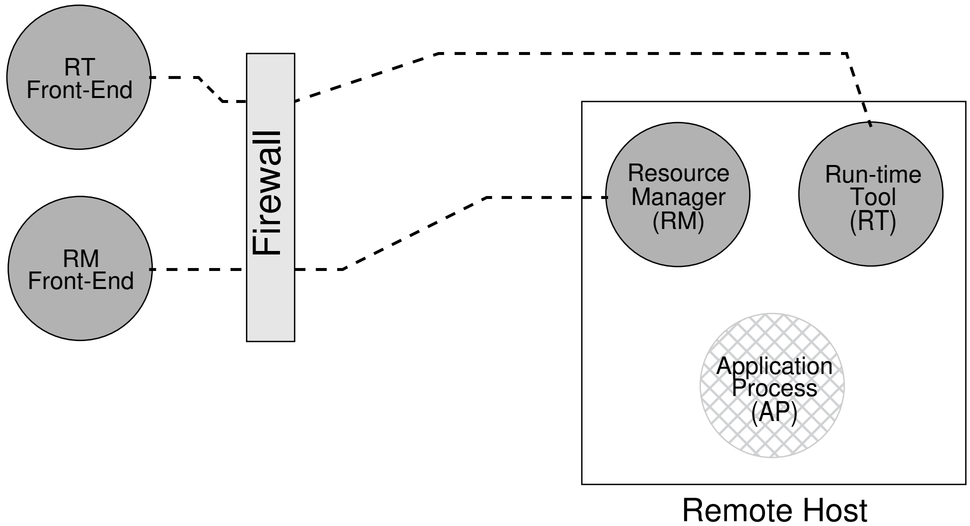


Figure 1: Remote Execution with Resource Manager and Run-Time Tool

To address this problem, we have analyzed a variety of job scheduling environments and run-time tools to better understand their interactions. From this analysis, we have isolated what we believe are the essential interactions between the run-time tool (RT), job scheduler and resource manager (RM), and application program (AP). Figure 1 illustrates the main components. We are proposing a standard interface, called the *Tool Dæmon Protocol* (TDP) that codifies these interactions and provides the necessary communication functions to the RT, RM, and AP.

There are several crucial interfaces needed by run-time tools to monitor and control programs. Our goal was to identify these interfaces, as used by both the RM and RT.

- *Process creation:* A fundamental operation of a RM is launching new processes. 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, this does not handle the case where the tool wants to attach to the process before it starts execution. There needs to be clean division of work between the RM and RT: (1) RM creates, but does not start, the application process, (2) the RT attaches to the process and performs its initial processing, and (3) the RM then starts the application. The RM must provide the appropriate information to the RT so that it can find and operate on the application program.
- *Tool creation:* The RM is responsible for launching the RT. The RT might be launched before the application is created (as above) or launched afterwards. In this second case, the RM must provide the appropriate information to the RT so that it can attach to and operate on the application.
- *Process control:* In the course of normal operation, a RT may pause and resume the application process. This change in process state might be viewed by the RM as a sign of faulty behavior, so the RT must coordinate these operations with the RM.
- *Status monitoring:* As just stated, the RM is interested in state changes of the application process, but the RT also needs to receive this information. There must be agreement as to which entity is responsible for this information and under what circumstances (and different operating systems, even different versions of Unix, have distinctly different behavior in this area). There also must be a mechanism by which this information is communicated to other entity.
- *Standard input and output management:* When a RT tool launches an application program, it often intercepts the applications standard input and output so that this information appears at the same location as the RT's front-end. This control of standard input and output could be in conflict with similar operations done by the RM. This operation properly belongs to the RM, but must be coordinated with the RT.
- *Tool daemon to tool front-end communication:* The common front-end/back-end configuration for a RT requires that a communication channel (typically a TCP/IP connection) be established between the front-end and back-end processes. In the case of private networks (using firewalls or NAT), communication from the to execution nodes may require a proxy (that is likely already provided by the RM for its own purposes). The RM must be able to estab-

lish a proxy tunnel for the RT's communication needs.

- *Tool daemon configuration and data files:* The RT may need configuration files transferred to the execution nodes. The RT might also generate output files that contain traces or summary data; if these trace files will be processed off-line, they must be transferred from the execution nodes after the application completes.
- *Auxiliary services (AS):* There are entities in addition to the RM and RT that may be required for the proper execution of a RT in a distributed environment. For example, software multicast/reduction networks are crucial to scalable tool use [1,6,11,16]. The RM must be aware of and willing to launch this second kind of non-application entity.
- *Fault detection and recovery:* Any of the three entities launched by the RM (AP, RT, AS) can fail during execution. The RM must be able to detect these failures, respond to them, and perhaps communicate their occurrence to the other entities. A clear and precise fault model is required. Note that modeling and detecting faults is ongoing work and beyond the scope of this paper.

There are some things that are explicitly not part of TDP. In general, TDP does not try to solve a problem in the distributed environment that is not already solved (in general usage) in the desktop environment, for example coordinating the interactions between multiple run-time tools. While TDP is designed to allow multiple tools to be launched for a given application, the interactions between those tools must be coordinated by the tools themselves. While some excellent experimental work has been done in this area [10], this feature is not found in tools that are in common use.

To address the problems presented above, both resource managers and monitoring tools to be aware of the existence of each other and to be prepared to execute under such conditions. Unfortunately, neither the currently available resource managers and monitoring tools nor those under development in on-going Grid projects support the above-mentioned functionality. This work deals with the analysis and design of basic services to overcome the drawbacks and limitations existing in middle ware services, as described above. TDP is an effort to address the problem of tool interoperability in distributed and Grid environments. By interoperability, we refer to the ability of different tools and resource managers to co-operate in controlling user applications by using common services and communication mechanisms.

To allow for the general deployment of run-time tools in distributed environments with maximum transparency and portability requires the extension and enhancement of both monitoring tools and resource management systems to make them interoperable. A consequence of this project is that run-time tools should be more easily deployed onto distributed infrastructures, easing the task of application program development. In other words, it will make distributed environments significantly easier to use for application developers and will allow tool builders to concentrate on key technologies rather than on repetitive porting efforts.

2 TDP INTERFACES

The Tool Daemon Protocol provides interfaces for creation of application processes, the subsequent monitoring and control of these processes, establishing connections between the various tool daemon components, and a means of exchanging configuration data. We start with a discussion of this data exchange facility, called the *attribute space*, as several of the other operations depend on it.

2.1 The Attribute Space

There are several cases in which the RM, RT, and AP must exchange information. Examples of such cases include the RM telling the RT the process ID of the AP, the RM providing the RT with the network address (host/port number) of its front-end, and the RT (or the RM) providing the AP with the network address of its standard input and output. Instead of designing special protocol messages for each type of information to be exchanged, we have organized our communication about a general purpose attribute-value space. The mechanism was inspired by the new MPI Process Daemon (MPD) from Argonne [2], and can be considered a highly simplified version of the Linda tuple space [3]. Note that the X-window server [14] also has similar mechanism for use by its clients.

This interface should exhibit desirable characteristics such as generality, portability and extensibility. The kind and format of information that may be exchanged does not have to be restricted to any particular combination of resource manager or monitoring tool. Rather, it should be based on flexible and extensible mechanisms that enable any pair of resource managers and monitoring tools to communicate effectively.

Each host on which an application process (and tool daemon) runs have a local instance of the attribute space server (LASS). There is also a central attribute space server (CASS) process on the host running the tool front-end. A process using the TDP library can access the attribute space of its LASS or the CASS, but cannot access the LASS's of other nodes. The LASS's are started by the RM, while the CASS is started by the RM front-end process. Figure 2 shows the same structure as in Figure 1, with the addition of the attribute servers.

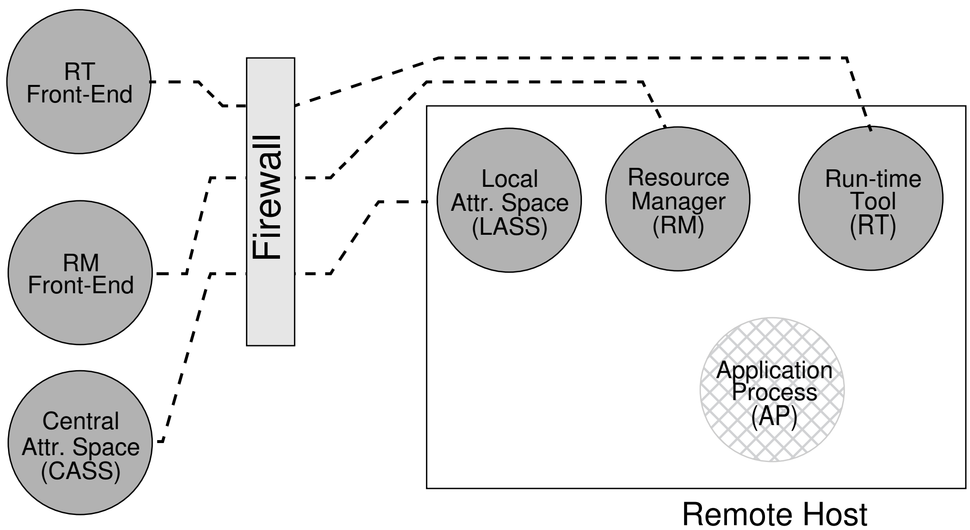


Figure 2: Remote Execution with Local (LASS) and Global (GASS) Attribute Space Servers Added

Attributes and values can be inserted and removed from the attributed space with a simple put/get interface. There is also a mechanism for providing asynchronous notifications. The details of this interface are provided in Section 3.2.

2.2 Application Process Creation

Run-time tools use a variety of schemes for creating application processes. These schemes include:

1. Create the application process and start it running: This scheme is typically the simplest, used in tools that perform no external initialization of the application program. Any needed initialization is done by code already compiled or linked into the application. This scheme requires the least mechanism to support. Tools such as Vampir and PCL use this technique.
2. Create the application, initialize it, and then start it running. This scheme allows the tool to perform initialization in the time before creating the application, but before it is started execution. In Unix terminology, both the fork and exec must complete, but execution then must be stopped. There is a question of how much initialization code in the application should be run, but the intuition is that program should be stopped before starting to execute the main function. Tools such as gdb, Totalview [5], and Paradyn [12] use this technique.
3. Attach to a running application process: Attaching is an important mechanism for operating on already-running programs (such as servers) or programs with complex start-up sequences. Attaching requires the following steps: (1) obtain control of the application, usually with /proc or ptrace; (2) pause the application; (3) perform some tool initialization, such as reading the symbol table or parsing the executable; and (4) continuing the application. Again, tools such as gdb, Totalview, and Paradyn use this technique.

In a resource manager environment, case 1 requires no special modifications to the tools, while cases 2 and 3 require change. In cases 2 and 3, the RT is no longer creating the application process; that responsibility now belongs to the RM. As a result, when using the TDP library, steps 2 and 3 look quite similar.

The sequence of steps for case 2 that must be coordinated by the TDP library is: (1) RM creates and starts the RT; (2) RM creates but does *not* start the application process; (3) RM sends information to the RT that identifies the application process, (4) RT attaches to the application as in case 3, (4) RT performs its initialization, and (5) RT tells the RM to start the application. The communication between the RM and RT use the Attribute Space operations provided by TDP. Note that if the RT has already been started (perhaps because it is already operating on another application process), step 1 might be skipped.

Case 3 under TDP works in a similar way, but with two minor differences. The first difference is that the RT might be created after the application process. Second, step (4) above requires attaching and pausing the application. Not all tools have the ability to use this attach technique. For example, the Vampir trace tool requires the tracing to be started before the application starts execution.

The goal of TDP is to standardize these interactions and encapsulate them in a library so that the details are hidden from the RM and RT. With a small amount of modification (as demonstrated by our experience with

Condor and Paradyn, reported in Section 4), both RM and RT can be reorganized to use the TDP interfaces. The result is RT will be able to operate directly in each RM environment that supports TDP and the RM can run any tool that supports TDP.

2.3 Application Process Monitoring and Control

Under TDP, the responsibility for controlling an application process and for monitoring its status belongs to the RM; i.e., process management operations are localized and encapsulated in the RM. This encapsulation is both good design and a practical necessity. For control operations, the single point responsibility eliminates confusing race conditions. Two different processes will never attempt conflicting control operations. For process monitoring, we also avoid the confusing and often conflicting semantics of various operating systems. For example, under Linux, the parent (RM) process may or may not be the recipient of the child process' termination code. The choice of process can depend on whether some third process (the RT) is attached to control the child (application) process. In one unusual case, the return code might go to both processes.

As for other TDP communication, this status and control information is exchanged using Attribute Space operations. When the RT needs to perform a process management operation, it contacts the RM. When the RM needs to notify the RT about a change in process status, it places a value in the Attribute Space; the RM optionally can use the asynchronous notification to hear immediately about the change.

2.4 Tool Communication

The RT and its front-end need to communicate and this communication is typically done with TCP/IP sockets. If the application is running on a private network with a firewall or gateway, they may not be able to establish a connection through out of the private network. A similar problem can occur if when the standard input/out of the application program needs to be connected to desktop machine of the user (or some other site outside the private network).

Process managers, such as Condor and Globus, provide proxy mechanisms to forwarding their connections in and out of a private network. TDP provides a standard interface to these mechanisms. In general, TDP will provide a host/port number pair to the RT to contact its front-end and to the application program to connect its standard input/out. If there are no routing or addressing restrictions, then the host/port number will be the actual address of the remote process. If the private networks block such connections, then the host/port number will be that of the RM's proxy, which will be responsible for establishing the connection and forwarding inbound and outbound messages.

TDP does not require a new proxy facilities with new permissions; it merely leverages existing ones (if present), and provides a standard interface to such a facility.

3 TDP SERVICES

TDP provides three main groups of services: process management, inter-daemon communication interface and event notification. We outline the main needs and features (and open issues) related to these three groups.

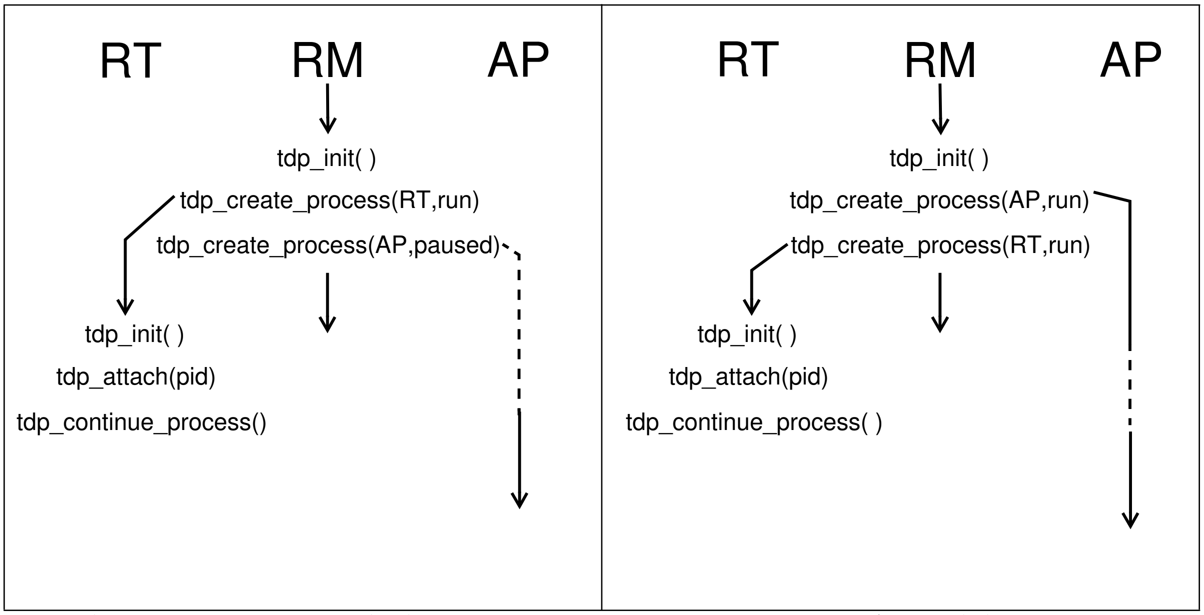
Developers of resource management systems and monitoring tools have been using Unix or Windows interfaces to create and control the execution of applications (fork/exec, /proc interface, ptrace are examples in Unix systems, and CreateProcess and WaitForSingleObject are examples in Windows). TDP provides its own set of interfaces that are OS neutral. The guidelines that were used in designing the API were the following:

- The API should be simple and the API set small.
- The API should be consistent with standard C library interfaces.
- A first implementation will be provided in C language.
- The library should be thread safe (it is expected that the developers will be linking the library from serial and multi-threaded codes).

3.1 Process Management.

TDP supports two scenarios for a RT to operation on an application process, create and attach. Figure 3 illustrates the steps that must be followed by each daemon to use TDP services under each scenario. For the create case (Figure 3A), once the RM is notified that the new application to be launched is going to be monitored, unlike a normal process creation, it will use a `tdp_create_process` function with a paused option, which will launch the process by stopping it at the very beginning (using Unix terms, the process will be stopped just after the execution of the `exec` call). At that point, the TDP framework is initialized by calling `tdp_init`. Finally, RM will launch the RT, which will be started as a regular process using `tdp_create_process`.

In the second scenario, the application is already running and controlled by the resource manager system. At a later time, a RT tool would like to attach to the application process to monitor/analyze it. In this situation, the RM might be notified that it must launch a RT to monitor the running application process. If a RT had been previously created, this step would be skipped. The



A: The RM *creates* the application process **B: The RM *attaches* to existing process**

Figure 3: Steps to Allow a RT to operation on an Application Program

Note that for the create case, the creation of the application process and RT can occur in either order

basic steps followed under this scenario are depicted in Figure 3B. When the user decides to use the RT on the application process, a TDP framework will be established by calling tdp_init and, subsequently, the corresponding RT will be launched. Once these initialization steps have been accomplished, the TDP communication channel will be established and both daemons will be ready to exchange information.

Notice that the state of the application process will be completely different in each of the two cases, but the behavior of the monitoring tool is similar. When the application has been created by the RT, the fork and exec have been successful, but the application process remains stopped at that point. The monitoring tool then will attach and have a chance to perform initialization and track the application's execution from its start. When the RT has attaches to an already running application, the application process will be stopped at some unknown point it its execution.

In both the attach and create cases, once the RT has completed its initialization of the application, it can restart the application using the tdp_continue_process operation.

3.2 TDP Inter-Daemon Communication

Each daemon process in the TDP environment (the RM and each RT) must perform basic initialization, including establishing connection with the Local Attribute Space Service (LASS) and exchanging some basic configuration information. The first step is that each daemon process must execute tdp_init. This function establishes the TDP communication channel between both the RM and RTs. On success, tdp_init will return a *tdp handle*, which will be used in any TDP subsequent action.

Once the TDP framework has been successfully set up, both the RM and RT communicate through the LASS to exchange configuration information. 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 only to be null-terminated strings. An attribute consists simply in a character string that names data in the shared space. While there is a standard list of attribute names for the set of data commonly exchanged between the different daemons (everyone RT and RM must understand this set), different tools and resource managers can extend this set with their own situation specific attributes.

Limiting attribute values to strings, while simple, brings up the problem that there may be the need to provide attributes that multiple or structured values. If we consider, for example, the arguments passed to an application, we would like to pass information that may be something like "-p1500 -P2000". This kind of attributes could be stored into the shared environment space using the simple put operation, and let use TDP client handle the parsing.

The local Attributed Space is initialized when tdp_init is called. A RM that deals simultaneously with several RT may initialize a different space for each RT. Each RT interacts with the RM through its own local Attribute Space, called a *context.* A different *context* parameter is used by the RM in each tdp_init call to create a different space. Communication with a specific RT is accomplished by using its particular context. Multiple tools can share the same space with the RM by using the same *context*, but current tool designs do not yet have a need for such a feature. An RT disengages from the TDP library and Attribute Space with tdp_exit function. An Attribute Spaced shared between a resource manager and several tools (using a common context) will be destroyed when the last element using the specific context calls tdp_exit.

We briefly describe the use of the main functions required to communicate information through the Attribute Space. The basic functions are *tdp_get* and *tdp_put* functions that have a structure whose synopsis can be sketched as follows:

```
tdp_get (handle, attribute, &value)
tdp_put (handle, attribute, value)
```

where *handle* corresponds to the identification returned by tdp_init, *attribute* is the string that identifies the information to store to or retrieve from the Attribute Space, and *value* is the information contained in that attribute.

These operations are blocking forms of communication between a daemon and the *LASS*. In the put case, the function will block until the new attribute is stored in the shared space. In the get case, the function blocks until the value of the corresponding attribute is returned or and error is returned if the attribute is not contained in the shared space.

Asynchronous versions for retrieving and storing information from the shared space are also available:

```
tdp_async_get (handle, attribute, &value, callback,
    callback_arg)
tdp_async_put (handle, attribute, value, callback,
    callback_arg)
```

Both functions will return immediately after being called, however, the storing or retrieval of information may not have been completed at that time. The callback function provided to these functions will be executed when the corresponding operation completes. A user-supplied argument (*callback_arg*) is also passed to the callback function. The use of these asynchronous functions will prevent a daemon process from being blocked in a communication operation with the shared environment space and keep with his own activities.

3.3 Event Notification

In principle, the callback function from an asynchronous get or put would be called once the operation has been completed. However, a pure asynchronous notification mechanisms may be hard to manage in some tools because the obvious UNIX implementation of such features in TDP on UNIX might use signals or threads. Signals are a problem for many run-time tools because the TDP use of these signals might conflict with the RT's use of the signal. Finding a signal that does not conflict with some tool is problematic. The use of threads also is a problem because of the plethora of thread packages. There is no way to select a thread package for TDP that would be compatible with the many possible packages used by RTs.

In many cases, a pure asynchronous notification mechanism is not necessary. Most RTs and RMs have a central polling loop where they use an operation such as the Unix poll or select to wait for the next event to process. In these cases, asynchronous events simply causes activity on a descriptor, so the daemon would return from the poll, find out that a given descriptor is active and call a function to extract an event and possibly act on it. This mechanism is also compatible with other operating systems like Windows that do not support UNIX signals. In this case, the client registers a callback per event type that would be used.

To support this behavior, the TDP library provides also tdp_service_event that will call any pending callback that has been registered previously in an asynchronous put or get. Under this scheme, the delivery of events related to communication actions will be checked at specific point using tdp_service_event within the RT or RM code and, consequently, the callback function will be called at a well-known and (presumable) safe point. The tdp_service_event function must be called whenever an activity has been detected in the *tdp handle*. It will identify which kind of event has been delivered and call the associated action function.

Pseudo-code for an example of the use of this asynchronous communication services is illustrated below:

```
tdp_fd = tdp_async_get (tdp_handle, "pid",
      &pid, my_callback1, my_arg1);
tdp_fd = tdp_async_get (tdp_handle,
      "executable_name", &exec_name,
      my_callback2, my_arg2);

/* main polling loop of the tool */

poll ( );
for i = 1 to descriptors
   /* first, the tool processes all events
      related to other descriptors */
   process_event (fd);

/* callbacks registered for completed TDP
   functions will be processed here */
tdp_service_events();
```

In this example, tdp_service_event would call out to my_callback1 or my_callback2 depending on which get action has been completed, respectively.

4 PARADOR: PROTOTYPING TDP

As an initial test of TDP protocol, we chose the Condor batch system and the Paradyn Parallel Perfor-

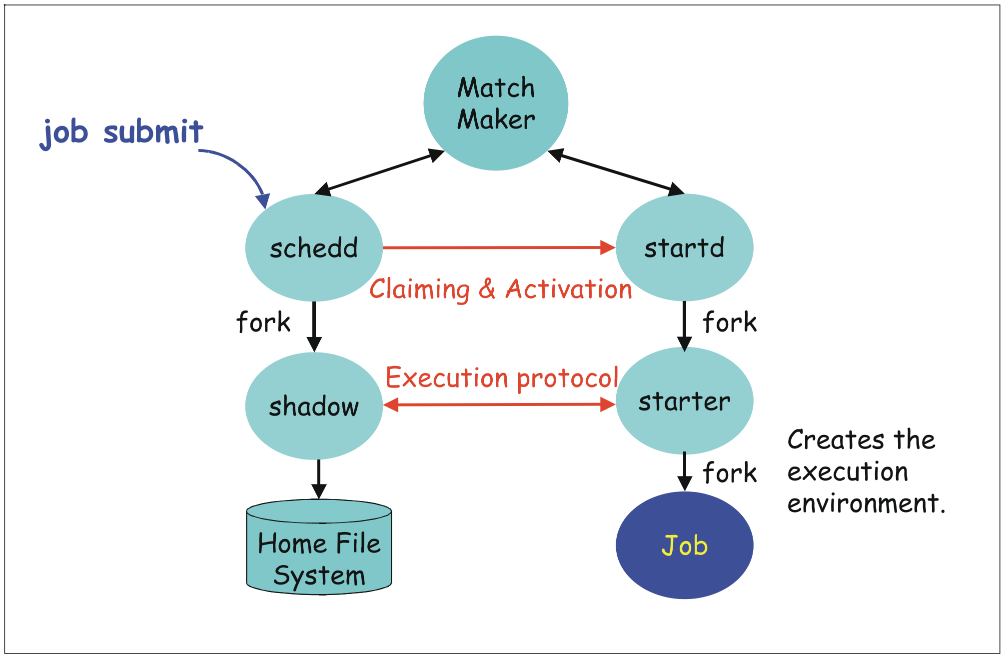


Figure 4: Condor Structure

The schedd and shadow run on the user's local machine and startd and starter run on the remote machine.

mance Tool as representative examples for resource manager and run-time tool. We outline the structure of each system and describe how they were modified to work together using TDP.

4.1 Condor structure

Condor is a widely-used system for scheduling jobs to run in distributed cluster and Grid environment. It provides all the mechanisms needed to submit jobs and run them remotely, including checkpointing and remote file access. Using Condor terminology the local host is the *submit machine* and the remote hosts are called *execution machines*. The submission of a job and the interaction between different Condor daemons is illustrated in Figure 4.

In the following paragraphs, we will describe the role of the different Condor daemons. We start by describing the daemons that run on the local machine (condor_schedd and condor_shadow) and continue with the daemons that run on the remote machine (condor_startd and condor_starter). The local daemons are:

❏ *condor_schedd:* This daemon represents resources requests to the Condor pool. Any submit machine needs to have a condor_schedd running. Basically, condor_schedd takes care of the job until a suitable and available resource is found for the job. The condor_schedd spawns a condor_shadow daemon (described below) to serve that particular request.

❏ *condor_shadow:* This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe, which perform remote system calls, do so via the condor_shadow. Any system call performed on the remote execute machine is sent over the network to the condor_shadow which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job.

The remote daemons are:

- *condor_startd:* This daemon represents a given resource (namely, a machine capable of running jobs) in the Condor pool. The condor_startd runs on each machine in your pool on which you wish to be able to execute jobs. When the condor_startd is ready to execute a Condor job, it spawns the condor_starter, described below.
- *condor_starter:* This program is the entity that spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter sends back any status information to the submitting machine, and exits. Together, the condor_startd and condor_starter form the RM.

Figure 4 shows also another element called the *match_maker*. It represents the entity that is responsible for finding a suitable machine on which to run the job.

The matchmaking algorithm is responsible for locating compatible resource requests with offers. When a compatible match is found, the matchmaker notifies the corresponding job and machine about it. Once a suitable matching is notified to the schedd, it contacts the corresponding startd. This is known as a *claiming protocol*, and either party may decide not to complete the allocation. There is another condor daemon, called the *condor_master* that is present on both local and remote nodes; its job is to keep track of the other Condor daemons. Figure 4 shows the existing relation between all these components.

4.2 Paradyn Structure

Paradyn is a performance profiling tool for parallel and distributed programs. Two of its major technologies are the ability to automatically search for performance bottlenecks (Performance Consultant) and dynamically inserting and removing instrumentation in the application program at run time (Dyninst). Paradyn has two main parts: the Paradyn front-end and user interface (*paradyn*) and the Paradyn daemons (*paradynd*), which are the agents that run on each remote host where the application program is running. Paradyn contains the user interface that allows the user to display performance, data visualizations, use the Performance Consultant to automatically find bottlenecks, start or stop the application, and monitor the status of the application. The paradynds operate under the control of paradyn to monitor and instrument the application processes. In TDP terminology, paradynd is the RT.

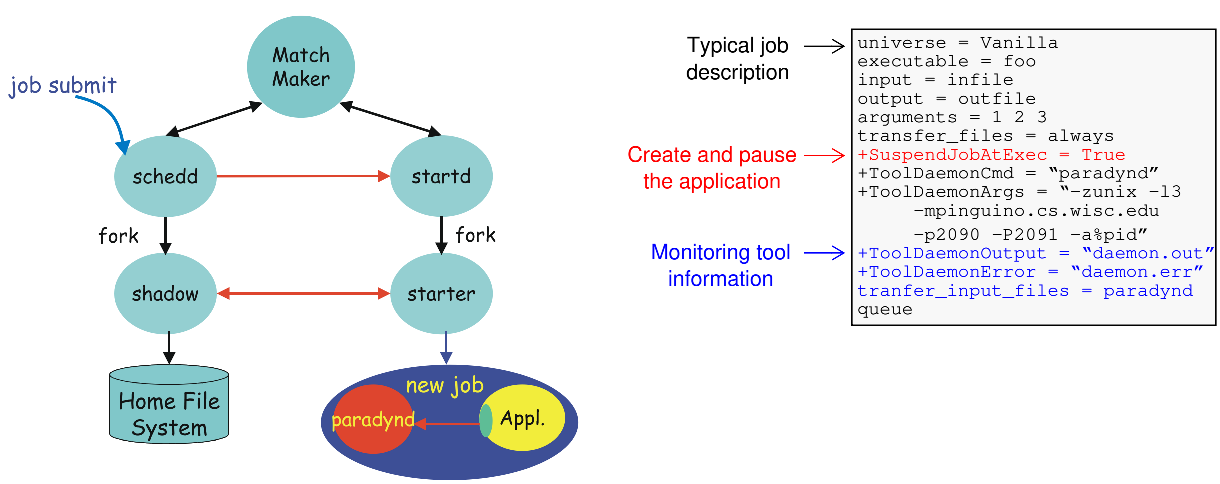
Paradyn interacts with the application program in one of two modes, as described in Section 3.1: starting the program (*create mode*) and attaching to an already-running process (*attach mode*).

❑ *Create mode:* Paradyn launches the application with the user providing information such as working directory, the application name and its arguments, and execution host machine. Once the front-end has all the information related to the new application, it will create the paradynd using either fork in the case of launching paradynd in the local machine or rsh or ssh when executing on a remote machine. Once paradynd has been successfully started, a connection is established between the front-end and paradynd. At this point, the paradynd is ready to launch the application process by forking a new process. Before allowing the user to start the application, some initialization is done:

- the paradyn run-time library is loaded into the application process,
- paradynd parses the executable to discover symbols and find potential instrumentation points, and
- a connection is established between the application and the paradynd.

After these steps, the user is able to control the execution of the application from the front-end.

❑ *Attach mode:* The user specifies the host and process id of the application process and the front-end launches a paradynd by either forking it or executing an rsh/ssh. In this case, paradynd will attach to the application process and, once the attach action has been done, it will pause the application and perform the same actions that were previously described for create mode.



A: The daemon structure **B: The Condor submit file with new entries**

Figure 5: Paradyn Running with Condor using TDP

4.3 ParadoR: Combining Both Worlds

The process control of both Paradyn and Condor were modified to use the TDP library. While these modifications involved some re-arranging of the related code in each system, the total code involved was less than 500 lines. A Condor user can submit a job that will create the application process (paused), create the paradynd, and provide paradynd with enough information that it can contact its front-end.

The code changes in Paradyn and Condor fell into two categories, rearrangement of basic operations to fit the TDP model and addition of TDP library calls. The rearrangement of operations has no net effect on the behavior of Paradyn or Condor; this rearrangement separates functions that were once combined. These changes can be considered permanent. The addition of TDP library calls allows Paradyn and Condor to operate with TDP, and these calls are only invoked in a TDP environment.

The prototype described in this section has been developed under the *create* mode, where the resource manager daemon (*condor_starter* in this case) creates both the RT *and* AP from the beginning. In addition, the prototype managed only the Local Attribute Space (LASS) at the remote host; no management of global attributes were included.

The logical view of this approach is depicted in Figure 5A. From the Condor point of view, the new job consists of two entities: the application process and paradynd. In the current prototype, new commands in the *job submit file* are used to notify Condor that the application process must be created but not started to allow paradynd to monitor the application process from scratch. For this purpose, the *SuspendJobAtExec* directive must be introduced in the Condor submit file as is shown in Figure 5B (line 7). Furthermore, the submit file must also contain all the information about the RT (in this case the paradynd). For this purpose, a set of lines initiated with the string *ToolDaemon* are introduced. These lines outline all the information needed to launch *paradynd* and are equivalent to the description of a regular job. That is, instead of *Arguments,* one will use *ToolDaemonArguments*, instead of *output*, one will use *ToolDaemonOutput,* and so on.

In our tests, the Paradyn Front-end was started first. This step was required because the front-end publishes two port numbers that paradynds must use to connect to it. As seen in Figure 5, port numbers were manually included in the submit file (-p2090 and –p2091) and the starter passed them directly to paradynd as arguments at starting time. In a complete TDP framework, port arguments should be published by Paradyn front-end and disseminated to remote sites as attribute values. On the other hand, the application *pid* was communicated using the Local Attribute Space. We used the -a%pid notation as a temporary mechanism to show which information the starter should put into LASS and which information should paradynd get from there. This attribute is used by paradynd to know it is running under the TDP framework. A more expressive mechanism should be defined by each resource manager or run-time tool in a real scenario. Once Paradyn front-end was created, it did not carry out any further action to create paradynds or application processes. This work was left to Condor by submitting a job like the one in Figure 5B. The front-end waited until Condor found an available machine to run the application.

Next, we briefly sketch how the TDP functions were included in both Paradynd and Condor to work under TDP. Figure 5A describes the daemons' behavior in the remote host. Once the Condor claiming and activation protocols are completed, a remote machine is ready to accept the submitted job. The Condor *startd* creates a *starter*, which will be in charge of the processes involved in the *new job*. In our case, the *new job* comes from a special *job submit file*, which includes the extra arguments as described earlier. These arguments are parsed by the *starter,* which detects that monitored job should be launched. Figure 6 shows the four steps that are followed by both *starter* and paradynd to complete the launching sequence:

- ❑ Step 1: The *starter* executes `tdp_init` to create the LASS through which starter and paradynd communicate. Once the TDP framework has been initialized, the starter launches the application process using `tdp_create_process` with a `paused` argument to indicate that the application process must be stopped before starting execution. In Unix terms, this means that the application has been stopped after executing the pair `fork/exec` calls and, consequently, the libraries dependencies have not been already loaded and initialized. At this point, the paradynd can not yet safely introduce its instrumentation points.
- ❑ Step 2: The starter launches the paradynd by using the `tdp_create_process` function but, in this case, the `paused` option is not used and the paradynd is created normally. When the paradynd parses its arguments, which were specified in the *job submit file*, it does not find any application process reference. Paradynd assumes then that it is working under a TDP framework.
- ❑ Step 3: At this point, paradynd calls `tdp_init` to contact the LASS. Once the contact has been successfully accomplished, paradynd immediately asks for the application pid. For this purpose, it calls a `tdp_get` with a PID attribute. Since `tdp_get` is a blocking function, paradynd is blocked until the

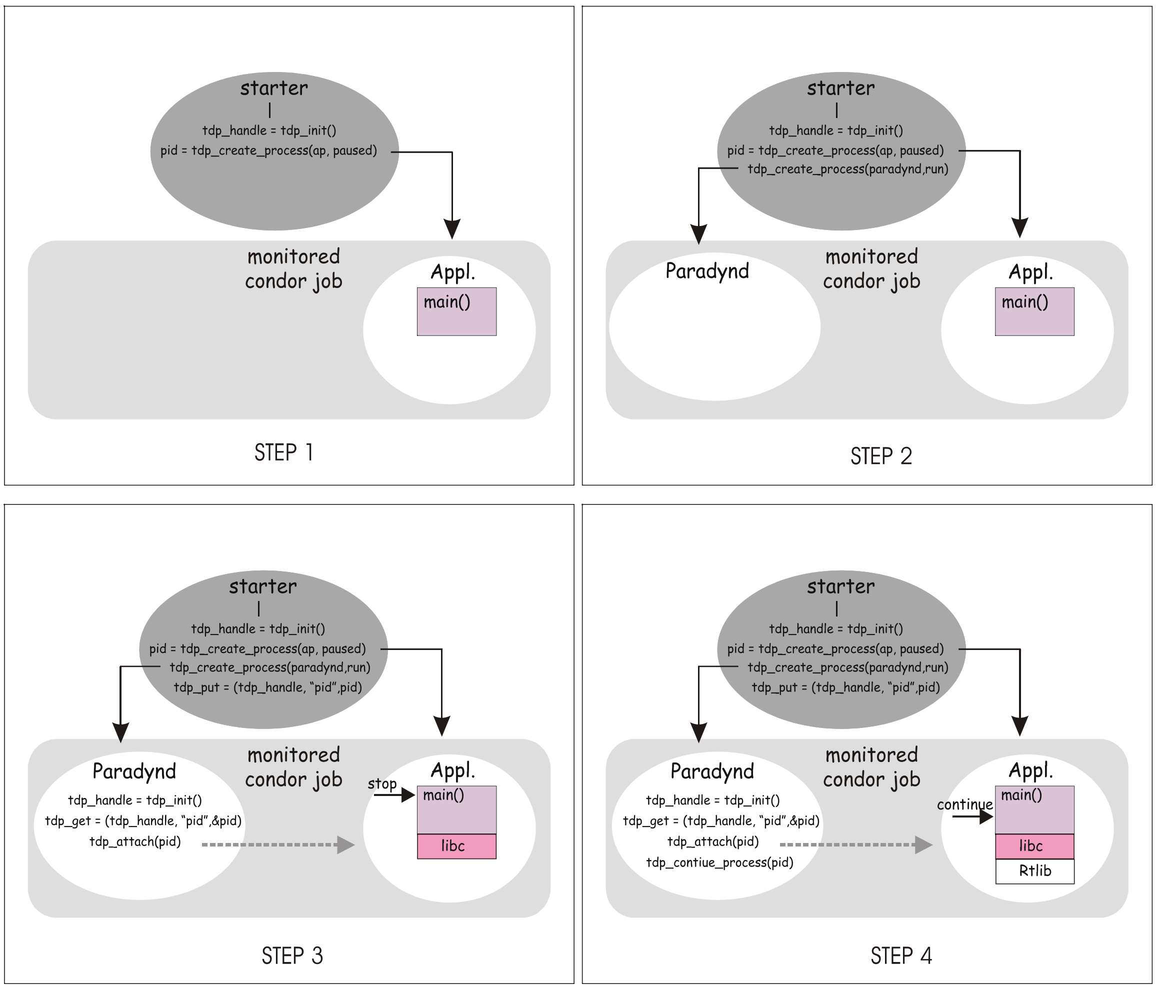


Figure 6: TDP Function Calls from the Condor and Paradyn Sides

starter stores in the LASS the corresponding application pid using `tdp_put`. Once paradynd receives the queried pid, it attaches to the application by calling the `tdp_attach` function. Afterwards, paradynd will run the application process until the beginning of the `main` function by issuing a `tdp_continue_process` call.

- Step 4: Paradynd will have the control of the application execution as usual.

Condor defines six different execution environments, called "universes", to run applications. Each universe is chosen according to the type of application that the user wants to run and specified by the *universe* attribute in the submit file. Slightly different mechanisms are used by the starter in each universe to spawn the application. Our prototype was demonstrated using the Vanilla and the MPI universes, which have similar starters.

The Vanilla Universe is used to run sequential applications, when no specific restrictions are applied to the job (i.e., any sequential job that runs outside of Condor will run in the Vanilla Universe without modification). When a vanilla application is run under the TDP framework, the starter creates the application and paradynd following the order depicted in Figure 3A. The paradynd is blocked in a tdp_get operation until the application *pid* is put by the starter into the LASS. The Paradyn front-end then is updated with the information about the application and the user is in control of the application as usual.

The MPI Universe is used to run parallel programs written with MPI. More specifically, applications must be compiled with the MPICH ch_p4 version of MPI. In this case, the submit file also specifies the number of nodes to be used in the parallel job. The application does not start until a suitable number of machines are allocated by Condor. Then, a first process (called "master process") is started. In MPI terminology, this process has rank 0. A paradynd is created afterwards, information is exchanged between starter and paradynd using the LASS, paradynd attaches to the process and, similarly to how we described for the Vanilla universe, control is passed to the user through the front-end. Once the user issues the run command, the rest of processes from the application are created with a paradynd attached to each one of them. Processes are created and stopped, paradynds attach to them and, after reporting to the front-end, they immediately issue a run command (using TDP_continue). At this point, the user is able to further steer and analyses the execution of the application as usual by using the commands from the Paradyn front-end.

The benefits of distributed resource sharing are well established, and numerous software environments and toolkits have evolved in recent years to support this mode of computing. Grids, which are considered to be the most generalized metacomputing systems, have gained tremendous popularity recently as enabling secure, coordinated, resource sharing across multiple administrative domains, networks and institutions.

5 Conclusions

Despite the potential benefits of large distributed systems, it is commonly accepted that they are inherently more complex than existing parallel systems or local-area clusters. In these large-scale distributed systems, resource managers play a crucial role as they are responsible for providing basic services to guarantee the execution of applications in remote resources. On the other hand, the use of on-line monitoring tools is an important approach for finding effective solutions to performance problems and to ensuring application reliability. Reliability and performance problems are not restricted only to user applications but also to the whole set of components that are commonly referred as system middleware. Subsequently, the use of on-line monitoring tools is extensible to these middleware services.

Large-scale distributed environments imply a new scenario that requires that both resource managers and monitoring tools be aware of the existence of one another and be prepared to execute in such conditions. This paper described our early experiences with TDP (Tool Daemon Protocol), a standard interface that aims to improve interoperability between resource managers and monitoring tools. By interoperability, we refer to the ability of different tools and resource managers to cooperate in controlling user applications by using common services and communication mechanisms. TDP is based on a small set of functions that are used both by resource managers and monitoring tools to create and control application processes. Additionally, it manages a common Attribute Space that is based on a flexible and extensible mechanisms that enable any pair of resource managers and monitoring tools to communicate effectively. The Attribute Space is used not only to exchange basic configuration and application specific information, but also to notify the occurrence of run-time events related to the application execution.

A first prototype of TDP has been applied to the Condor batch system and the Paradyn Parallel Performance Tool as a proof of concept. Appropriate daemons of Condor and Paradyn were modified to work together using the TDP library. As a result, we were able to run jobs in a Condor pool (both sequential and MPI) in which the job was also monitored and controlled by Paradyn. This prototype focussed mainly on interoperability problems between a resource manager daemon and a run-time monitoring tool daemon at the execution site (where the Local Attribute Space is used).

Acknowledgements

This work has been supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, Lawrence Livermore National Lab grant B504964, VERITAS Software, the *Dirección General de Universidades* under grant PR2001-0425 and the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) under contract TIC2001-2592. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] S.M. Balle, B.R. Brett, C.-P. Chen, D. LaFrance-Linden. A New Approach to Parallel Debugger Architecture. *Sixth International Conference PARA 2002*, Espoo, Finland, June 2002. Published as *Lecture Notes in Computer Science* **2367**, J. Fagerholm et al (Eds.), Springer, Heidelberg, June 2002, pp. 139–149.

[2] R. Butler, W. Gropp, and E. Lusk, “A Scalable Process-Management Environment for Parallel Programming”, *EuroPVM/MPI 2000*, Balaton, Hunger, August 2000. Spring Verlag LNCS 1908.

[3] N. Carriero and D. Gelernter, “Linda in Context”, *Comm. of the ACM* **32**, 4, April 1989, pp. 444-458.

[4] Cray Computer Inc., “NQE Users Guide”, Version 3.2, January 1997.

[5] Etnus LLC, “TotalView User’s Guide”, Document version 6.0.0-1, January 2003. <http://www.etnus.com>

[6] D.A. Evensky, A.C. Gentile, L.J. Camp, and R.C. Armstrong. Lilith: Scalable Execution of User Code for Distributed Computing. *Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC ‘97)*, Portland, Oregon, August 1997, pp. 306–314.

[7] I.T. Foster and C. Kesselman, “The Globus Project: A Status Report”, *Seventh Heterogeneous Computing Workshop*, Orlando, Florida, March 1998.

[8] A.S. Grimshaw and W.A. Wulf, “Legion - A View from 50, 000 Feet”, *5th International Symposium on High Performance Distributed Computing* (HPDC ’96), Syracuse, NY, August 1996.

[9] IBM Corporation, “Load Leveler Users Guide”, Version 1.2. 1995.

[10] T. Ludwig, R. Wismüller and M. Oberhuber, “OCM - An OMIS Compliant Monitoring System”, *Third European PVM Conference*, München, Germany, October 1996, Springer Verlag LNCS 1156.

[11] S. Madden, M.J. Franklin, J.M Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, Massachusetts, December, 2002.

[12] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn Parallel Performance Measurement Tools”, *IEEE Computer* **28,** 11, (November 1995). Special issue on performance evaluation tools for parallel and distributed computer systems.

[13] M.J. Mutka, M. Livny, and M.W. Litzkow, “Condor - A Hunter of Idle Workstations”, *8th Int’l Conf. on Distributed Systems*, San Francisco, Calif., June 1988.

[14] Adrian Nye, **Xlib Programming Manual**, 3rd edition, O’Reilly and Associates, Inc., July 1992.

[15] Platform Computing Inc, “LSF Users Guide”.

[16] P.C. Roth, D.C. Arnold, and B.P. Miller, “MRNet: A Software-Based Multicast/Reduction Network For Scalable Tools”, *SC 2003*, Phoenix, AZ, November 2003.