1986

The Use of Services in the TILDE Environment

Craig E. Wills

Report Number:
86-656

http://docs.lib.purdue.edu/cstech/568

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
THE USE OF SERVICES IN THE TILDE ENVIRONMENT

Craig E. Wills

CSD-TR-656
December 1986
The Use of Services in the TILDE Environment*

Craig E. Wills
Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD-TR-656

December 18, 1986

Abstract

As computing environments grow from a single machine to a distributed environment of machines connected by networks, the capability to access remote resources in a transparent manner is desirable. Transparency means that how the resource is accessed and where it is located does not need to be known by the user of the resource. This work looks at transparently extending command execution from a single machine to a multi-machine computing engine. Rather than continue to treat commands as executable files, as done by many distributed systems, we investigate a new approach that treats all commands as services. Our approach separates the identification of a service from the performance of a service. This abstraction allows details of service invocation to be hidden from the user, and allows the same service to be given the same name regardless of how it is provided or what is the processor type of the machine providing the service. In this paper we model the essential components of command execution, and detail our experience with the approach to treat commands as services in a distributed environment.

*This work was supported in part by grants from the National Science Foundation (MCS-8219078), SUN Microsystems Incorporated, and Digital Equipment Corporation.
1 Introduction

As computing environments grow from a single machine to a distributed environment of machines connected by networks, the capability to access remote resources in a transparent manner is desirable. Transparency means that how the resource is accessed and where it is located does not need to be known by the user of the resource. The work described in this paper is specifically concerned with understanding issues and problems of performing computations in a multi-machine computing engine. A computing engine is a cluster of heterogeneous machines loosely coupled with high-speed local area networks, which serves as the environment for research into distributed computing in the TILDE project [CKTM84]. More specifically, this work is part of the DASH project, which looks at how the user accesses the distributed environment [Kor84].

In a computer system, computations are commonly performed by executing named commands. A command is a named, user-level operation that performs a defined computation when invoked. In most single machine operating systems, a command is an executable file that is invoked by loading it into memory. Our work looks at transparently extending command execution to a multi-machine computing engine. Rather than continue to treat commands as executable files, as done by many distributed systems, we investigate a new approach that treats all commands as services. We define a service as a named computation that is provided by one or more machines in the network. Our approach separates the identification of a service from the performance of a service. This abstraction allows details of service invocation to be hidden from the user, and allows the
same service to be given the same name regardless of how it is provided or what is the processor type of the machine providing the service.

Our research relates to other work into multi-machine command execution. In present production computing environments, such as UNIX 4.3BSD, the computing engine at a particular site consists of a number of autonomous machines with explicit facilities for accessing remote files and executing remote commands [UNI86]. In addition, research is being done to combine the computing facilities of a set of workstations [SH82, TLC85, Hag86]. Work on multi-machine command execution has been done as part of research into distributed operating systems [WPE*83, CZ83, ABLN85]. Other work has also looked at sharing computational facilities between machines at the procedure level using a facility called remote procedure call [BN84]. These systems will be discussed in more detail as we describe our work.

In this paper we present our approach for the use of services in a distributed environment and describe our experiences with services in the TILDE computing engine. The initial portion of the paper presents an overview of the TILDE environment, models the essential components of command execution, and describes our approach for command execution in a distributed environment. The latter portion describes a prototype implementation we have built, discusses what we learned from the implementation, and summarizes our results.
2 The TILDE Computing Environment

The TILDE project is concerned with problems in the development of distributed computing environments. The TILDE system employs a process-oriented model of computation much like the UNIX operating system [RT74]. Processes are used to access system resources, communicate with other processes, and spawn child processes for execution of additional tasks. In addition, nontransient processes, servers, are used to control specific resources within the TILDE system.

An important goal of the system is to incorporate transparency in order to present the user with an integrated interface to a distributed network of heterogeneous resources. Work on transparency has proceeded in two directions: transparent command access, the topic of this paper; and transparent file naming, the topic of this section.

The Tilde file naming mechanism breaks the name evaluation procedure into two components: a per-process local naming environment, and a global, access transparent mechanism [CM85,Dro86]. Rather than a global or hardware-dependent naming convention, the Tilde naming system organizes network files into collections of related files known as Tilde Trees. Each Tilde Tree is organized in a hierarchical manner, much like the file system in UNIX. For example, the name of a Tilde Tree would be \texttt{~system} and a file within the Tree would be \texttt{~system/db/passwd}. Each Tilde Tree name is bound to an underlying Medusa Name, which is a universally known name, independent of any particular network component. Associated with each process is a Tilde Forest, a set of bindings between Tilde names and Medusa Names that describes the file naming environment.
for the process.

To investigate Tilde naming, an implementation was constructed by modifying a version of the UNIX operating system running on a network of VAX computers. The implementation incorporated the Tilde naming mechanism into the UNIX kernel and associated with each process a Tilde Forest that was used to evaluate all file names. In order to gain access to remote files, the Network File System (NFS) was used as the underlying file access mechanism [SMI85]. NFS allows file systems from remote machines to be mounted in a local machine's file name space. For the implementation, file systems between machines were mounted and named in a similar fashion on all machines in order to ensure the same Medusa Name on one machine referred to the same physical file as a Medusa Name on another machine (see [Dro86] for more details). Thus, the Tilde file naming environment was supported by an underlying network-wide remote file access mechanism.

3 Action Execution Model

Our work in the TILDE project has focused on a transparent command execution mechanism. Not only is it important to provide the user access to files located on other machines as is done with systems such as NFS and Tilde naming, but the user may also want to perform computations that are not available on the local machine, or may be performed more efficiently on a remote machine.

To aid in investigating command execution in a distributed environment we define a model that identifies the essential parts of executing an action in a
system. In describing the model, we use the more general term action, rather than command, to refer to an operation that is performed. Once we have defined the model, we will use it to investigate the execution of commands in a distributed system.

The model is based on the following questions that are used to distinguish the important components for the performance of actions in a system.

1. *When* should this action be performed?
2. *What* action should be performed?
3. *Which* action should be used given a partially specified action name?
4. *Where* should the action be performed?
5. *How* is the action going to be performed?

The first two questions must be answered by the user of the system. An action can be performed immediately or it may be performed when a set of conditions become true [KW86]. A simple example is a user interacting with a command interpreter who decides he immediately wants to list the files in his current directory. He must make a "mental mapping" between what action he would like to perform and how to specify the name of the action in the system.

This research concentrates on the last three questions that are answered once the user has specified the name of the action. These questions are answered by three components of the action execution model. The components take a partially or completely specified action name and proceed to carry out the appropriate action. First, a partially specified action name is *resolved* to an action name. This component defines a mechanism for mapping an incomplete action specification
to a particular action within the system. Second, the action is located on one of the machines in the system. The location component is needed when more than one machine is available to perform an action. Finally, the action is bound to the user’s current state within the system and invoked, which causes the action to actually be carried out. The process of resolving, locating, and invoking an action (partial or complete name) is denoted as action execution.

To illustrate the model we can look at how each of these components is provided when using a command interpreter to specify commands for the single machine UNIX operating system. Given a partial command name

- it is resolved to an executable file name using a search path,
- then trivially located on the machine,
- and invoked by loading it into memory.

More important for our research is how these components are provided in distributed systems. In systems such as LOCUS [BP84], and the V-System [VDG86], the UNIX model of naming commands the same as files has been retained, except more than one machine may be able to execute a particular file. If the system contains multiple types of processors then each executable file is “tagged” with an extension for each processor that can execute the command. When the command is invoked the correct tagged file is chosen. For these systems, the components of the action execution model are provided as follows. Given a partial command name

- it is resolved to an executable file name using a search path,
- then a machine is located that can run the file,
- and the file is invoked on the appropriate machine.
4 A New Approach

To investigate the issues of distributed command execution, our research concentrates on a new approach that no longer maintains the direct link between commands and executable files. Instead, we treat all commands as services, managed by servers, that are offered by one or more machines in the computing engine. The use of servers to manage a set of resources such as files, printers, and windows is common in distributed systems. This work is new because it focuses on the use of commands as another type of resource managed by servers, in contrast to most distributed systems that retain the notion that each command is an executable file.

To provide the facilities for service execution, each participating machine provides two servers, a service server and a cycle server. The service server is a name server that maintains a database of the name and other information for each service available on the machine. This information is used when client processes that wish to execute a service, such as a command interpreter, query the service server to locate the service. Partially specified service names may be used and cause the service server to first resolve the name using a search path mechanism. Once a service is located for execution, the cycle server on the selected machine is contacted to invoke the service. The cycle server knows how to actually perform the service and is responsible for setting up the runtime execution environment of the service. From the viewpoint of the action execution model, the components are provided as follows. Given a partial service name

- it is resolved to a service name through a search path resolution mechanism,
then a machine is located, using information from the service servers, that provides an instance of the service,

and the service is invoked on the appropriate machine by contacting the appropriate cycle server.

When compared to the LOCUS and V approach for distributed command execution, the approach to treat commands as services provides a number of advantages.

- It does not depend upon a shared file system between machines, thus services can be shared between machines in different file systems.

- It defines strict orthogonality between the functions of the components, so for example, how a service is located is independent of how it is invoked.

- It avoids early binding of services to files, so that how a service is actually carried out is hidden until the machine is reached that performs the service.

Treatment of all computations as services in a distributed environment raises issues about how services should be named and how the components of service execution should be handled by a service execution mechanism. The remainder of this section highlights these issues with more discussion of the issues following the presentation of a prototype implementation.

Our approach for naming services in a distributed environment is similar to Tilde file naming in that we break a service into two parts: its name and its underlying implementation. The Tilde file name space (or any other file name space) was not used because we wanted to separate how a service is named from how it is provided. Not only do we not want to be dependent upon a particular
file naming mechanism, but we also do not want to limit consideration of services that are not implemented as executable files.

Instead, services are named in a two-level hierarchy of the form: action-class where action is an indication of what the service does and class is the name for a group of services under the control of a single user. Specifically, a class name is either a standard name, such as std or local, authorized by the system administrator, or a user account name (userid) authorized by that user. Because the same service may be offered by more than one machine, the user who is the authority for a service is responsible to make sure all services with the same name provide the same function. This naming convention partitions the service name space by userids and guarantees each service name is unique as long as all userids within a computing engine are unique. Example services are:

- tex-std: a standard service validated by the system administrator.
- tex-cew: a service validated by the user cew.
- tex: a partial service name that needs to be resolved.

For a client process that wishes to execute a service, the three components of the action execution model must be carried out. Because the service servers in the computing engine know about the available services, a client process contacts the service servers to resolve and locate services. Resolution and location can be combined into a single transaction in communicating with a service server. The client process sends a message to the service server containing a partial service name and any user specific resolution information. The service server resolves the partial name to a complete name by finding a class in a search path that contains the partial service name.
How a client process that wishes to execute a service uses the service servers in
the network to resolve and locate the service is not clear. One approach is for the
client to broadcast a request to all service servers each time a service is specified
and wait for responses. Each service server then resolves the service name, if
partially specified, and sends a response back to the client which is responsible
for deciding which of the machines to use for invocation. This solution can cause
network overhead for each service execution, but allows the client process to base
the decision on where to invoke the service on the most up-to-date information.
Alternately, each service server could cache information about services in the
network. Then, a client process would only need to contact the local service
server because the server could perform resolution and location based on its
cached information. However, depending on how often the local cache is updated,
information may be outdated or wrong. Many solutions exist between these two
extremes with the "right" answer dependent on the rate of service access, service
change, and the speed and size of the network.

Regardless of how service location is performed, another question is which
machine to select given a service is provided by multiple machines? Users of the
system expect the machine to be selected that will provide the best performance.
Unfortunately, correct selection of such a machine requires information to be
known about each machine (such as its load average and processing power),
about the service itself, and about the amount and location of data. Because
some of this information may be difficult or even impossible to obtain, selection
of the best machine is not always possible.

Once the client process knows which machine to use for invocation, it com-
municates with the cycle server on that machine through a standard protocol. The client passes the name of the service and the user’s execution environment to which the invoked process for performing the service should be bound. The execution environment is the set of bindings between names and their associated objects. It includes parameters, local bindings, and global bindings. Parameters are defined at service compilation time and the objects associated with these names are bound at invocation time. These bindings include service options (such as flags) and standard data streams. Local bindings are specific to an invoked process and are initialized at invocation time. For example, in UNIX, these bindings are the environment variables for a process. Global bindings are the names bound to objects, such as files and devices, that are not part of the process itself.

One of the features of the Tilde file naming mechanism is to give each process control over its global bindings by allowing the process to manipulate its own Tilde Forest.

5 A Prototype in the Tilde Environment

To explore these ideas for service execution, a prototype implementation was built in conjunction with the Tilde naming environment described in Section 2. The prototype implements potential solutions to questions raised in the previous section on the use of commands as services in a multi-machine environment.

The prototype was implemented by adding a service server and cycle server, executing as user processes, to each participating machine running the modified UNIX kernel, which supports the Tilde naming mechanism. In our model, the
service server on each machine knows all information about all services available
on that machine. For the prototype, an implementation decision was made that
each service server would also know the names of services available from other
machines. This decision was made in order to ensure that service resolution and
location could be done on the local machine. Thus, in addition to maintaining a
local database of services, each service server advertises those services available to
other machines in the computing engine, and caches the services available from
other machines. Along with any newly available services, each service server
periodically broadcasts its load average and a measure of its computing power to
other servers. This information is used to select a machine for invocation when
more than one machine provides the same service.

The use of services in the system can be viewed from two perspectives: from
the standpoint of a programmer who wishes to provide a service, and from the
standpoint of a user who wishes to use a service. For the programmer who has a
service he wishes to provide to others, he must register it with the service servers
on the machines that can perform the service. Three client routines are used for
registration of services with a service server: `addserv`, `rmserv`, and `lsserv`. These
three routines are actually registered as local services on each machine and are
used to manipulate the available services of the machine.

`Addserv` registers a service by passing to the local service server three pieces
of information.\(^1\) First, it passes the name of a service which includes the service
class. Only the system administrator may register services with a class other
than a userid. Second, `addserv` passes the name of the executable file and the

\(^1\)All communication with a service server is done using a UDP based protocol [Pos80].

12
Medusa Name of the file's Tilde Tree that are invoked to provide the service.\footnote{Currently, under the TILDE implementation, all services are provided by loading an executable file.}

Third, it passes an indication whether this service is available to users on other machines. Some services (such as editors) may only be provided to users on the local machine because they use a device (in this case the terminal) only accessible on one machine. Other services are available to any machine that shares the Tilde file naming environment. In addition, \texttt{rmseru} unregisters services, and \texttt{lsserv} lists the available services.

To execute a service, a new library function, \texttt{sexecve()}, has been added. The new function can be substituted for the UNIX system call \texttt{execve()} in order to execute services rather than binary files. The arguments to \texttt{sexecve()} are a service name (partial or complete), the service parameters, and the environment variable values for the service to be invoked. These arguments are the same as \texttt{execve()} except a service name, rather than an executable file name, is used. The new library routine was substituted in the UNIX command interpreter \texttt{csh} \cite{UNI86}, to provide the user a familiar environment for the use of services. The modified shell is known as the service shell, or \texttt{scre}.

The use of \texttt{sexecve()} hides all details of service execution from user programs. It works by first contacting the local service server and passing a service name and resolution information in order to resolve and locate the service. If a complete service name, such as \texttt{tex-std}, is given then the service server locates the “best” machine (based on load to processing power ratio) for invocation. If a partial service name, such as \texttt{tex}, is given then the service shell checks if the given service name is actually a full path file name or a file located in the current directory. If so then the file is loaded directly.\footnote{As an escape to the UNIX system, the service shell checks if the given service name is actually a full path file name or a file located in the current directory. If so then the file is loaded directly.}
service name, such as \textit{tex}, is given then the service server first resolves this name to a complete service name. This task is performed by examining the value of the user's service set, which is passed from the client along with the service name.\footnote{The service set is maintained by the shell as an environment variable.} The service set is an ordered set of classes that are searched in order to try and complete the partial service name. For example, the service set may have a value of \texttt{cew:local:std} indicating that services from the user \texttt{cew}, local, and standard services should be used. Once a match is found the same service location algorithm is performed.

Upon locating a machine for invocation of the service, the service server passes back to the client the name of the machine. In addition, if the local machine is selected then the appropriate file name is also passed back to the client and a process is spawned that loads the file directly for invocation of the service. If a remote machine is selected then the routine spawns a helper process to serve as the client for communication with the remote cycle server.

The helper process communicates with the remote cycle server using a TCP based protocol [Pos81]. The protocol is similar to the \texttt{rsh} protocol in UNIX except the cycle server invokes the service with an execution environment passed to it rather than a default environment. When invoking a service, the helper process connects to the appropriate cycle server and sends the userid, service name and parameters, current working directory, environment variables, and Tilde Forest. The cycle server then uses the connection for exchanging input and output data with the helper process.

In order to obtain information on how to actually invoke the service, the cycle
server on the remote machine contacts its local service server and receives the Medusa Name and path of the file to be loaded. Before loading the file, the execution environment for the spawned child process is set with the Tilde Forest, the current working directory, and the environment variables passed from the client process. The Tilde Tree for the file to be invoked is also added to the Tilde Forest in order to provide access to the executable file. A problem occurs if this Tilde Tree conflicts with any in the client's Tilde Forest. The conflict is resolved by binding a different Tilde name to the underlying Medusa Name of the executable file and changing the name of the file accordingly. In either case the file is then loaded for execution.

6 Discussion

This section discusses experience gained from building and using the prototype for service execution and points to future work on this topic.

The chosen naming convention works well for the prototype. Even though services are named differently than UNIX commands, the partial names commonly used in each environment are the same. This similarity results because UNIX commands are organized in directories of binary files much like services are organized into classes. However, because the name of a service is separate from how it is performed, all executable files for a class of services do not have to reside in the same directory. This feature fits nicely with results of work with the Tilde file name mechanism, where the user frequently creates his own Tilde Tree for a particular software subsystem. Users maintain all executable files, source
files, and documentation for the subsystem in a single Tilde Tree, rather than dis-
persing these files to different places in the file system. Treatment of commands
as services allows the services from different Tilde Trees to be consolidated into
a single class.

The prototype does violate our premise that all commands are services by
allowing executable files in the user's current directory and full path names to be
invoked directly. Instead, we should have a "run" service that invokes a named
executable file. Thus, all commands would be services and any name specified
would always be invoked as a service. This solution also avoids problems of the
user accidently specifying a partial service name that happens to be an executable
file in his current directory.

A problem with the UNIX organization for commands is that potentially each
directory in the user's search path must be checked to resolve a command. This
condition leads to all executable files existing in a few, well-known directories
that contain a large number of unrelated commands. The abstraction of service
naming allows the services for software systems with a large number of services,
such as mail or revision control systems, to be grouped into their own class.
With more classes that contain services for specific software systems, the user
can choose his service set to better reflect the set of services used by himself,
and lessen the chance of unwanted conflicts occurring in the resolution of partial
service names.

One problem with our simple approach to service naming is its inability to
express functional relationships between services. For example, two services may
be equivalent, except one of the services may provide a few more options or
produce output in a different manner. Examples are two search services that
use different algorithms, but provide the same results, or two services that each
output the current time, but each in a different format. In many cases the user
does not care which service is used, but he is forced to choose one service over the
other. One possible solution is to introduce the concept of service equivalence,
which would allow users to group two or more services together as a new service.
Then, when the user executes this new service, any of the equivalence services
may be chosen.

Another naming problem is the conflict of service names as we move from
a world of autonomous machines to an environment where services are shared
between machines. For example, the standard C compiler on two different archi­
tecture machines may normally each be given the name cc-std, but if these two
machines share services then one service must be given a different name because
services with the same name must provide the same function. One possible so­
lution is to treat cc-std as a generic name that automatically evaluates to the
C compiler service on the user's "current" machine. Determination of the user's
current machine can be difficult, though, as the user accesses files from a variety
of machines.

To resolve and locate a service we have chosen to cache, and periodically
update, the load and available services from other machines in the computing
engine. This decision appears to work well because service access predominates
services changing in our environment. Caching of information maintains slightly
out-of-date information, but results in acceptable delays in order to resolve and
locate a service. If the service information was not cached, but instead a request
broadcast for each service, then network overhead would be added for each executed service, even for services that were invoked locally. More work needs to be done to determine how much network overhead is involved, and what threshold is satisfactory for the latency of a local cache.

An important problem not addressed by our prototype is scaling the service execution mechanism to a larger number of machines. For resolution and location of services, periodic broadcast, and caching of information worked in the prototype because there were a small number of machines and an underlying physical network existed to support broadcasting. On a large scale these conditions no longer hold. In future work, to investigate service use in a larger computing engine, we will try to take advantage of two observations. First, services on the local machine, or services from machines in the same file name space, are used the most. Second, services are accessed more often than they are created or destroyed. These observations point to an approach of caching the locations of frequently used services on the local machine, and caching other service information at well known machines in the network that can be queried if the location of a given service is not known locally.

Another difficult problem concerning service location is choosing the best of many machines that provide a service. In the prototype, we used a simple heuristic based on load and processing power of machines. We know such a low-cost heuristic does not provide optimal location of services, but believe the heuristic is useful to avoid locating services on heavily loaded machines.

We also found in the prototype that the simple approach of offering services either locally or globally is not satisfactory. Other factors need to be considered
to decide if a service on one machine can be used by a user another machine. These factors include whether the two machines share the same file name space, whether the service is limited for use by a particular group of users, and whether the service is only available for users with an account on the remote machine.

The cycle server, along with an appropriate client process, implements the protocol for invoking a service. The use of a separate server for invocation maintains an independence between how a service is located and how it is invoked. Thus, services can be invoked on a remote machine without first contacting the service server on that machine. We found the use of the cycle server to set up the execution environment and actually invoke the service to be a good approach in the prototype. For future work, the cycle server may not be appropriate as we think of invoking services that are implemented by other server processes. For such services, we would like to contact the appropriate server directly.

The availability of the Tilde naming implementation provided a convenient mechanism for describing the user's current file naming environment (his Tilde Forest) to the remote machine. The only problem was when conflicts occurred between the Tilde Tree of the invoked service and the user's Tilde Forest. In these cases, the Tilde name, but not the Medusa Name, of the file to be loaded was changed. If the resulting process expected to access a file using the original Tilde name then that name would be bound to the Medusa Name inherited from the client process. This conflict resolution mechanism could cause problems if the invoked service expected to have its original Tilde name bound to a particular Medusa Name. In our experience this was not a problem.

We also need to consider the use of services between machines that do not
share the same file name space. Because many existing commands in UNIX depend on access to named objects in the file system, the use of these commands as services may not be possible by users on another machine that does not share the same global environment. In such cases, services must be used that receive data from the parameters and local environment, as well as communicating data through the stream connection between the invoked service and the helper process. Alternately, a temporary global environment could be created on the remote machine by copying files at setup time. This solution depends upon being able to identify these files at invocation time and being able to give the file the same name on the remote machine. With present specification syntax that is not consistent between services this solution appears to be possible only on a special case basis.

7 Conclusion

This paper has described a model detailing the important components of command execution and shown how these components can be provided in a distributed environment. For a set of machines sharing the same global file name space, such as provided by the Tilde file name mechanism, the described service execution mechanism provides a transparent extension of the user's computing services from one to many machines. This work provides similar functionality as has been provided by other remote execution mechanisms, but is not incorporated as part of a distributed operating system. In addition, our approach is extensible to other machines outside of a shared file environment. This characteristic allows
services from any machine in the computing engine to be used in a transparent manner. On a larger scale, services from machines in other computing engines could conceivably be used.

Not only is the service model applicable to a variety of machines, but it is also applicable to a variety of computations. In addition to treating the commands of a machine as services available to other machines in the network, the service abstraction can be applied to other computations, such as remote procedure calls and network services. We can employ the same principles of resolution and location described in this paper to locate these services, and use computation specific protocols to invoke the services. These protocols may differ from the protocol described in this paper by invoking the service directly without contacting a cycle server.

Use of standard communication protocols allows services to be used between different types of machines. In addition, the treatment of commands and other computations as services can be implemented on specific systems to take advantage of efficient, system specific protocols, such as IKP in an environment of V-System machines. New protocols, such as VMTP [Che86], provide reliable datagram delivery and multicast facilities that could allow better implementations of the service location and invocation protocols.

Because this prototype used the Tilde name implementation, it was done entirely on VAX machines running UNIX. We want to move towards the use of workstations as the primary nodes of service access with hosts serving as execution servers. Much like Sun's Network File System extends the file objects available to a user at a workstation, our mechanism extends the computations
available to the user. The treatment of commands as services provides a convenient abstraction for access of existing services in a multi-machine, shared file environment, as well as a framework for access to other computations in and outside of a shared global environment.

References


