Virtual Private Grid : A Command Shell for Utilizing Hundreds of Machines Efficiently

Kenji Kaneda
University of Tokyo
kaneda@yl.is.s.u-tokyo.ac.jp

Kenjiro Taura
University of Tokyo
tau@logos.t.u-tokyo.ac.jp

Akinori Yonezawa
University of Tokyo
yonezawa@yl.is.s.u-tokyo.ac.jp

Abstract

We design and implement Virtual Private Grid (VPG), a shell that can easily and securely utilize a large number of machines distributed over multiple administrative domains. Today, many people have an access to a large number of machines across multiple subnets or geographically distributed places. These machines are managed by different administrators, and for the sake of security and administration cost, they impose various restrictions on their use. Methods to work around these restrictions are found on a case-by-case basis and require human intervention. Therefore, it increases the user's cost to utilize remote machines significantly, and consequently decreases the utilization of computational resources. VPG works around these restrictions automatically and can easily utilize a large number of machines in multiple administrative domains. We run VPG on approximately 100 nodes (270 CPUs). Experimental results show that VPG utilizes remote machines more efficiently than other job submission tools.

1. Introduction

The recent improvement of computers and networks is impressive. Supercomputers and clusters of workstation/PCs become connected with high-speed networks and spread over multiple subnets or geographically distributed places (*Computational Grid* [10]). Consequently, many people have a chance to harness a large number of computational resources.

These machines are usually managed by different administrators, who impose various restrictions on their use for the sake of security and ease of administration. Examples of these restrictions are,

Firewall A firewall protects local machines by restricting access from external hosts. For example, IP filtering restricts connections from/to hosts with a particular IP address.

Private IP A private IP is an IP address visible only within a subnet. Hosts outside the subnet cannot initiate direct connections to hosts that have only a private IP.

DHCP client DHCP is a mechanism to enable machines to extract their network configuration from a server. An IP address of a DHCP client changes dynamically whenever it extracts a new configuration.

Ad-hoc methods to work around these restrictions are found on a case-by-case basis and require human intervention. For example, hosts behind a firewall are usually reached first by logging on a gateway machine and then onto the target. For another example, accessing a DHCP client requires some database that stores its address. Situation is even more complicated when those addresses are local IP addresses. Overall, these restrictions significantly increase the user's cost to utilize remote machines, and consequently, decrease the smooth utilization of computational resources.

We illustrate the above problem with a practical scenario. Consider a network shown in Figure 1. *Harp*, *tuba*, ... in Figure 1 represent host-names and let us assume we would like to submit jobs (commands) from *tuba* to all the other hosts in this network. The network consists of three subnets including private hosts and DHCP clients. Connections are restricted by firewalls; the only allowed in-bound connection is SSH [3] to the gateway hosts (*harp*, *cscl0*, and *ise0*). Such a configuration is fairly typical.

In this network, job submissions often become cumbersome with commonly used tools (e.g., rsh, SSH, and PBS [1]).

- Job submissions from outside a firewall to inside. We typically must first log onto a gateway host ((i) in Figure 1).

- Job submissions to a host that has only a private IP address. Similarly to the above, accessing such a host requires first entering the subnet of the target machine ((ii) in Figure 1).

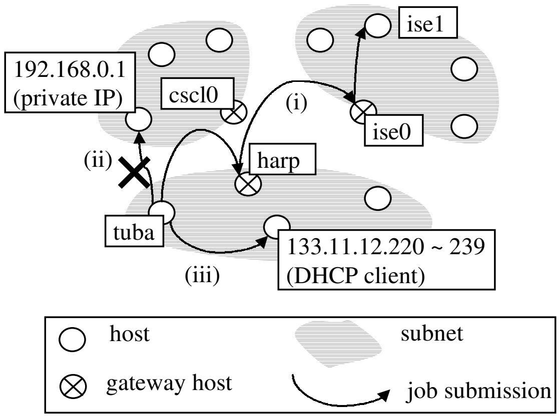

Figure 1. Practical example

- Job submissions to a DHCP client. We must somehow obtain the current IP address of the machine ((iii) in Figure 1).

When the number of machines is small, it may be possible to work around restrictions by human intervention. The user remembers intervening gateway machines to each host, and keeps some ad-hoc database to keep track of DHCP clients. However, this method obviously does not scale to a large number of machines and subnets. The user would like to have a solution in which all the machines can be reached directly and transparently, with names fixed over time.

One way to implement such a transparent job submission is to connect all the hosts via connections, and relay commands through a path on the graph. This is how VPG basically works, but there are issues in connection setup. As the number of the machines increases, the number of allowed connections each machine can initiate also increases, so simply creating all allowed connections has a severe scalability limitation. Thus, we would like to select a small number of connections to make the graph connected, but manually configuring such connections would be too cumbersome for users, especially when the available machines change from time to time.

To summarize, it is cumbersome to submit jobs with commonly available tools, and is not trivial to automate construction of the connected graph. There must be an algorithm that keeps only necessary connections in order to reach all the available hosts.

The goal of this research is to enable a user to utilize all his/her machines through a shell at his/her local host. Thus we design and develop *Virtual Private Grid (VPG)*, a shell for Grid computing, which automatically works around the aforementioned restrictions. VPG dynamically constructs a spanning tree among hosts and provides the user with a simple view in which any host can be reached with a name that does not change over time, even if its IP address does. As a result, the user can easily harness a large number of machines in multiple administrative domains.

```
path@nickname
path@nickname > file@nickname
path@nickname < file@nickname
path@nickname | path@nickname
```

Figure 2. Shell syntax

The remainder of this paper is organized as follows. Section 2 describes the design and the user interface of VPG and Section 3 its implementation. Section 4 details the algorithm for spanning tree construction. Section 5 shows experimental results. Section 6 mentions related work. The final section summarizes the paper and states future work.

2. Virtual Private Grid (VPG)

2.1. Features

The following summarize functions provided by VPG.

- It gives each host a (per-user) unique name that doesn't depend on a DNS name or a fixed IP address. (*nicknaming*).
- It provides a job submission to any nicknamed host.
- It provides a redirection from/to a file on any host.
- It provides a pipe between commands executed on any host.

We make the above functions accessible by the combination of the simple shell syntax and existing commands (See Figure 2). A user can submit a job to a remote host by adding `@` followed by its nickname. Standard input/output of a program can be connected to output/input of a program running on a different host (i.e., pipe over network).

With VPG, the user can directly access remote machines which would have been several hops away from his/her local host. VPG implements this by constructing a spanning tree among available machines without changing administrative policy. In addition, VPG can tolerate dynamic addition/removal of machines by reconfiguring the tree automatically.

2.2. Submission Example

We show two examples of remote job submissions with VPG. Assume the same network topology as Figure 1 and that *tuba* is the home host, which a user initially logged in.

VPG constructs a tree on the network and gives each host a unique nickname. Figure 3 illustrates a tree constructed

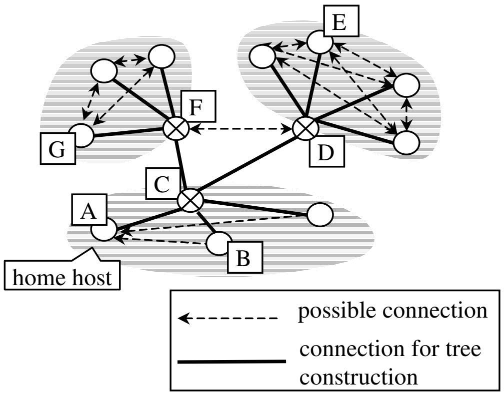

Figure 3. Submission example

by VPG. A dashed arrow from host u to v is a possible connection which u can initiate to v. A solid line between u and v is a bi-directional connection which u and v keep permanently between them.

A, *B*, . . ., and *G* in Figure 3 represent nicknames, which do not change among execution of commands. For example, *A* and *B* are nicknames of *tuba* and the DHCP client in Figure 1 respectively.

The user submits jobs from the home host (i.e., host *A*).

`ps@B`
: This command is executed on the DHCP client whose nickname is *B*. Because host *A* and *B* belong to the same subnet, the request is sent to *B* directly, in the same way as common remote job submission tools (e.g., rsh).

`tar@E -c file | tar@G x`
: This command archives **`file`** on *E*, transfers it to *G*, and extracts it on *G*. Because host *A*, *E* and *G* belong to different subnets, VPG automatically detects a forwarding route ($A \to C \to D \to E$), submits the first **`tar`** command to *E* through the route, and execute it on *E*. Similarly, the second **`tar`** command is submitted from *A* to *G* (its route is $A \to C \to F \to G$), and the output of the first command is transfered from *E* to *G* (its route is $E \to D \to C \to F \to G$). As in the UNIX pipe, they are executed in parallel.

3. Implementation

3.1. Overview of Implementation

A brief overview of implementation is as follows.

1. VPG daemons boot on all the available hosts a user wishes to use. These daemons can boot in any order. Configuration information that should be given to daemons is described in Section 3.2.

2. The daemons create and keep some bi-directional (TCP) connections. They create connections necessary to make a single connected graph of all the hosts by exchanging information with neighbors. In this scheme, hosts with dynamic and/or private IPs become reachable because they initiate bi-directional connections to the outside.

3. Eventually, the spanning tree among the hosts is constructed, and daemons stop creating connections.

4. A shell starts on the home host. It keeps track of the topology of the whole network. It detects a route to any participating host in order to submit jobs, redirect input/output of a command, and so on.

5. Hosts and connections may fail, or new hosts may become available. Whenever the network topology changes, the daemons create a new spanning tree by adding/removing connections to make all the hosts available.

Many firewalls block connections to all the unprivileged ports, and in this case, SSH is usually the only way to log on to hosts behind them. VPG daemons use *SSH port forwarding* in order to connect to such machines when regular connections are not allowed.

3.2. Daemon Configuration

Currently, VPG daemons need the following configuration information about the network.

nickname Each daemon needs a *nickname*, a name of the host that does not change over time. Each nickname must be unique throughout all the available hosts.

port number A daemon tries to contact other daemons at this port.

list of connections Network configuration is specified by a list of connections each daemon can initiate. For example, if a host has only a private IP address, connections to this host from outside its subnet will not be listed. Similarly, connections *to* DHCP clients will not be listed, but those *from* them will. Each connection is labeled either as 'regular' or 'ssh', the former indicating it can be a regular connection and the latter it should tunnel through SSH. Daemons construct a spanning tree by selecting only a small number of connections from this list.

The amount of configuration is fairly large and may be cumbersome for users. Note, however, that a configuration file is typically written only once, and need not be very precise. For example, it does not affect correctness to add non-existing hosts to the list. They are simply regarded as down hosts, and our algorithm tolerates them. Similarly, it does not affect correctness to list connections that are actually blocked and not to list connections that are actually possible.

They are currently required only for performance. Listing too many hosts or connections that do not exist causes daemons to try many connections that only fail. We are planning to address this issue in our future work. The tree construction algorithm operates in such a way that each daemon only needs information about allowed connections adjacent to it. Thanks to this distributed nature of the algorithm, probing necessary configuration online should not involve much technical difficulty.

3.3. Other Implementation Issues

In this section, we describe several implementation issues which have not been mentioned yet.

SSH port forwarding Because daemons need to create SSH connections automatically without entering a user's password, SSH must use the public key authentication with an empty pass-phrase.

daemon sharing An original design criteria of VPG is to run daemons in the user level, thereby not requiring changes to system administration. This design, on the other hand, may impose a large overhead on the system if many users run their daemons. This problem can be fixed by running a daemon at the root privilege that, with an appropriate authentication, forks a user process on demand. We are going to implement VPG so that it can be run either way. This is not a peculiar problem to VPG, but a common problem in any network service. Whether a service is run with the root privilege or with individual user's privilege is a matter of choice at each host, based on the popularity of the service, system administration policy, and so on.

4. Spanning Tree Construction

In this section, we describe a network model and the algorithm to construct a spanning tree. The network model formalizes administrative restrictions (e.g., Private IP) and is used by the tree construction algorithm. The algorithm is composed of two parts: construction of self-stabilizing spanning tree and calculation of routing table. Daemons select necessary connections to form a spanning tree by using the former algorithm. The shell detects a route to any participating host for a job submission by using the latter algorithm.

4.1. Network Model

We model a configuration of a network as a directed graph $G = (V, E)$, where V is the set of hosts, and E the set of possible (i.e., allowed) connections. An edge is labeled either 'regular' or 'ssh'. Let u and v be hosts. If (u, v) is in E and labeled 'regular', u can initiate a regular connection to v. If (u, v) is in E and labeled 'ssh', u can initiate a connection to v through SSH port forwarding. In the following, labels are omitted when not important.

The real network can be modeled in this framework as follows.

DHCP client If u is a DHCP client, G satisfies the following.

$$\forall v \in V - \{u\}.\ (v, u) \notin E$$

That is, because a DHCP client has no fixed IP address, any other host cannot initiate a connection to the client.

Private IP If u is a host with a private IP address, we have,

$$\forall v \in \{\text{hosts outside the subnet}\}.\ (v, u) \notin E$$

Firewall If a firewall blocks all in-bound connections to a subnet,

$$\begin{aligned}&\forall u \in \{\text{hosts outside the subnet}\},\\ &\forall v \in \{\text{hosts inside the subnet}\}.\\ &(u, v) \notin E\end{aligned}$$

Let g be a gateway reachable via SSH from any host. Then we have,

$$\forall u \in V.\ (u, g)_{ssh} \in E$$

The above pieces of information come from the configuration file.

4.2. Self-stabilizing Spanning Tree Algorithm

We use the self-stabilizing spanning tree algorithm[7][8] for automatic tree construction. This algorithm has following features: (1) each daemon asynchronously builds a spanning tree without knowing the whole network. (2) it can construct a tree even if the network topology changes dynamically.

The algorithm regards the graph as a spanning forest, that is, a set of rooted tree. Initially, this forest consists of single-node trees (each node is a root). Starting from this state, the

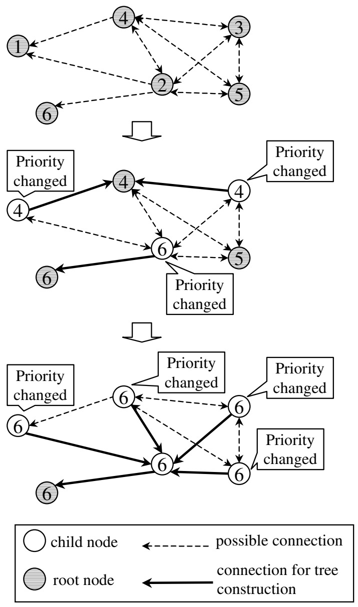

Figure 4. Process of tree construction

nodes gradually coalesce into large trees. Eventually, all the nodes in the graph form a single spanning tree. When the network topology changes dynamically, they cope with it by resetting their local states.

Each node maintains three variables: UID, $Parent$, and $Priority$. We subscript a variable with node name. UID_u is a unique identifier of node u and $Parent_u$ a node name of u's parent. $Priority$ is explained shortly.

Each daemon asynchronously connects to its neighbors specified in G, and when two daemons find them to be in different trees, these two trees are merged. $Priority$ determines how they are merged. Omitting some details, $Priority_u$ is initialized to UID_u, and when node u notices a neighbor v that has a higher $Priority$ value, u becomes a child of v's tree, and $Priority_u$ becomes equal to $Priority_v$. Therefore, trees with higher $Priority$ overrun trees with lower ones, and finally the algorithm constructs a single spanning tree with the highest $Priority$.

Figure 4 illustrates a process of tree construction. A dashed edge from u to v means that $(u, v) \in E$. A solid edge from u to v means that v is u's parent. The value of each node indicates its $Priority$ and changes when connections are created. For example, a node whose $Priority$ is 1 initiates a connection to a node whose $Priority$ is 4, and its $Priority$ becomes equal to 4. Eventually, $Priority$ of all the nodes becomes equal to 6 and the tree is constructed.

We refer the reader to [7][8] for the detail of the algorithm.

$\mathbf{ROUTING_u}$

if (u is the home host)
then
  $ToHome_u \leftarrow u$ [A]
else if (u is a leaf)
then
  $ToHome_u \leftarrow u$'s parent [B]
else if ($|\{v \in \mathbf{Neigh_u} | ToHome_v \neq u\}| = 1$)
then
  $ToHome_u \leftarrow$ the element of the above set [C]
else if ($\exists v \in \mathbf{Neigh_u}$
  $(ToHome_v \neq \text{nil})$ **and** $(ToHome_v \neq u)$)
then
  $ToHome_u \leftarrow$ any such v [D]
else
  $ToHome_u \leftarrow$ nil [E]

where:
$\mathbf{Children_u} = \{v | Parent_v = u\}$
$\mathbf{Neigh_u} = \mathbf{Children_u} \cup \{Parent_u\}$

Figure 5. Routing algorithm

4.3. Routing Algorithm

When submitting jobs, the shell needs to calculate the shortest path to the destination host. For this purpose, the shell keeps track of the whole network topology by receiving fragments of topology information from daemons and joining them together. Every time the network topology changes, the daemons send their new information to the shell and the shell updates its topology information.

Note that at first the daemons do not know the location of the home host where the shell is running. Then, each daemon calculates a route from itself to the home host by using an additional variable $ToHome$. $ToHome_u$ is basically equal to v, where v is u's neighbor on the tree and one hop nearer to the home host than u. Only if u is the home host, $ToHome_u$ is equal to u. If u has not detected the route to the home host yet, $ToHome_u$ is equal to nil ($ToHome$ is initialized to nil).

Node u calculates $ToHome_u$ by repeating the algorithm shown in Figure 5 at regular intervals. Note that in the following, we regard the root of the spanning tree as the home host. Therefore, calculating $ToHome_u$ corresponds to detecting u's parent.

- If u is the home host, $ToHome_u$ becomes equal to u ([A] in Figure 5). Node u does not need to calculate a route to itself.

- If u is a leaf of the tree, u has the only one neighbor on the tree. Therefore, u must send a message to the home host via the neighbor and regard the neighbor as

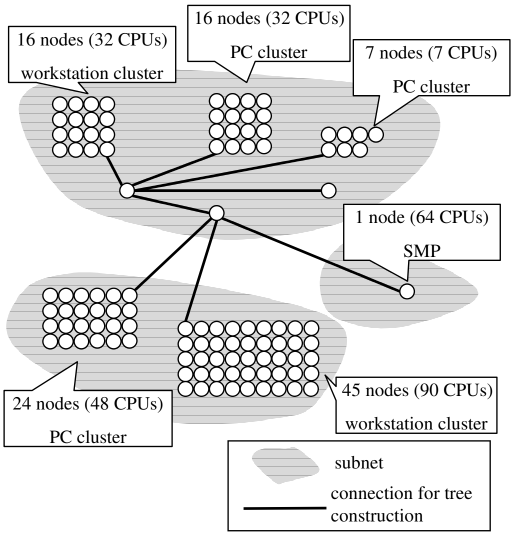

Figure 6. Experimental environment

its parent ([B] in Figure 5).

- Note that if u's neighbor v satisfies $ToHome_v = u$, v is u's child. If all u's neighbors except v are u's children, u must be u's parent. Node u sends a message to the home host via v ([C] in Figure 5).

  If there are more than two neighbors which are not u's children, u cannot judge which node is actually u's parent. Thus $ToHome_u$ does not change.

- If u's neighbor v is not u's child and v satisfies $ToHome_v \neq$ nil, v has already detected a route to the home host and the route does not go through u. Therefore, u can send a message to the home host via v. Node u regards v as its parent ([D] in Figure 5).

- If u does not satisfy all the above conditions, u cannot find a route to the home host. Thus $ToHome_u$ becomes equal to nil ([E] in Figure 5).

5. Experiments

5.1. Experimental Environment

We ran VPG in the network shown in Figure 6. The network consists of three subnets, and machines are equipped with several operating systems (Solaris, Linux, and IRIX) and CPUs (SPARC, x86, PowerPC, and MIPS).

We ran 160 daemons on approximately 100 nodes. In this experiment, daemons constructed a spanning tree whose diameter was 5. The topology of a spanning tree depends on UIDs.

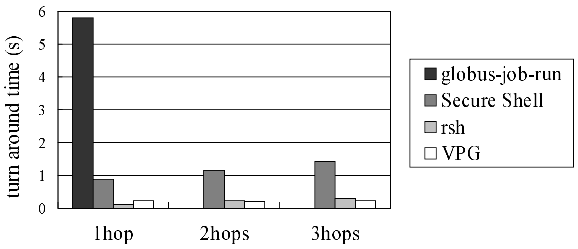

Figure 7. Comparison to other remote job submission tools

5.2. Comparison to Other Job Submission Tools

We compare VPG with three other job submission tools (rsh, SSH, and globus-job-run[1]). Rsh uses Rhost authentication, SSH public key authentication (1024 bit RSA), and globus-job-run X.509 authentication (1024 bit RSA).

We measure the turn around time by submitting a small job. This time is almost equal to the overhead of a remote job submission itself. In addition, we measure the time of the job submission to hosts several hops away from the home host with rsh, SSH, and VPG. Rsh and SSH submit a job to a destination host by relaying a job submission itself several times (e.g., **`ssh hostA 'ssh hostB command'`**).

Figure 7 illustrates the result of this experiment. The overhead of VPG is less than that of SSH and globus-job-run. In addition, the overhead of job submissions except VPG increases in keeping with the number of relays. The main overhead of SSH and globus-job-run is authentication. Because globus-job-run and SSH perform involved authentication using a public key, their overhead is larger than that of rsh and VPG.

VPG forms a tree by keeping connections permanently. Because jobs and their input/output are relayed through connections which have already been created, it does not require creation of new connections (i.e., authentication) when submitting jobs. Therefore, VPG can submit jobs through secure connections without a large overhead.

5.3. Low Level Interface

A communication mechanism of VPG is mainly composed of a redirection and pipe (i.e., unidirectional communication through standard input/output). Thus, whereas simple programs can take advantage of VPG infrastructure, many network programs cannot. For instance, VPG cannot utilize client-server programs which require bi-directional connections.

[1]globus-job-run is a remote job submission tool provided by Globus[14]

Therefore, we have derived a communication library out of VPG implementation. This library is similar to SOCKS [4], and low level programs that use sockets can take advantage of VPG infrastructure with the library. For example, the library enable parallel programs to communicate with each other transparently without considering administrative restrictions.

The library relays communications by using VPG daemons as proxies. The following indicates how process A and B communicates with the library.

1. When process A intends to connect to process B on a remote host, it calls **`vpg_connect()`** instead of **`connect()`** by specifying the host where B is running with a nickname. Then, the connect request is sent to daemon $\tilde{A}$ running on the same host as A. $\tilde{A}$ accepts the request from A, and a connection is established between them.

2. $\tilde{A}$ tries to relay a communication to B. But it does not know a location of B. Therefore, $\tilde{A}$ asks the shell to a route from A to B, and relays the request through a path on a tree constructed by VPG.

3. The request from $\tilde{A}$ is sent to daemon $\tilde{B}$ running on the same host as B. When accepting the request, $\tilde{B}$ sends a connect request to B by calling **`connect()`**. When B accepts the request, the connection is established between B and $\tilde{B}$.

4. Hereafter, process A and B can communicate with each other by relaying messages via VPG daemons.

We compare the overhead of this library with the original UNIX socket library by measuring the execution time of distributed ray-tracing program on 4, 8, 16, and 32 nodes. The result of the experiment shows that the overhead of the VPG library is very small (less than 1%). We can conclude that the library takes advantage of VPG with a small overhead.

6. Related Work

6.1. Resource Management

Many remote job submission tools on clusters or on Grid environments have been developed (e.g., Condor[11], Nimrod[9]). Most of them have been focusing on scheduling. Globus Meta-computing Toolkit[14] provides basic infrastructure for global computing environment.

To the author's knowledge, none of them provide efficient methods to work around restrictions imposed by administrators. The original Globus is blocked by typical firewall configurations and cannot submit jobs from outside a firewall to inside. In addition, because globus-job-run needs to specify a host with its host-name or IP address, it is difficult to submit jobs to a host that has no unique and consistent IP address. In Condor, jobs are automatically submitted to a machine which satisfies the requirements of jobs. Its implementation requires that the host which a user logs in can initiate a direct connection to the destination host.

JXTA[2] is a set of generalized peer-to-peer protocols, and JXTA shell[6] is a command-line interpreter which interacts with the JXTA core services. In the project, a method to bypass firewalls or NAT seems ongoing [5].

SSH provides secure access to remote machines. As the number of machines increases, it becomes difficult for human to utilize many machines easily and efficiently with SSH. If a user establishes and keeps a connection per-host permanently, he/she must maintain many shell windows. If a user initiates connections every time he/she submits a job, it incurs a large overhead as we have mentioned in Section 5.2.

6.2. NAT & Firewall

There are several mechanisms to work around NAT and firewalls.

SOCKS[4] is closest to our work, but has a limited functionality. It is a networking proxy protocol and provides general framework to bypass a firewall transparently and securely. In typical scenarios, hosts inside NAT/firewalls connect to a SOCKS server and hosts outside reach these hosts through the server.

There are two main differences between SOCKS and VPG. First, SOCKS does not have nicknames, so naming DHCP clients remains as an issue, and ensuring the uniqueness of local IPs (in different subnets) is up to the user. Second, more importantly, forwarding connections through multiple SOCK servers is supported but must be configured manually. Therefore it will be difficult to manage hundreds of machines across many (e.g., > 5) subnets.

VPG provides a unique naming scheme for DHCP clients or private hosts. In addition, the tree construction and routing algorithm minimize the need for manual configurations.

To be fair, SOCKS is implemented as a general communication library, whereas VPG only exposes shell functions, that are, remote job submissions, redirections, and pipes. We, however, have derived a similar communication library out of VPG implementation as described in Section 5.3.

RMF (Resource Manager beyond Firewall)[13] is a modification of Globus Resource Allocation Manager (GRAM), and utilizes resources inside firewalls. For example, RMF supports a job submission from outside a firewall to inside, whereas the original Globus does not.

RMF implements this function by using a proxy which relays TCP communications beyond a firewall. It is similar to SOCKS and thus incurs the same problems as SOCKS does. RMF requires cumbersome manual configurations to manage machines across multiple subnets and has no naming scheme for private IPs and DHCP clients.

Virtual Private Network (VPN)[12] is a mechanism to connect multiple private networks through a public network. For instance, subnets over geographically distributed places share their file system through the Internet. Because packets are relayed through the public network, VPN requires establishing secure connections by using authentication, packet tunneling, etc.

These technologies usually require changing administrative restrictions. Therefore, VPG mainly differs from VPN in that VPG constructs a private network at the user level and places major emphasis on a remote job submission.

7. Summary and Future Work

In this paper, we have described *Virtual Private Grid (VPG)* that can easily and securely utilize a large number of machines distributed over multiple administrative domains. It constructs a self-stabilizing spanning tree on the network, and a user can utilize remote machines reachable from the home host through a path on the tree.

We ran VPG daemons on approximately 100 nodes (270 CPUs) and compare VPG with other remote job submission tools. The results of the experiments show that the overhead of VPG is less than that of SSH and Globus. The latest implementation of VPG is available at **`http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/vpg`**

Our future work is to provide easier and more efficient utilization of remote computational resources.

Simplification of daemon configuration As we have described in Section 3.2, the current design requires a user to write configuration files. It consists of a daemon's nickname, a list of connections which the daemon can initiate, etc.

We are planning to minimize the amount of required information. One simple scheme is to omit a list of connections from configurations and to assume that all the hosts can communicate with each other. Because daemons can construct a tree in spite of false configurations, this scheme seems useful.

However, because daemons try to connect to all the other hosts, it takes considerable time to detect a neighbor node with the highest priority and to construct a tree. We are planning to modify the tree construction algorithm to solve the above problem.

Automatic resource selection With hundreds of machines, the user wants jobs and data to be distributed over remote hosts automatically without explicit annotations. We are planning to design a simple task placement algorithm that takes the location of input/output files, communication through pipes, and machine architecture into account.

References

[1] Portable Batch System. *http://pbs.mrj.com/*.

[2] Project JXTA. *http://www.jxta.org/*.

[3] Secure Shell. *http://www.ssh.com/*.

[4] SOCKS Version 5 Protocol. *http://www.socks.nec.com/rfc/rfc1928.txt*.

[5] Project JXTA: An Open, Innovative Collaboration. *http://www.jxta.org/project/www/docs/*, 2001.

[6] Project JXTA: Technical Shell Overview. *http://www.jxta.org/project/www/docs/*, 2001.

[7] Y. Afek, S. Kutten, and M. Yung. Memory Efficient Self Stabilizing Protocols for General Network. *4th Workshop on Distributed Algorithms*, 1990.

[8] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithms. *13th Conferences on Foundations of Software Technology and Theoretical Computer Science*, 1993.

[9] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. *4th International Conference on High Performance Computing in Asia-Pacific Region*, 2000.

[10] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan kaufmann Publishers, 1998.

[11] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *10th IEEE Symposium on High Performance Distributed Computing*, 2001.

[12] C. Scott, P. Wolfe, and M. Erwin. *Virtual Private Networks, 2nd Edition*. O'Reilly, December 1998.

[13] Y. Tanaka, M. Sato, M. Hirano, H. Nakada, and S. Sekiguchi. Resource manager for globus-based wide-area cluster computing. *1st IEEE International Workshop on Cluster Computing*, pages 237–244, 1999.

[14] The Globus Project. *http://www.globus.org/*.