A Scalable High-performance Communication Library for Wide-area Environments

Hideo Saito, Ken Hironaka and Kenjiro Taura
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo, Japan
{h_saito, kenny, tau}@logos.ic.i.u-tokyo.ac.jp

Abstract

We report our progress on SSOCK, a scalable high-performance communication library for wide-area environments. SSOCK has an API similar to that of the Socket library, but solves the connectivity and scalability issues involved with WANs. In one experiment, SSOCK was able to connect 1,262 processes with each other in a 13-cluster environment with firewalls and NAT, without any of the connectivity and resource allocation problems that were encountered when the Socket library was used. In another experiment in which 100 processes simultaneously tried to establish connections, SSOCK was able to establish connections between all pairs of processes in 1.2 seconds, while the Socket library suffered from a large number of packet losses and timed out after 189 seconds.

1. Introduction

With the recent increase in the bandwidth of Wide Area Networks (WANs), it is becoming increasingly common to connect clusters located at different sites in order to perform parallel computation. However, connecting clusters is not a trivial task, because connectivity is often limited by firewalls and nodes without global IP addresses.

In the past, several methods have been proposed to traverse firewalls and NAT [1, 3, 6, 7]. While these methods solve the connectivity problem, many parallel applications also have rigorous scalability and performance requirements as described below:

Scalability In order to scale to a large number of nodes, simplistic schemes that establish a large number of connections need to be avoided. Wide-area connections especially consume a lot of resources, causing various resource allocation problems. For example, the number of sessions that a NAT gateway can handle is limited to about 65,000 (the number of ports), and the number of sessions that a stateful firewall can handle is also limited (Table 1 lists the number of concurrent sessions that some common firewalls can handle). These limitations apply to communication using UDP as well as that using TCP, because NAT gateways and stateful firewalls remember states for both protocols.

Performance Establishing a large number of wide-area connections can also hurt performance. For example, using a large number of wide-area connections in an uncoordinated fashion can result in low communication performance due to packet losses. The number of connections can be reduced by using an overlay network, but this will also result in low communication performance unless the overlay network is constructed carefully.

Therefore, we are developing Scalable Sockets (SSOCK), a communication library that aims not only to solve the connectivity issues involved with WANs but also to achieve high scalability and performance. We have developed a prototype implementation, and performed some preliminary experiments using a real wide-area

Table 1. Number of concurrent sessions for some common firewalls (throughput is also given as a measure of the scale of the firewall).

<table>
<thead>
<tr>
<th>Firewall</th>
<th>Concurrent Sessions</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firebox®Edge X55e</td>
<td>10,000</td>
<td>100 Mbps</td>
</tr>
<tr>
<td>NetScreen-208</td>
<td>128,000</td>
<td>375 Mbps</td>
</tr>
<tr>
<td>CR1000i</td>
<td>400,000</td>
<td>1 Gbps</td>
</tr>
<tr>
<td>FortiGate-3600</td>
<td>1,000,000</td>
<td>4 Gbps</td>
</tr>
</tbody>
</table>
environment that consisted of 13 clusters and had a total of 506 nodes (1,264 cores).

The rest of this paper is organized as follows. We discuss related work in Section 2. We then describe the design and implementation of SSOCK in Section 3, and present our preliminary experimental results in Section 4. Finally, we state our concluding remarks and future work in Section 5.

2. Related work

Some existing methods for traversing firewalls and NAT include UDP hole punching [8] and TCP splicing [2, 5]. Unfortunately, these methods only work in some common situations. For example, they do not work when filtering rules are complicated or when NAT is used inside a network already using NAT.

The use of proxy servers is another method for traversing firewalls, and is the method used by SOCKS [6]. Proxy servers solve connectivity problems by performing communication in place of other nodes that cannot communicate directly with each other. However, some communication may need to go through multiple proxy servers, and manually configuring routes so that a large number of nodes may communicate all-to-all can become quite a difficult task. Moreover, connecting a large number of nodes using proxy servers will result in a large number of wide-area connections.

SmartSockets [7] is a Java library that allows programmers to use sockets in wide-area environments without worrying about connectivity issues. SmartSockets provides extended versions of the Socket and ServerSocket classes that transparently connect normally unconnectable nodes using techniques such as reverse connects and forwarding daemons. Unlike SOCKS, SmartSockets involves almost no configuration because the forwarding daemons automatically construct routing tables. Yet just like SOCKS, SmartSockets may potentially establish a large number of wide-area connections.

One way of connecting a large number of nodes in a scalable fashion is to use Distributed Hash Tables (DHTs), such as Chord [11] and Pastry [9]. In overlay networks that use DHTs, $n$ nodes can communicate with each other with an average of $O(\log n)$ hops if each node maintains information about just $O(\log n)$ other nodes. However, these overlay networks make routing decisions based on addresses assigned in the P2P address space, so messages often end up taking long detours in the physical network.

Ganguly et al. have proposed IP over P2P (IPOP) [3], which performs IP tunneling using Brunet [1], a P2P overlay that works in environments with firewalls and NAT. Ganguly et al. have also proposed to establish shortcut connections between nodes that communicate frequently [4], because messages may take long detours with Brunet just as with Chord or Pastry. However, this method will result in a large number of shortcut connections for applications in which many node pairs communicate frequently. Moreover, the virtual interface that IPOP uses to perform IP tunneling has an overhead that is unacceptable for many parallel applications; while the latency within a cluster is several to several tens of microseconds, the overhead incurred by the virtual interface is several milliseconds.

3. Design and implementation

3.1. Overview

Scalable Sockets (SSOCK) is a scalable high-performance communication library for wide-area environments. It has an API that is similar to that of the Socket library, but provides the following additional features:

- Allows any node to connect to any other node.
- Allows all nodes to connect to each other without reaching various resource limits of the system.
- Has comparable point-to-point communication performance to the Socket library.
- Behaves better than the Socket library when many processes communicate simultaneously.

For the following reasons, SSOCK performs communication within a LAN directly and communication between LANs via forwarding daemons:
• Nodes in the same LAN can almost always communicate directly. Nodes in different LANs sometimes cannot communicate directly, but they can often communicate indirectly with the help of forwarding daemons.

• Forwarding messages within a LAN has a large effect on latency. For example, if we suppose that the latency between all nodes inside a LAN are the same, forwarding a message doubles the latency even if forwarding overhead is ignored.

• Establishing a large number of connections between LANs consumes a large amount of resources. Resource consumption can be greatly decreased by limiting the connections between LANs to those between a small number of forwarding daemons.

Whether communication is performed directly or indirectly via forwarding daemons (ssockds) is determined transparently to the application by the communication library (libsock). As shown in Figure 1, libsock establishes connections within LANs as real connections and connections between LANs as virtual connections. This design is similar to that of SmartSockets, but there are two important differences:

• Connections between LANs are routed through ssockds even when direct communication is possible. This is crucial for achieving high scalability and improving the behavior of when many processes communicate simultaneously.

• Multiple ssockds can be brought up in each LAN. We have not evaluated the performance of using multiple ssockds yet, but we believe that this is an important feature when a single ssockd cannot saturate the bandwidth between LANs and when load balancing is necessary.

We describe libsock and ssockd in more detail in Subsections 3.2 and 3.3. We also describe bootserv, the server used to bootstrap libsock and ssockd, in Subsection 3.4.

### 3.2. Libsock

Libsock is the library component of SSOCK, and is linked to applications that use SSOCK. Ultimately, libsock will support the primary parts of the Socket API, but currently it has an API that is similar but not quite identical to that of the Socket API; it provides functions such as `ss_connect` and `ss_send` that have the same arguments and return values as `connect` and `send`.

To the application, libsock behaves as if it directly connects endpoints, but internally it connects endpoints in different LANs via the overlay network constructed by ssockds. Libsock accesses the ssockd overlay via a ssockd in the same LAN; upon the first invocation of a libsock function, libsock queries bootserv for the ssockd to which it should connect, and establishes a connection with that ssockd.

When `ss_connect` is invoked with an endpoint belonging to the same LAN, libsock establishes a real connection with the remote node by simply passing the arguments to the `connect` system call. Meanwhile, when `ss_connect` is invoked with an endpoint belonging to a different LAN, libsock establishes a virtual connection with the remote node by performing a handshake via the ssockd overlay.

Other functions work in a similar manner, simply passing the arguments to a system call for remote endpoints in the same LAN, and using the ssockd overlay for remote endpoints in different LANs. For example, when `ss_send` is passed a descriptor of a real connection, libsock performs the send by simply passing the arguments to the `send` system call. Meanwhile, when `ss_send` is passed a descriptor of a virtual connection, libsock performs the send via the ssockd overlay.

### 3.3. Ssockd

Ssockd is the daemon component of SSOCK, and is responsible for connecting libsocks in different LANs. At least one ssockd must be brought up in each LAN, but more can be brought up in order to improve bandwidth utilization or to perform load balancing.

When an ssockd is brought up, it establishes a connection with bootserv and learns of the endpoints of other ssockds through bootserv. The ssockd then tries to connect to those endpoints, and all of the connections that are successfully established become a part of the ssockd overlay. Even if some ssockds are not globally accessible, it is guaranteed that virtual connections can be established between any pair of nodes as long as the ssockds can construct a connected graph.

Routing is performed by periodically sending link state messages to each other, so there is no need to start all ssockds at the same time; ssockds can be added or removed on demand. However, delivery and arrival order of messages is not guaranteed when the overlay is changing. Thus, ssockds may be added or removed in between application runs, but they should not be added or removed during an application run.

Once the overlay network is stable, an ssockd simply forwards data among libsocks and ssockds. Unlike a proxy server, an ssockd does not create a new connection for every session that goes through it; only one connection is established between each libsock-ssockd pair and between each ssockd pair. This greatly reduces the number of connections...
between LANs, but even the number of connections established between ssockds may become a problem when the number of ssockds become large. Thus, we ultimately plan on applying the method proposed in [10] to establish connections between ssockds selectively yet in a locality-aware manner.

3.4. Bootserv

Bootserv, the bootstrap server, is the means by which libssocks and ssockds learn of each other. One bootserv must be brought up with a globally accessible endpoint, and libssocks and ssockds must be configured with this endpoint (bootserv is the only component of SSOCK that needs to be globally accessible, and its endpoint is the only item that needs to configured). When it accepts a connection from an ssockd, it receives the endpoint of the ssockd and sends back the endpoints of all other ssockds. When it accepts a connection from a libssock, it receives the endpoint of the libssock and sends back the endpoint of an ssockd in the same LAN as the libssock.

4. Experimental results

4.1. Experimental setup

We performed some preliminary experiments using the environment shown in Figure 2. The environment consisted of 13 Linux clusters located in different parts of Japan and had a total of 506 nodes (1,264 cores). The network configuration of each cluster is shown in Table 2. Here, Firewall, Global and NAT denote the following:

### Table 2. Network configuration of each cluster.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Network</th>
<th>Cluster</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chiba</td>
<td>Global</td>
<td>Kototoi</td>
<td>Global</td>
</tr>
<tr>
<td>Hiro</td>
<td>Global</td>
<td>Kyoto</td>
<td>NAT</td>
</tr>
<tr>
<td>Hiro</td>
<td>Global</td>
<td>Kyushu</td>
<td>Global</td>
</tr>
<tr>
<td>Hongo</td>
<td>Global</td>
<td>Mirai</td>
<td>Global</td>
</tr>
<tr>
<td>Imade</td>
<td>NAT</td>
<td>Okubo</td>
<td>Global</td>
</tr>
<tr>
<td>Istbs</td>
<td>Global</td>
<td>Suzuk</td>
<td>Global</td>
</tr>
<tr>
<td>Keio</td>
<td>Global</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kobe</td>
<td>Firewall</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Firewall* Nodes in this configuration had global IP addresses, but incoming connects from Istbs and Kototoi were filtered. All other traffic, including outgoing connects to Kototoi and Istbs, was not filtered.

*Global* Nodes in this configuration had global IP addresses, and traffic was not filtered.

*NAT* Nodes in this configuration only had private IP addresses. A multi-homed gateway performed Network Address Translation for these nodes, but the gateway itself was not used in our experiments.

4.2. Connectivity and scalability

In our first experiment, we demonstrated three things. First, we demonstrated the connectivity problems that the Socket library encounters in wide-area environments. Next, we showed that simplistic schemes that establish a large number of connections do not scale. Finally, we showed that SSOCK is able to solve both connectivity and scalability problems.

We began by bringing up a process on each core and using the Socket library to try to establish connections between all pairs of processes. In order to avoid packet losses, we coordinated the connects so that connections were established one pair at a time (simultaneous connects are discussed in Section 4.3). We encountered many connectivity problems, the simplest of them being that connections could not be established between Imade and Kyoto, because nodes in both clusters were behind NAT gateways.

We continued with our experiment by ignoring Kyoto and using the remaining 12 clusters, which could be connected all-to-all as long as certain connects were performed in the “correct” direction (i.e., Kobe performs outgoing connects to Istbs and Kototoi, and Imade performs outgoing connects to all other clusters). This time, the problem that we encountered was that we reached the limit on the number of file descriptors that each process can use. There were 1,212 cores in the 12-cluster setup, so each process had to...
connect to 1,211 other processes, but the operating system limit on the number of file descriptors was 1,024. We were able to proceed with our experiment by increasing this limit on every node, but this may not always be possible because it requires administrative privileges.

Even after increasing the limit on the number of file descriptors, we encountered another resource allocation problem: the limit on the number of connections that can be handled by a single NAT gateway. Imade’s gateway would need to handle 65,340 connections if the 54 processes in Imade established connections with the other 1,210 processes, but the gateway’s NAT table became full after 53,800 connections were established.

Meanwhile, when we used the SSOCK library with one ssockd per cluster, we were able to establish connections between all 1,212 processes without encountering any of the previously mentioned connectivity or scalability problems. Inside the library, intra-cluster connections were established as real connections, while inter-cluster connections were established as virtual connections forwarded by several ssockds. For example, connections between Kobe and Kototoi were forwarded by two ssockds (Kobe and Kototoi), and connections between Imade and Kyoto were forwarded by three ssockds (Imade, Kobe and Kyoto).

4.3. Simultaneous connects

In our next experiment, we compared the behaviors of the Socket library and SSOCK when a large number of connects were performed simultaneously. As the Socket library could not traverse firewalls or NAT, we only used the 10 clusters that could communicate freely with each other (those whose network configuration is Global in Table 2). We brought up the same number of processes in each cluster, and every process tried to connect to every other process simultaneously using non-blocking connects.

Figure 3 shows the results for 10 to 100 processes (1 to 10 processes per cluster). With the Socket library, it took 43 seconds to completely connect 60 processes, and the operation failed for 70 or more processes because some of the connections did not complete within 189 seconds and timed out. With SSOCK, the operation only took 1.2 seconds even with 100 processes.

The Socket library performed so poorly, because many SYN packets were dropped by routers; routers often prevent a large number of SYN packets from passing through them for security reasons. As a result, the number of connects that are performed simultaneously needs to be controlled in order to avoid packet losses. This slows down the connection establishment phase, and it also makes programming difficult.

Meanwhile, SSOCK did not suffer from packet losses because connects between nodes in different LANs were realized by normal data (not SYN packets) transmitted through pre-established connections. Therefore, unlike the Socket library and other simplistic schemes that establish a large number of wide-area connections, SSOCK does not require connects to be “paced.”

4.4. Point-to-point performance

In our final experiment, we performed the ping-pong test to show that SSOCK has comparable point-to-point communication performance to the Socket library.

---

1 In Linux, the `connect` system call times out after 189 seconds.
Figure 4 shows the results for when both ping-pong processes were executed in the same cluster (Kototoi). As SSOCK established a real connection and performed no forwarding, the Socket library and SSOCK had almost identical performances.

Figure 5 shows the results for when the two ping-pong processes were executed in different clusters (Hongo and Okubo). Even though SSOCK established a virtual connection that was forwarded by two ssockds, SSOCK still performed just as well as the Socket library.

5. Conclusion

In this paper, we reported our progress on SSOCK, a scalable high-performance communication library for wide-area environments. Using a total of 506 nodes (1,264 cores) in 13 clusters, we showed that SSOCK solves the connectivity and scalability issues involved with WANs and that it also performs well.

Future work includes the following:

- Improved compatibility with the Socket API so that existing wide-area-unaware applications and middleware can be executed with little or no modification.

- A study on the effects of bringing up multiple ssockds in each LAN and ultimately a method to automatically determine the number of ssockds that should be brought up.

Acknowledgements

This research was partially supported by “New IT Infrastructure for the Information-explosion Era” of the MEXT Grant-in-Aid for Scientific Research on Priority Areas.

References