Single IP Address Cluster for Internet Servers

Hiroya Matsuba[1] and Yutaka Ishikawa[1,2]

[1]The University of Tokyo
Information Technology Center

[2]The University of Tokyo
Graduate School of Information
Science and Technology

Abstract

Operating a cluster on a single IP address is required when the cluster is used to provide certain Internet services. This paper proposes SAPS, a new method to assign a single IP address to a cluster. The TCP/IP protocol is handled at a single node called the I/O server. The other nodes, called application nodes, provide the socket interface to applications. The I/O server and applications nodes are connected using a cluster-dedicated network, such as the Myrinet network. The key benefit of the proposed method is that the TCP/IP protocol does not care about congestion and packet loss in the cluster, which often happens if multiple nodes send packets to the bottleneck router. Instead, the cluster-dedicated network manages the packet congestion more efficiently than the TCP/IP protocol. The result of the bandwidth benchmark shows SAPS fully utilizes the bandwidth of the Gigabit Ethernet. The result of the SPEC Web benchmark shows SAPS handles 7.9% more requests than the existing method.

1. Introduction

In recent years, Web or FTP services are often provided using clusters if the number of users is so large that a single computer cannot handle all the requests. Such systems provide a single IP address from the user's point of view, while the server consists of physically separated computers.

The most traditional way to realize the single IP address is the Round-Robin DNS method[6, 8, 3]. In this method, a Round-Robin DNS server maps a single host name to multiple IP addresses whose hosts handle services. When the DNS server receives a query from a client, it selects one of the IP addresses and returns the address to the client. By selecting the IP address in the round-robin manner, requests are distributed to the servers that provide the actual services. This method leads to load imbalances because clients may cache the IP address, and they then send requests to a specific server.

Another way to provide a single IP address is called the Virtual Server method[20, 13]. This system consists of a load balancer and backend servers. The load balancer receives all the data from clients and forwards it to one of the backend servers. By distributing the requests to multiple backend servers, the load of the servers are balanced. Unlike the Round-Robin DNS method, Virtual Server balances the requests, but in turn, it has an issue with the network performance. Usually, all the cluster nodes share a single Internet connection and the router is the bandwidth bottleneck. When multiple backend servers send data back to the clients, packets congest the router. The traffic pattern from the Internet servers is so irregular that the congestion avoidance mechanism of the TCP protocol cannot avoid losing packets[5]. The resulting issue is that a TCP connection stays in the slow-start mode, or burst traffic causes heavy packet loss and re-transmission of many packets[18]. Both cases lead to limited network bandwidth. This performance issue caused by the nature of the TCP protocol is a major drawback of Virtual Server.

This paper proposes a new method that assigns a single IP address to a cluster, while overcoming the network performance issue found in a cluster using the TCP/IP protocol. The proposed method is called SAPS (Single Address Protocol Stack). A cluster with SAPS consists of two kinds of nodes: an I/O server and the applications nodes. The I/O server has the TCP/IP protocol stack so that it handles all TCP packets. An applications node provides the normal TCP socket interfaces which use the TCP/IP protocol stack at the I/O server. The I/O server and the applications nodes communicate with each other using a reliable high-performance communication protocol called PM[16] in the Myrinet network.

SAPS overcomes the network performance issue with the TCP protocol by not using the TCP protocol inside a

1-4244-0910-1/07/$20.00

cluster. Because the communication between the I/O server and the applications nodes is handled by the PM high performance library, the TCP protocol does not encounter the packet loss caused at the bottleneck router. Although PM may encounter the congestion instead of the TCP protocol, PM is more tolerant to heavy traffic and congestion in a cluster than the TCP protocol[16]. Thus, all the data from the applications nodes is sent to the I/O server with sufficient bandwidth, which overcomes the network performance issue with the TCP/IP protocol in a cluster.

In order to evaluate the performance of SAPS, the network bandwidth and the web server performance benchmarks are used. The result of the bandwidth benchmark shows SAPS fully utilizes the bandwidth of the Gigabit Ethernet. The result of the web server benchmark shows SAPS handles 6.6 to 7.9% more requests than Virtual Server.

2. Design

The design goals of SAPS as summarized here provide the following three features:

- The SAPS cluster has a single IP address to provide Internet services.
- SAPS does not use the TCP/IP protocol for communication among the cluster nodes in order to avoid the network performance issue of the TCP protocol in a cluster.
- SAPS provides an application programming interface that is compatible with the existing TCP/IP socket.

To achieve these goals, SAPS is designed as shown in Figure 1.

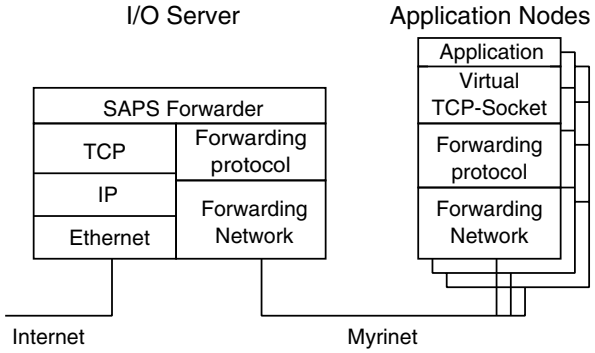


Figure 1. Design of SAPS

SAPS consists of two types of cluster nodes: an I/O server and applications nodes. The I/O server is the only node that has an Internet connection. It has an IP address known to clients. This address is the single IP address assigned to the cluster. The main role of the I/O server is to perform TCP/IP protocol handling. When it accepts a new connection from a client, it selects one of the applications nodes that handles this connection. Incoming TCP packets are handled at the I/O server and the stream data is re-assembled. Then it is sent to the applications node using the internal cluster network. As for outgoing data, the I/O server makes TCP packets when it receives data sent by the applications node.

An applications node provides the socket interface to applications. This socket is called a *Virtual TCP-Socket*. An application such as a web server program runs using this socket interface without any modification to the program.

The high-speed network, in this case, the Myrinet network, is used to connect the I/O server and applications nodes. We will call this network the *Forwarding Network*. Using this network, the *Forwarding Protocol* provides a connection-oriented communication mechanism between the I/O server and applications nodes. The *Forwarding Network* and the *Forwarding Protocol* provide a congestion tolerant communication method inside a cluster. Although the Myrinet network is currently used, SAPS is designed to be capable of using other cluster interconnects, such as Infiniband[4], as the *Forwarding Network*.

We describe the overall behaviour of SAPS. In the rest of this paper, we call the connection provided by the *Forwarding Protocol* the "SAPS connection." If we say simply "connection," this means a connection on the TCP/IP protocol.

- **Socket creation**

When an application running on an applications node issues the socket() system call, a *Virtual TCP-Socket* is created. The *Forwarding Protocol* on the applications node makes a request to the I/O server to make a new SAPS connection. The I/O server accepts the SAPS connection and creates the TCP socket associated with the new SAPS connection.

- **Initiation of an active connection**

When an application issues the connect() system call, the request is forwarded to the I/O server using the SAPS connection. The I/O server establishes the TCP connection between the I/O server and the peer of the TCP connection.

- **Acceptance of a new connection**

The application issues bind() and listen() system calls. These requests are forwarded to the I/O server using the SAPS connection. The application issues the accept() system call and waits for a new connection. This system call does not make a request to the I/O server, but waits for a new SAPS connection from the I/O server. When a new TCP connection is requested to the I/O server by a client,

the I/O server establishes the TCP connection. Then the I/O server makes a new SAPS connection with the applications node. The applications node accepts the SAPS connection and makes a new *Virtual TCP-Socket*. The applications node returns the new *Virtual TCP-Socket* to the user program as a return value of the accept() system call.

On normal TCP/IP, it is prohibited for more than two programs to listen on a single TCP port. On the other hand, SAPS allows this multiple listening if these programs are running on different nodes. In this case, the I/O server selects one of the listing processes that accepts the new connection in a round-robin manner.

- **Transmission of messages**

An application issues the send() or write() system call. The data is transmitted to the I/O server using the SAPS connection. The I/O server makes the TCP packets and sends them to the other end.

- **Reception of messages**

When the I/O server receives data on the TCP connection, the data is transmitted to the applications node using the SAPS connection. The applications node receives the data and stores it in the receive buffer. The data is copied to the application on the recv() or read() system calls.

2.1. Design Details

2.1.1 Forwarding Protocol

The design of the *Forwarding Protocol* is shown in Figure 2.

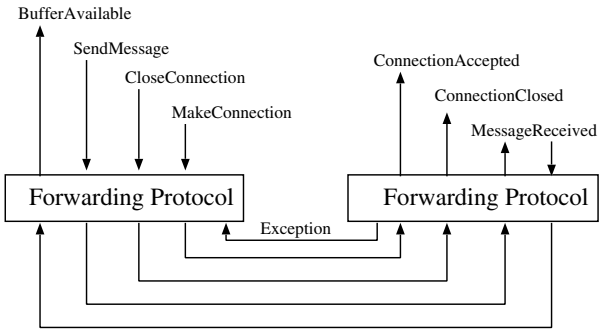


Figure 2. Forwarding Protocol of SAPS

The *Forwarding Protocol* provides the following three functions.

- **MakeConnection()**

This function makes a new SAPS connection.

- **CloseConnection(connection)**

This function closes the connection.

- **SendMessage(connection, message)**

This function sends the message to the peer of the SAPS connection.

When these functions are called, the appropriate message is sent to the other end. On the receiver side, the following callback functions are defined.

- **ConnectionAccepted()**

This function is called by the *Forwarding Protocol* when a new SAPS connection request is received. If this request succeeds, an ack message is sent back to the sender. Otherwise, an exception message is sent back to the sender.

- **ConnectionClosed()**

This function is called by the *Forwarding Protocol* when the SAPS connection is closed by a peer. No reply or ack message is sent back to the sender because it is assumed that the action always succeeds.

- **MessageReceived()**

This function is called by the *Forwarding Protocol* when a message is received. The amount of empty buffer space is sent back to the sender.

Another callback function is defined on the sender side.

- **BufferAvailable()**

A message is sent from the receiver when enough buffer space is available on the receiver side. This function is called when the message arrives at the sender side.

The *Forwarding Protocol* is implemented in a suitable way for the underlying *Forwarding Network*. A high-speed network such as the Myrinet network or Infiniband is supposed as the *Forwarding Network*. The interface between the *Forwarding Network* and the *Forwarding Protocol* is not defined in SAPS. Although it is possible to use a generic interface such as kDAPL[7], it will perform better if it is implemented using a hardware specific interface.

2.1.2 I/O Server

The I/O server performs the protocol conversion between the *Forwarding Protocol* and the TCP/IP protocol. The easiest way to realize the protocol conversion is to use a user process that receives data from the TCP/IP socket and sends it to the applications node, or vice versa. However, in this design, the transmitted data is copied twice: from one socket to the user process, and from the user process to another socket. It is obvious that this user process becomes a bottleneck.

In order to avoid copying data in the I/O server, we have designed it so that the protocol conversion is performed in the operating system kernel. As shown in Figure 1, this protocol converter is called the *SAPS Forwarder*. The *SAPS Forwarder* forwards the data transmitted on TCP connections, handles the system call requests from the applications nodes, and notifies the events on the TCP connection to the application node associated with the connection.

The *SAPS Forwarder* uses the interfaces of the *Forwarding Protocol* as follows:

- **MakeConnection**

When a new TCP connection is accepted, the *SAPS Forwarder* calls this function to make a new SAPS connection that corresponds to the accepted TCP connection.

- **CloseConnection**

The *SAPS Forwarder* calls this function when the TCP connection goes to the CLOSE or TIME_WAIT state and the SAPS connection is no longer used.

- **SendMessage**

The *SAPS Forwarder* calls this function to send data or events to the applications node.

- **ConnectionAccepted**

This function is called when the socket() system call is issued on the applications node. *SAPS Forwarder* makes a new TCP socket corresponding to the socket on the applications node.

- **ConnectionClosed**

This function is called when the applications node closes the socket. The *SAPS Forwarder* disconnects the TCP connection and destroys the TCP socket.

- **MessageReceived**

This function is called when the applications node sends a message. There are two types of messages, a system call request to the socket, and the data to be transmitted to the peer of the TCP connection. The *SAPS Forwarder* takes the action appropriate for the TCP socket layer in response to the received message. If the message is a system call request, the *SAPS Forwarder* will send the result of the request back to the applications node.

- **BufferAvailable**

This function is called when new receive buffer space becomes available on the applications node. The *SAPS Forwarder* picks up a packet from the receive queue of the TCP connection and sends it to the applications node.

2.1.3 Applications Node

The *Virtual TCP-Socket* is provided at the applications node. The *Virtual TCP-Socket* exchanges the transmitted data or system call requests with the *SAPS Forwarder*.

The *Virtual TCP-Socket* uses the interfaces of the *Forwarding Protocol* as follows:

- **MakeConnection**

The *Virtual TCP-Socket* calls this function when it is created.

- **CloseConnection**

The *Virtual TCP-Socket* calls this function when the socket is closed by the application.

- **SendMessage**

The *Virtual TCP-Socket* calls this function to send data written by the application or system call requests to the I/O server.

- **ConnectionAccepted**

This function will eventually be called after the I/O server calls MakeConnection. The *Virtual TCP-Socket* makes a new socket that handles the new connection.

- **ConnectionClosed**

This function will eventually be called after the I/O server calls CloseConnection. If this function is called, the *Virtual TCP-Socket* goes into the error state. If the application program calls a system call of the socket in the error state, the error is reported.

- **MessageReceived**

This function is called when a message arrives from the I/O server. If the message is the result of a system call request sent previously, the system call is completed. If the data is received, it is enqueued in the receive buffer.

- **BufferAvailable**

This function is called when new send buffer space becomes available for the TCP connection on the I/O server. When this function is called, the *Virtual TCP-Socket* wakes up the applications that are waiting for the new buffer space on the write() or poll() system call.

2.2. Flow Control on a SAPS Connection

As the *Forwarding Network*, a cluster-dedicated network such as the Myrinet[12] network is used. Because such networks provide reliable communication, SAPS does not need to care about the reliability of the communication. A problem exists with the flow control on SAPS connections. As described above, one SAPS connection is made for one TCP

$socket(\phi : status) \rightarrow request(\phi) \mid wait(status)$
$connect(address, port : status)$
$\quad \rightarrow request(address, port) \mid wait(status)$
$bind(port : status) \rightarrow request(port) \mid wait(status)$
$listen(backlogsize : status)$
$\quad \rightarrow request(backlogsize) \mid wait(status)$
$accept(\phi : peeraddr) \rightarrow \Phi \mid wait(peeraddr)$
$shutdown(how : status)$
$\quad \rightarrow request(how) \mid wait(status)$
$close(\phi : status) \rightarrow request(\phi) \mid wait(status)$
$write(message, len : len)$
$\quad \rightarrow request(message, len) \mid \Phi$
$read(\phi : message, len) \rightarrow \Phi \mid wait(message, len)$
$setsockopt(type, val : status)$
$\quad \rightarrow request(type, val) \mid wait(status)$
$getsockopt(type : val) \rightarrow request(type) \mid wait(val)$

Figure 3. Protocol to Handle System Calls

socket. Because thousands of TCP sockets may be used at the same time, the number of active SAPS connections may reach several thousands. On the other hand, the *Forwarding Network* is usually not designed to handle thousands of independent communication channels at the same time, due to the limited memory space on the network interface card. Thus, it is impossible to map one SAPS connection to one channel on the *Forwarding Network*.

In order to keep many SAPS connections on any kind of *Forwarding Network*, we use only one channel of the *Forwarding Network*. All the data is transmitted on this single channel with a tag that identifies which SAPS connection the data belongs to. This method requires a flow control mechanism that is independent of the one provided by the *Forwarding Network*. Without this flow control mechanism, some SAPS connections may block other SAPS connections. For example, suppose two independent sockets exist on an applications node and both of them are receiving data. If the application of one socket stops receiving while data is being transmitted, its receive queue will soon become full. If the I/O server continues to send data for this socket, the flow control mechanism of the *Forwarding Network* will stop sending all the data, including the data that belongs to another connection. In order to avoid such a situation, we have defined a flow control mechanism which works on each SAPS connection. The algorithm is described in Appendix A.

2.3. System Call Handling

Using the protocol stack described above, SAPS handles the system call requests as shown in Figure 3. In this figure, let us define
$connect(address, port : status) \rightarrow$
$\quad request(address, port) \mid wait(status)$
means "The $connect$ system call takes the arguments $address$ and $port$ and returns $status$. When this system call is issued on the applications node, it sends the request to the I/O server with the parameters $address$ and $port$. Then it waits for the I/O server to reply. The replied value is $status$ and it is the return value of this system call." The symbol ϕ means "no parameter" and Φ means "no message." If some error happens on the I/O server, this error is reported asynchronously to the applications node. This error is returned the next time the application issues any system call. All the system calls perform this error check at the beginning of the system call handler, thus it is not shown in Figure 3.

3. Implementation

SAPS is currently implemented as kernel modules of Linux 2.6.14. This section describes the implementation of SAPS.

3.1. I/O Server

This section describes an efficient way to implement the protocol conversion mechanism between the *Forwarding Network* and the TCP/IP protocol.

In the original Linux kernel, when a network interface card (NIC) receives a TCP/IP packet, the interrupt handler invokes a software interrupt handler. The software interrupt handler picks the packet from the NIC and delivers it to the TCP/IP protocol stack. The TCP/IP protocol stack performs a procedure to guarantee the reliability of the communication, that is, it requests the peer to resend the lost packets, reorder the packets, and so on. Then, it enqueues correctly received packets in the receive queue. In SAPS, the procedure mentioned above is identical. The following steps differ. The next step in the original kernel is that the TCP/IP protocol stack wakes the receiving process. Then, the process picks up the packet in the receive queue and copies the payload of the packet to the buffer of the application. On the other hand, in SAPS, the TCP/IP protocol stack calls the *SAPS Forwarder*. It picks up the first packet in the receive queue and rewrites the header of the packet with the header of the *Forwarding Protocol*. The payload of the packet is preserved. Then, the *SAPS Forwarder* sends the packet off to the *Forwarding Network*. As for the data from applications nodes, the software interrupt handler picks up a packet from the *Forwarding Network* and delivers it to the *SAPS Forwarder*. Then the *SAPS Forwarder* sends it to the TCP/IP protocol stack. The TCP/IP protocol stack rewrites

Table 1. Specifications of cluster nodes and clients

	I/O server	Applications Node	Client Type 1	Client Type 2
CPU	Opteron 248 × 2	Xeon 2.8GHz × 2	Opteron 2.2GHz ×2	Pentium4 3.0GHz
PCI	PCI-X (64bit 133MHz)	PCI-X (64bit 100MHz)	PCI-Express	PCI-X (64bit 66MHz)
Ethernet	Broadcom BCM5703X	Intel Pro/1000 Server	Broadcom BCM5703X	Intel Pro/1000 Server
Myrinet	Myrinet XP	Myrinet XP		

Table 2. Ethnernet Switches

Model	Cisco Catalyst 3750 24-T-S	Dell PowerConnect 2716
Queue	75 for input, 40 for output	unknown
Buffer	12MBytes shared	unknown
Flow Control	Not Supported	Off

the header of the packet with the one of the TCP/IP protocol.

3.2. Applications Node

Because the existing TCP/IP socket interface is implemented assuming that the TCP/IP protocol stack is also implemented on the same node, the existing implementation of the TCP/IP socket cannot be used for SAPS. Thus, a new implementation of the socket interface, *Virtual TCP-Socket*, was made. The *Virtual TCP-Socket* is implemented so that it provides a socket interface which is fully compatible with the existing TCP socket. In order to provide a compatible interface, the *Virtual TCP-Socket* replaces the whole implementation of the socket of the PF_INET protocol family. Thus, SAPS and the original TCP/IP sockets cannot be used at the same time. The processes on the cluster may communicate with each other by connecting to "localhost."

3.3. Forwarding Network

As the *Forwarding Network*, we currently use the Myrinet network. PM/Myrinet[16] is used as the communication library. In order to use PM/Myrinet in SAPS, we made extensions to PM/Myrinet.

Because PM/Myrinet is designed to be used for high-performance applications, PM/Myrinet provides its communication API only to the user mode application. Thus, the kernel mode interfaces must be added to PM/Myrinet. PM/Myrinet realizes the user mode communication by mapping the memory of the network card to the virtual address space of the user application and writing communication requests to this memory space. In order to add a kernel interface, this memory area is also mapped in the kernel space. The kernel level driver writes the communication requests to this memory space.

We also add an interrupt driven receive handler to the original PM/Myrinet library. In order to eliminate the overhead of the interrupts, PM/Myrinet polls the receive buffer while it waits for the arrival of messages. This is not allowed in the kernel because polling may lock the whole kernel. Thus, we modify the firmware of PM/Myrinet to raise an interrupt when it receives a message. This interrupt is handled by the driver and the received packet is passed to the *Forwarding Protocol* handler.

3.4. Forwarding Protocol

The *Forwarding Protocol* is implemented as a new protocol layer in the Linux kernel. This is implemented as an independent kernel module from the *SAPS Forwarder* or the *Virtual TCP-Socket*. This implementation is used both in the I/O server and applications nodes. Currently, the *Forwarding Protocol* is implemented on Myrinet using the kernel interface of the PM/Myrinet library, which is described above.

If another cluster interconnect, such as Infiniband, is used as the *Forwarding Network*, another implementation of the *Forwarding Protocol* must be made using the interface of this *Forwarding Network*.

4. Evaluation

In this section, we evaluate the performance of SAPS. First, the basic network performance is measured. Then, as a benchmark of the real applications, the web server performance is measured using the SPEC Web benchmark[15]. All results are compared with those of Virtual Server via direct routing[13].

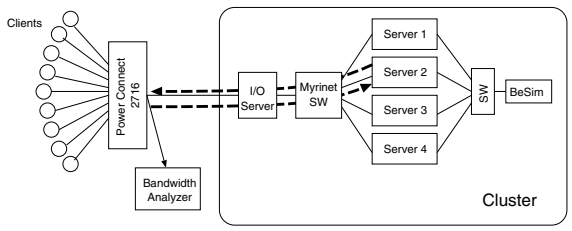


Figure 4. SAPS Network Topology

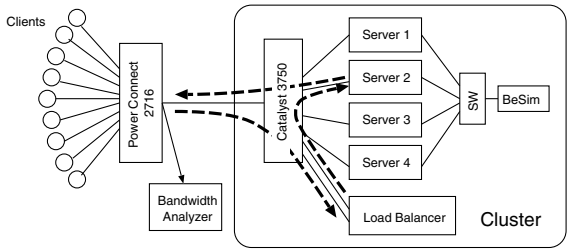


Figure 5. Virtual Server Network Topology

4.1. Evaluation Environment

SAPS is evaluated using a 5-node cluster, one is the I/O server and the others are applications nodes. As client computers that communicate with the cluster, nine computers are used. The specification of the computers and the network switches are shown in Tables 1 and 2, respectively.

The network topologies for SAPS and Virtual Server are shown in Figures 4 and 5. As for Virtual Server, the data transmitted from a client to a server is forwarded by the load balancer while the reply packets from the server are sent directly to the client. In those figures, a special computer marked "BeSim" is installed to run the SPEC Web benchmark. The role of this computer will be shown later. Using the port mirroring facility of the Ethernet switch, all the packets going into the clients are captured by the computer marked "Bandwidth Analyzer." This computer measures the aggregated bandwidth of all the streams from the servers to the clients.

4.2. Basic Performance

First, the point-to-point round trip time is shown. It was measured at both the user and TCP levels. User level one is the time from sending a 4-byte message and receiving its echo at the application level. In order to measure the end-to-end latency in the TCP level, the time elapsed from sending the SYN packet and receiving the SYN-ACK packet was measured at the connection establishment phase. The results are shown in Table 3. The user level round trip time of SAPS is longer than that of Virtual Server, on average. But SAPS performs more stably than Virtual Server. The TCP level round trip time of SAPS is shorter than that of Virtual Server. This is because all packets, including ACK packets, detour in Virtual Server, while ACK packets are handled in the I/O server without forwarding in SAPS.

The point-to-point bandwidth is shown in Figure 6. This is the bandwidth of a single burst stream from the server to the client. SAPS performs better than Virtual Server with short messages. This is because the TCP level round trip time is shorter in SAPS, and thus the TCP congestion window grows faster than Virtual Server. Both SAPS and Virtual Server utilize the available bandwidth of the Gigabit Ethernet at their peak bandwidth.

Table 3. Round Trip Time (μs)

	User		
	Min.	Max.	Avg.
SAPS	210	217	213
Virtual Server	162	236	173
	TCP		
	Min.	Max.	Avg.
SAPS	92	106	96.6
Virtual Server	105	187	134.8

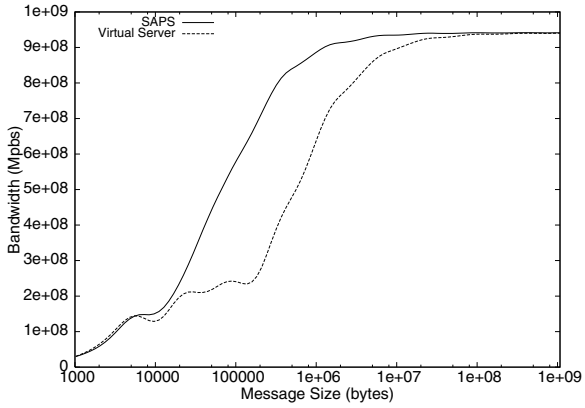
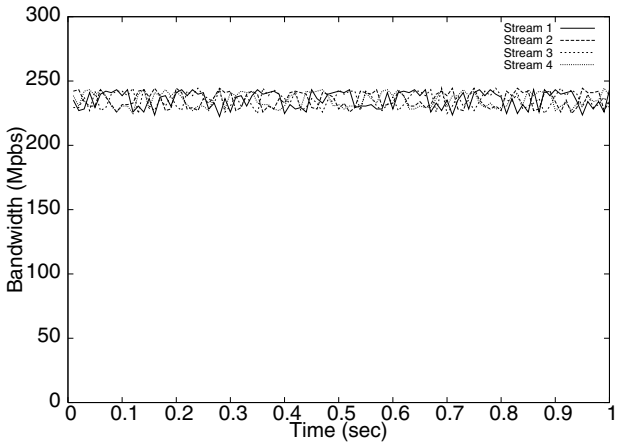
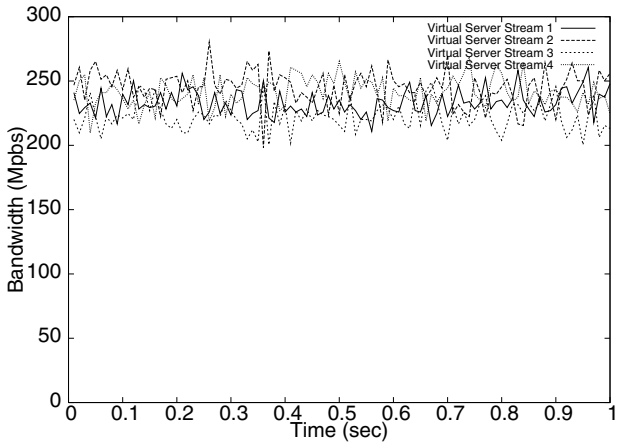


Figure 6. Point-to-Point Bandwidth

Figure 7 shows the bandwidth of four burst streams. These streams flow from four servers to four clients. The bandwidth is measured every 10 milliseconds at the "Bandwidth Analyzer." This result shows the SAPS behaviour is more stable than that of Virtual Server.



SAPS



Virtual Server

Figure 7. Bandwidth of Four Streams

4.3. Web Server Performance

As the benchmark of a real application, the performance of the web server was measured using the SPEC Web 2005 benchmark[15]. The SPEC Web 2005 benchmark provides three kinds of benchmarks: *e-Commerce*, *Banking*, and *Support*. The *e-Commerce* and *Banking* benchmarks simulate web sites that provide online shopping and online banking, respectively. In these benchmarks, the web server has files that keep information about users who are currently logged on to the web site. In order to run these benchmarks, these files must be shared among all cluster nodes using a network file system. Because this file sharing becomes a bottleneck for performance, the *e-Commerce* and *Banking* benchmarks do not reflect the network behaviour. The *Support* benchmark simulates a vendor's support web site. Users search for a product, or for available downloads. Some users may download a large file (up to 30 MBytes). This benchmark requires enough network performance to reply to the users' download request. Thus, this benchmark is suitable for comparing the network performance of SAPS with Virtual Server.

The network topology used for the SPEC Web benchmark has already been shown in Figure 4. In Figure 4, a special computer marked "BeSim" is the simulator of a database server required to run the SPEC Web benchmark.

The network topology of Virtual Server has also already been shown in Figure 5. No special software is required to set up a computer as a load balancer. It is implemented in the standard Linux 2.6 kernel. As the web server program, the Apache httpd server version 2.0.54[1] and PHP version 5.1.2 are used. The basic configuration parameter of the Apache httpd server is shown in Table 4.

For a fair comparison between SAPS and Virtual Server, the same buffer size (128kb) is set for all the socket buffers, including the TCP/IP buffer on the I/O server, the *Virtual TCP-Socket* buffer on the applications node, and the TCP socket buffer used for Virtual Server.

Table 4. Apache Configuration

Keep Alive	On
Keep Alive Time	unlimited
Keep Alive Requests	unlimited
# of server process	1500 at startupx
	2500 max

4.3.1 Results

We ran the benchmark for five minutes, excluding the start up and warming up time. Although five-minutes runtime is enough to compare SAPS with Virtual Server, it is too short to be used for formal results of the SPEC Web benchmark. Thus, the results shown in this section are not compared with the ones on the web page of the SPEC benchmark.

Our results are shown in Table 5. *Simultaneous Sessions* means the number of users supposed to be accessing the web site. The benchmark ran three times. As for Virtual Server, the second trial failed because no response was returned for a request within one minute. On the other hand, SAPS worked well three times. The *Requests* field in the table represents the number of requests handled during the benchmark. SAPS handled 6.6% to 7.9% more requests, than Virtual Server.

The benchmark also measures the time required to reply to the users' request. If the response time is shorter than three seconds, the *TimeGood* value is incremented. If the response is returned within five seconds, the request is counted in *TimeTolerable*. Comparing *TimeGood* and *Time-*

Table 5. SPEC Web Results

	Simultanious Sessions	Run	Requests	TimeGood	TimeTolerable	Error
SAPS	2800	1	75569	88.5%	97.6%	0
		2	75777	88.8%	97.7%	0
		3	75307	87.8%	97.7%	0
Virtual Server	2800	1	70227	73.6%	93.0%	13
		2	Failed	-	-	-
		3	70641	74.8%	93.3%	2

Tolerable, SAPS handled 25% to 30% more requests in *TimeGood* and 11% to 13% more requests in *TimeTolerable* than Virtual Server.

The *Error* field shows the number of failed requests. An error is reported when a time out happens. The difference from the second trials of Virtual Server, in which the benchmark aborts, is that partial responses were received for the requests. SAPS handled all the request without errors while Virtual Server failed to reply to some requests. These results show that SAPS performs better than Virtual Server as a network system of Web servers.

Table 6 shows the detailed statistics. These values were taken during the first one minute of each run and the average values are shown. The *Retransmission* column shows the number of retransmitted packets per second. This value is gathered using the SNMP facility of the Linux kernel (i.e. reading /proc/net/snmp). For SAPS, it was taken at the I/O server. For Virtual Server, it was taken at each server, and the aggregated value is shown. The *Bandwidth* column shows the bandwidth of the transmitted data from the servers to the clients. This value was taken at the "Bandwidth Analyzer." This value includes the bandwidth of the retransmitted data. The *CPU usage* column represents the load of the I/O server or the load balancer. This value is taken by reading the /proc/stat file. It is shown that Virtual Server drops many packets and does not utilize the available bandwidth. On the other hand, SAPS retransmits fewer packets. According to the detailed SNMP statistics, the retransmissions of SAPS are caused by the timeout of retransmission timer, which should not happen. We are investigating the reason of this time out. SAPS consumes more CPU resources at the I/O server than Virtual Server. This may affect the scalability of SAPS. However, at least for this benchmark, both SAPS and Virtual Server will not perform better even if more servers were available because the available bandwidth is already consumed by four servers.

5. Related Work

The Kerrighed[10, 11] project makes a single system image cluster for high-performance computing. This system provides many features that make a cluster a single system, such as software distributed shared memory. In order to provide the single IP address, a head node receives all the packets and distributes them to cluster nodes. Because TCP/IP protocol handling is performed on each node, the TCP performance is influenced by packet congestion just as it is with Virtual Server.

OSF/1 AD TNC[19] is a UNIX operating system providing a single system image on multicomputers. SAPS is similar to the protocol stack of this operating system in that the network protocol is handled on a different node from the one where the socket interface is provided. Although OSF/1 AD TNC has similar functions to those of SAPS, it is reported that its base system, OSF/1 AD, does not perform well and cannot utilize the bandwidth of the high-speed interconnect[14].

TCP pacing [9, 17, 2] is an important technique to avoid network congestion. In this method, the bandwidth of the TCP stream is limited in order not to exceed the bandwidth of the bottleneck link. TCP pacing is effective if the available bandwidth is known or precisely estimated. However at the cluster of web servers, it is impossible for each node to estimate the bandwidth that can be consumed by the node because the web traffic is so random. Thus, TCP pacing cannot be applied to avoid the congestion that happens in the cluster of web servers.

6. Conclusion

This paper has proposed SAPS, a new method that enables a cluster to run on a single IP address. In this system, the TCP/IP protocol is handled at the I/O server, and a socket interface is provided at the applications nodes. The I/O server and the applications nodes are connected by a cluster-dedicated network. This design avoids using the TCP/IP protocol inside a cluster, in order to overcome the network performance issue on the existing method. The issue is that the irregular pattern of traffic from servers breaks the congestion control mechanism of the TCP protocol, which causes the loss of many packets at a bottleneck router and results in limited bandwidth.

Table 6. Detailed Statistics

	Retransmission (packets/s)	Bandwidth (Mpbs)	CPU usage (%)
SAPS	103	930.0	45
Virtual Server	11984	917.8	6

We have implemented SAPS using the Linux operating system. In the evaluation using the bandwidth benchmark, it was shown that SAPS utilizes the available bandwidth of Gigabit Ethernet, which means the I/O server does not become the performance bottleneck. The result of the SPEC Web benchmark showed that SAPS has handled 30% more requests in *TimeGood*, 13% more requests in *TimeTolerable*, and 7.9% more requests in total, than Virtual Server. During testing this benchmark, more than five thousand TCP connections were established at the same time. The benchmark result shows not only that SAPS performs well, but also that the implementation is stable.

Acknowledgement

This work has been partially supported by the e-Society project led by Ministry of Education, Culture, Sports, Science and Technology, MEXT, Japan, and JSPS Research Fellowships for Young Scientists.

References

[1] Apache Httpd Server. http://httpd.apache.org/.
[2] M. Aron and P. Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. In *Technical Report TR98-318, Rice University Computer Science*, 1998.
[3] T. Brisco. Dns support for load balancing. *RFC1794*, 1994.
[4] InfiniBand. http://www.infinibandta.org.
[5] V. Jacobson and M. Karels. Congestion avoidance and control. In *ACM Computer Communication Review*, pages 314–329, 1990.
[6] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2):155–164, 1994.
[7] kDAPL. http://www.datcollaborative.org/.
[8] T. T. Kwan, R. McCrath, and D. A. Reed. NCSA's world wide web server: Design and performance. *IEEE Computer*, 28(11):68–74, 1995.
[9] M. Matsuda, T. Kudoh, Y. Kodama, R. Takano, and Y. Ishikawa. TCP Adaptation for MPI on Long-and-Fat Networks. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2005.
[10] C. Morin, P. Gallard, R. Lottiaux, and G. Valle. Towards an efficient single system image cluster operating system. *Future Gener. Comput. Syst.*, 20(4):505–521, 2004.
[11] C. Morin, R. Lottiaux, G. Valle, P. Gallard, D. Margery, J.-Y. Berthou, and I. Scherson. Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.
[12] Myrinet. http://www.myri.com.
[13] P. O'Rourke and M. Keefe. Performance Evaluation of Linux Virtual Server. *LISA 2001 15th Systems Administration Conference*, 2001.
[14] S. Saini and H. D. Simon. Applications performance under OSF/1 AD and SUNMOS on Intel Paragon XP/S-15. In *SC*, pages 580–589, 1994.
[15] SPEC Benchmark. http://www.spec.org/.
[16] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa. PM2: High performance communication middleware for heterogeneous network environments. In *Proceedings of the IEEE/ACM SC2000 Conference*, 2000.
[17] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa. Design and Evaluation of Precise Software Pacing Mechanisms for Fast Long-Distance Networks. In *Proceedings of PFLDnet 2005*.
[18] V. Visweswaraiah and J. Heidemann. Improving restart of idle tcp connections. In *Techinical Report 97-661, University of Southern California*, 1997.
[19] R. Zajcew, P. Roy, D. Black, and C. Peak. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proc. of the Winter 1993 USENIX Conference*, pages 449–468, San Diego, California, 1993.
[20] W. Zhang. Linux Virtual Servers for Scalable Network Services. *Linux Symposium*, 2000.

Flow Control Algorithm

The sender has the following variables:
A_s : The number of octets that have already been sent
L_s : The upper limit that the sender may send
The receiver has the following variables:
A_r : The number of octets that have already been received
B_r : Size of the receive buffer
E_r : Current free space of the receive buffer
L_r : The upper limit that has been reported to the sender
The flow control mechanism works as follows.
1. The receiver initializes L_r with B_r at the beginning of the connection
2. The receiver reports L_r to the sender.
3. The sender updates L_s with the received value L_r.
4. The sender may send while $A_s < L_s$
5. if $A_r + E_r > L_r + B_r/2$, the receiver updates L_r with $A_r + E_r$
6. goto 2.