Abstract

This paper tries to tackle the problem of providing retrofitting network QoS in clustered configurations. For this purpose, we designed a QoS manager which runs on each of the internal cluster nodes and controls network I/O of local interface cooperating with peer managers on other nodes towards a certain QoS policy.

First, we show the design of control framework, contending that an end-host manager-based mechanism is a desirable approach, which utilizes an end-host oriented network control primitive, Netnice.

Second, for flexibility of configuration, we propose object-oriented modeling of the QoS manager with event-handler based configuration mechanism, and show the design of an object-oriented configuration language that allow simple and flexible definition of QoS policies.

Lastly, results from two simple experiments with a Web server cluster are analyzed.

1 Introduction

Due to the recent growth of the Internet traffic, single servers are sometimes no longer sufficient to accommodate the requested load, and a great deal of server clusters have been developed to meet the required service performance. However, it seems that realization of network Quality of Service (QoS) for the clustered configuration has not gained much attention in the research field.

This trend may be attributed to the following reasons. First of all, load-balancing technologies, usually coupled with the clustering techniques, attempts to enhance the service quality through balancing of request loads, which consequently decreases somewhat the focus on the network QoS. Second, note that it is not sufficient to consider only the network traffic to achieve desired QoS, since the behavior of end-host operating system and applications directly influence the QoS. This makes traffic control of server clusters currently the art of system tuning to cater to specific cases, rather than a generalizable communication technology.

However, server clustering technologies make traffic of various types with different traffic profiles share a common link, which directly influences the QoS of more stringent applications. For instance, it is unwise to have continuous media streaming traffic over UDP and bulk data transfer over TCP share a single link without proper traffic control. Therefore, we believe that a network QoS solution is needed for the server clusters in service.

To this end, we propose a retrofitting network QoS management framework, in which QoS managers capable of controlling their own network I/O reside on each cluster node and cooperate toward a certain QoS policy.

The paper is organized as follows. First, in Section 2, we present various approaches in the problem domain, and clarify our approach. In Section 3, we show the configuration mechanism of the QoS manager. In Section 4, a practical experiment with a web server cluster is shown for a proof of concept, followed by a discussion in Section 5. Then, we briefly review some related work in Section 6, and conclude the paper in Section 7.

2 Control Framework

In this section, we first discuss where the control mechanism for the retrofitting network QoS for server cluster should reside, and show that an end-host based approach is well suited for this problem domain. Second, we show a manager-based model is more appropriate than self-regulating servers. Third, we list required properties for the traffic control primitive that works on the end nodes, and
propose to use our end-host oriented traffic control primitive, Netnice, to realize such control. Lastly, we combine the arguments and present the proposed framework.

2.1 Location of Control

We present an end-host oriented QoS manager as a retrofitting solution for the QoS problem. In this design decision, a controversial topic is the location of control. We advocate to do it on the end-host.

First, we can plausibly make an assumption that the cluster nodes are accessible more easily than the intermediate nodes in clustered settings. The routers on the communication path would be out of administrative domain, and even the first hop could be inaccessible, in the sense that internal algorithms are not modifiable. It is obvious that the cluster node is a reasonable point of control for retrofitting QoS.

Second, as the traffic from the cluster nodes grows, traffic at the first hop router increases, resulting in the need for proper resource management. One option to address the problem is the control at the router. However, if the end nodes are capable of controlling their own traffic, it would contribute to better goodput, and thus, would provide more efficient solution.

Lastly, the intermediate nodes have neither enough information about the big picture nor enough details about every flow. For example, a service with heavy file access (e.g., web search engine or a distributed file system) is bound by disk I/O of end-hosts, totally independent of the network traffic. A transaction with high computational complexity (e.g., CODEC of multimedia data) is clearly bound by the node CPU resources. These types of applications make the network traffic quite unpredictable, and limit the effectiveness of approaches that work solely on the intermediate nodes. On the other hand, end-host control would provide efficient solution to the problem, since the nodes can exploit the information kept at the end-hosts (demonstrated later in the paper).

2.2 Agent of Control

Assuming the end-node approach, there are two types of control models that can be used, namely internal control model (individual server programs autonomously adapt themselves to meet a certain policy) and external control model (a QoS manager externally controls target flow). We argue that the manager based approach is better.

First, since program modification is sometimes expensive, or even impossible (e.g., with commercial server products), it is obvious that manager-based solution is advantageous as a retrofitting solution.

Second, the external manager can provide both polling-based and event-driven control, given the proper support of event delivering mechanism. For example, realization of polling-based QoS control in the autonomous approach can be laborious, if the server program is not structured for such control. Or, even with a polling-based approach, the original program might require substantial modifications of the polling interval to match the required control.

On the other hand, we admit that modification of the server program would be beneficial, if the server is event-driven. Simple modification for message passing could reduce overhead for status inspection and contribute to better system performance. However, this ground does not degrade the advantage of the manager based model.

2.3 Primitive of Control

The claim that the control should be made on end-hosts would necessarily require that end-hosts have primitives for traffic control.

For this purpose, we could utilize several existing implementations that work on end-hosts [9, 5]. Basically, they take some matching rules for packet classification, such as address-port pair, and queue configuration parameters for traffic control, such as bandwidth and delay. This is the traditional model for intermediate node based traffic control.

However, we contend that existing primitives are restrictive. First, although the model is well applicable to control of aggregated flows, it is not convenient for per-flow control required for services such as Constant Bit Rate (CBR). Second, the model for intermediate node control cannot make most of the information kept at the end-node, such as status of the sender program.

Therefore, we will use a primitive that works on terminating entities of the communication on the host, such as processes and sockets, since it is preferable for end-host based approach. Consequently, we propose to utilize our Netnice solution [8] that realizes an abstraction of network interfaces, which is hierarchically attachable to terminating entities such as processes and sockets on the host, to realize detailed control of their network I/O.

2.4 Proposed Framework

Combining the arguments introduced above, we propose a new control framework, illustrated in Figure 1. The rounded box (dashed line) denotes a cluster node, that is, the end-host. Each end-host has a QoS manager (the Netnice daemon, or netniced), monitoring application processes (denoted by circles). The communication among managers in different end-hosts is depicted as horizontal arrows. At the hardware boundary, we see a real network interface (shadowed cylinder), which connects four virtual network interfaces, VIFs, denoted as cylinders. The virtual interfaces are attached to network I/O of the processes. The
QoS manager controls traffic flow by VIF operations such as creation, deletion, and parameter setting of the virtual interfaces (depicted as the gray curvy arrow).

A global manager observing the entire system might be more powerful in policy enforcement, but, it would introduce a bottleneck and a single point of failure. Our proposal is a distributed solution, in which the system comprises cooperating managers on all end-hosts. This sacrifices the global point of view, however, there is no single point of failure and would be relatively scalable compared with the centralized solution.

Although the scheme requires modification of underlying operating system, the QoS manager does not change the existing cluster hardware, nor software system. Thus, we say it is a retrofitting solution, applicable to legacy systems.

3 Configuration Mechanism

In this section, we present how to achieve flexibility in the control for various QoS policies. First, we show the control model of the manager. Second, we present a language approach, to address the flexible configuration issue. Third, a simple example, utilizing the mechanism proposed here, is given for illustration.

3.1 Event-Handler Based Control Model

We argued that end-host manager-based control can support event-driven and polling-based control. To realize both approaches in one scheme, we present an event-handler approach, as a control model of the QoS manager.

In this approach, the behavior of the manager is described as a set of event-handling routines, such as `init()`, `timer()`, `update()`, and `term()`.

Event `init` is raised when the manager starts up, and initializes the system. The `timer` event is raised periodically and realizes the polling behavior of the manager. Event `update` occurs when the system receives a message from a peer netniced residing on a neighbor host, and is used to update system parameters of the local system, such as bandwidth allocation. Event `term` is raised to terminate the manager, for graceful termination.

The event-handler approach realizes both the event-driven control and polling control, in a single coherent model. Also, there is enough room for system expansion, by implementing more event-delivering routines in the manager program.

3.2 Configuration language

For description of the event handlers discussed above, we need a flexible configuration mechanism. To realize such scheme, several requirements must be met.

First of all, algorithmic flexibility must be guaranteed. One possibility is a simple parameter-based configuration, often seen in schemes that configure network QoS policies. However, this is not sufficient here. For example, constant bit rate (CBR) service can be easily supported by a simple static configuration, but available bit rate (ABR) service is not well expressed in a set of parameters. In other words, higher degree of freedom to express dynamic behavior of the manager is indispensable, and thus, we need a flexible configuration language, not just a set of parameters.

Second, since the managers deal with various system entities, such as processes, sockets, VIFs, and peer nodes, our proposal is an object-oriented language with a class library support for QoS management. Additionally, for the simplicity of the language, we support special data types such as bandwidth type.

Based on the above discussion, we extended the well-used object-oriented language, Java, and designed a class library for QoS algorithms. A summary of the class library

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>Boolean value class.</td>
</tr>
<tr>
<td>Int</td>
<td>Integer class.</td>
</tr>
<tr>
<td>Process</td>
<td>Abstraction of a process on the system.</td>
</tr>
<tr>
<td>ProcessGroup</td>
<td>A group of processes on the system, useful for iterative operation.</td>
</tr>
<tr>
<td>Socket</td>
<td>Abstraction of a socket.</td>
</tr>
<tr>
<td>SocketGroup</td>
<td>A group of sockets, useful for iterative operation.</td>
</tr>
<tr>
<td>Str</td>
<td>Strings class.</td>
</tr>
<tr>
<td>System</td>
<td>Representation of the manager.</td>
</tr>
<tr>
<td>Tupple</td>
<td>Abstraction for the inter-node communication.</td>
</tr>
<tr>
<td>VIF</td>
<td>Virtual Network Interface.</td>
</tr>
</tbody>
</table>

Table 1. Standard library
/* A Simple Event Handler */
Tupple bandwidth;

init()
{
    Int fairshare;
    VIF root;
    set_if("fxp0");
    root = get_root();
    root.bandwidth = 100Mbps;
    root.select_by_file("httpd").wrap("Priority",
    1,
    16Mbps);
    bandwidth = 16Mbps;
    fairshare = (100Mbps - bandwidth.total())
    / bandwidth.size();
}

Figure 2. A simple event handler.

we developed is shown in Table 1. (For complete description of the library, refer to www.netnice.org.)

3.3 Example

For illustration, in Figure 2 we show a sample event-handler, init(). On lines 7-8, two variables, fairshare and root, are declared. Line 10 informs the system about the real network interface of interest. The method to set the interface, set_if(), is a method of System class. Since init() is another method of System, the method call does not require an object for reference. On line 12, get_root() method is called, and returns a root VIF attached to the real interface, fxp0. On line 13, the flow specification of the virtual interface is set. Note that the script language supports a special bandwidth type, which is not supported by the original language. Lines 15 through 21 demonstrate powerful nature of this approach, as explained below.

Method VIF.select_by_file(Str) returns a class ProcessGroup, representing a set of processes, whose program name match the provided string. Invoking Method ProcessGroup.wrap(Type, Param, Bandwidth) creates a new VIF with a bandwidth associated with it, and attaches all processes in the process group to that VIF. Therefore, the expression on lines 15 - 17 isolates every web server process (httpd) on the system, and wraps it in a newly created virtual interface, with up to 16Mbps bandwidth. The entire operation is depicted in Figure 3.

Lines 19 – 21 calculate the fair share of the node in the cluster, using Tupple. Tupple class is a primitive for distributed programming, providing simple inter-host communication means, as shown in Figure 4.

As shown, Tupple allows a node to write to its own slot in the tupple space. Additionally, it supports calculation methods, such as total(), max(), ave(), min(), and size() so that the distributed algorithm is concisely expressed. The first four methods return the result of the calculations performed on each tupple item. The last, size() returns the number of items in the tupple space, shared by the peer nodes.

In the example, bandwidth is a global variable of type Tupple. Thus, bandwidth.total() at line 20 returns sum of the consumed bandwidth by all the nodes in the cluster, assuming each node properly maintains their statistics, by writing onto the tupple. The whole expression on lines 20 – 21 sets the local fair share of available bandwidth to fairshare (the value is not used in the sample code).

3.4 Discussion

First, note that the language approach can bring about high flexibility in policy description. Although we have not described other language approaches (e.g., a declarative approach in which the QoS policy is described by a set of declarations), we believe that procedure-based descriptions would be more expressive and more natural to programmers.

Second, in our event-handler approach, the complex
behavior of the QoS managers is expressed concisely, thanks to the object-oriented language and the built-in objects. The event-handler framework enables reuse of control framework, substantially reducing the code size. The ProcessGroup class aggregates a set of processes with the same property, and simplifies the policy description. The Tuple class conceals underlying details of the communication between hosts, contributing to the simplicity of algorithm description.

Third, this architecture supports easy extensions of the system functionality. Since every built-in object is realized by the class library, by simply extending the library, functionality of the entire system would easily expand. Likewise, types of the event handlers are extensible, by extending the System class that represents the manager process itself.

Lastly, by extending an existing well-known language, we can harness the rich programming expertise in the community. Further, minor violations of the language standard, as is seen in bandwidth type, contribute to the simplicity of the code, while keeping the language approach friendly to many programmers in the field.

4 Evaluation

We implemented the framework on FreeBSD, and a prototype system, utilizing a small web cluster.

First, we give an introduction of the experiment, with brief justification of the approach. Second, the experimental setting is described. Lastly, results are given to show the effectiveness of the system.

4.1 Introduction

We choose a web server cluster, a common and practical application of clustering technology, for the following reasons.

First of all, the web server cluster well represents the problem of network QoS for server clusters. Whenever the cluster is underperforming, it is easy to increase the cluster performance by adding processing nodes into the cluster. However, the link bandwidth is not easily expandable and if the connection between the cluster and the Internet has bandwidth constraints, network QoS will be needed.

Second, in the cluster of web servers, end-host based approach would provide better control for proper differentiation of service classes. For example, a flow from a CGI script differs greatly in its traffic pattern. In the CPU bound transaction, how/when the requests are served by the clusters is not known at the intermediate nodes. There are possible hints in the request headers (e.g., a URL in the request), but such hints do not necessarily contains sufficient information. Our QoS manager is expected to exhibit its usefulness and effectiveness in such a case.

Third, turning back to the control model issue, the manager-based approach better fits in this situation. The server programs are not structured to communicate with peer nodes for QoS control. Hence, this is a good application to demonstrate our retrofitting approach where no modification of the existing system is required for distributed control.

4.2 Experiment

Setting The experimental setting is given in Figure 5. The cluster comprises four homogeneous nodes running Apache web server [12] and four client machines capable of generating requests for the cluster.

At the router, we employed Dummynet [9], and emulated T3 link bandwidth (45Mbps) for all the in-bound and out-bound traffic passing through.

The hardware specification used in the experiments is as follows. Processor: Intel Celeron processor (733Mhz), Memory: 64MB, Hard-drive: Fujitsu MPG3204AT (20GB), Network interface: 3Com EtherLink XL 905B 10/1000, Motherboard: A-Open MX3S, Chipset: Intel i815E. The machines are connected to a fastether switch, Netgear FS516 at the client side, and Netgear FS508JP at the cluster side. Interfaces of the router and the switches are 100BaseT, as indicated.

QoS Manager In each cluster node, we added our QoS manager, netniced, in a retrofitting manner. We use a FreeBSD kernel with Netnice patch, and simply run the manager with two sample configurations.
The manager monitors the execution mode of the processes and their network usage, and controls virtual network interfaces, exchanging traffic information with peers.

In the control algorithm, we defined three modes of operation for a web server; normal http, CGI, and pseudo-stream. In normal http mode, the process simply transfers files. In CGI mode, the process executes an external program for document generation. In pseudo-stream mode, the process transfers stream data, using TCP.

To classify the processes this way, we added a method `ProcessGroup.VIF.select_by_openfile(Str)` to the VIF class. With this method, the manager inspects the open file table of each process attached to the VIF and selects matching processes. The pseudo-stream mode is easy to discern, since a file extension appears in its accessing file name. The CGI mode is detectable, since `httpd` opens a pipe for CGI execution. We defined the normal mode as the complement of these two modes.

We then configured the QoS manager for two experimental policies, namely, ‘Constant Bit Rate Stream’, and ‘Premium CGI Service’, each detailed below.

**Case 1: Constant Bit Rate Stream**

In this scenario, constant bit rate service is tested. Figure 6 shows conceptual overview of the control. A cylinder denotes a virtual interface, and a circle depicts a process.

In the configuration, there are three basic VIFs; `Root`, `Http`, and `Stream`. `Root VIF` on each host is configured as a traffic shaper, and configured to have the fair share of the T3 uplink. The share is automatically calculated as follows, using a tuple: \( \frac{45\text{Mbps} - 5\text{Mbps}}{2} = 10\text{Mbps} \) each, where the 5Mbps is the reserved amount for issuing requests. `Http` is a VIF for normal files and CGI programs. `Stream` is a VIF for streaming processes. These two VIFs are configured as priority queues, and we gave higher priority to stream VIFs. In addition, we configured the manager to automatically wrap the streaming processes with a dedicated VIF in a traffic shaper mode (4Mbps at max). This way, streaming processes generate constant bit rate flow.

The manager first generates the basic VIFs at startup, and monitors executing mode of each process. When the manager finds a process streaming a continuous media data, but connected to `Http VIF`, it wraps the process with a dedicated VIF for traffic shaping, and connects it to the stream VIF. If the manager finds a normal process is connected to a stream VIF, it returns the process back to the `http VIF`, deleting the unnecessary traffic shaper.

As a test scenario, we executed Webstone [7] on each client to generate requests for their respective cluster node (e.g., C1 sends burst requests to N1, C2 to N2, and so forth). For the pseudo-stream request, we manually executed `fetch` command.

**Case 2: Premium CGI Service**

In this scenario, the CGI output has highest priority. Overview of this control is depicted in Figure 7. As illustrated in the figure, there are three basic classes. For fair sharing of the premium CGI queue, we wrap each process connected to the queue with a dedicated VIF in fair queuing mode [6]. As described above, the manager monitors the execution mode of each process, and connects the processes to their appropriate VIFs.

As a test scenario, we again executed Webstone for background traffic, and made the intermediate router close to saturation. Then, we manually executed `fetch` command twice, for the CGI requests. For this purpose, we wrote a dummy CGI that simply transfers a large file on the hard-drive.
4.3 Results

To verify the impact of the QoS manager on performance, first, we conducted a run with normal configuration and measured its traffic profile using the same workload as the CGI scenario (see Figure 8). It is easy to observe that each node is out of control, resulting in quite unstable flow.

The two controlled experiments (cases 1 and 2) are shown in Figures 9 and 10. For case 1, we started Webstone at time 0, and a CBR request at time 15. C1, C2, C3 and C4 are background traffic for each client, from their respective source node. N1 is a CBR flow, from node 1 to client 2. We can observe that each cluster node conforms to its calculated upper limit. Further, when CBR request comes, node 1 yielded 4Mbps for the pseudo-stream, as expected.

For case 2, we started Webstone at time 0, and CGI requests at time 15. C1, C2, C3, and C4 are the same as above. N1 and N2 are CGI flows from node 1 to clients 2 and 3, respectively. As shown, each cluster node keeps their limits, and efficiently shared the 45Mbps link. When CGI requests arrive, node 1 throttled its output, to make room for the premium flows. The flows shared the interface fairly, due to the fair queuing control.

5 Discussion

First of all, the scheme worked without modification of the existing cluster software. Further, since it works solely within the end-hosts, it did not require modification of the hardware configuration. However, it is clear that we utilize a customized kernel, which we hope will become a standard feature when the network control services on end-hosts becomes more popular.

Secondly, we demonstrated that the scheme can meet various QoS policies for a server cluster. For example, it worked with priority queuing, fair queuing, and non work-conserving queuing. Further, combination of the VIFs exhibited more powerful control, such as a priority queuing with traffic shaping, which cannot be realized with traditional approach.

However, a limitation must be mentioned. The manager model works perfectly if the server process accommodates one connection at a time. However, if a multi-threaded server accommodating several connections is used, the external control might be inadequate. In such case, we need another mechanism that can inspect internal behavior of the server processes. One possibility is to modify the thread library to pass detailed information about the socket use to the manager, though we have not explored this issue in detail.

Lastly, although not argued in detail, we believe that the proposed scheme can be applied to various scenarios if hosts are kept homogeneous and cooperative. For example,
in a wireless LAN with homogeneous hosts, the managers on wireless nodes would provide simple and easy QoS to the nodes, cooperating together for a certain policy. Even in a typical LAN setting without proper QoS capabilities of network devices, this software solution would contribute to reduce the trouble caused by uncontrolled traffic by introducing the manager into servers with heavy traffic.

6 Related Work

Our event-handler model brings network QoS to the cluster without modification of server application nor clustering configuration. This retrofitting property was inspired by [3]. However, our paper mainly focuses on the application layer components, design of the QoS manager and its configuration mechanism.

Server clustering technology has been developed mainly to gain higher performance, and there exist various approaches; for various network services, such as web server and mail server [10, 4], for load-balancing technology [2], and for provisioning of QoS with cooperating end-nodes [1, 11]. Compared with the existing ones, we focused on the issue of network QoS for server clusters, which has been overlooked thus far.

7 Concluding Remarks

We presented a retrofitting mechanism to provide network QoS for server clusters. We argued that an end-host manager-based approach is a preferable solution, and proposed an event-handler model and an object-oriented scripting language for description of the management algorithm.

The event-handler based QoS manager provided an appropriate framework for network control, and controlled the cluster traffic reasonably. The scripting language made flexible configuration of QoS policies possible, enabling description of dynamic algorithm. The object-oriented feature and the class library simplified description of the algorithm.

We hope our QoS manager solution contributes to establishing control principles and useful algorithms for server clustering, as well as to realize various network QoS with cooperating end-hosts.

Source Code

The source code, implementation and related information are available from http://www.netnice.org.

References