Lustre NRS TBF Policy

Version	Date	Comment	Author
1.0	07/29/2013	Initial revision	Shuichi Ihara sihara@ddn.com

Terminology

Acronym/Term	Meaning	Description
NRS	Network Request Scheduler	A PTLRPC layer subcomponent
TBF	Token Bucket Filter	An NRS policy that schedules and throttles RPCs for traffic control purposes.
FIFO queue	First-In, First-Out queue	Waiting area for incoming RPCs handled in the order of arrival.
QoS	Quality of Service	Special I/O requirements that a given service tries to satisfy.

1 Introduction

The Lustre Network Request Scheduler (NRS) enables services to schedule RPCs in a variety of ways. Using this new framework, various scheduling policies have been implemented. Most of these policies are aimed at improving I/O throughput. The Token Bucket Filter (TBF) policy is aimed at a very different purpose, namely (client) I/O Quality of Service QoS).

TBFs are widely used algorithms in packet-switched computer networks and telecommunications; their goal is to ensure that data transmission rates conform to given limits for bandwidth and burstiness. A simple TBF algorithm can be conceptually understood as follows:

- A token is added to the bucket every 1/r seconds.
- The bucket can hold at the most b tokens. If a token arrives when the bucket is full, it is discarded.
- When a packet of n bytes arrives, n tokens are removed from the bucket, and the packet is sent to the network.
- If fewer than n tokens are available, no tokens are removed from the bucket, and the packet is considered to be non-conformant.

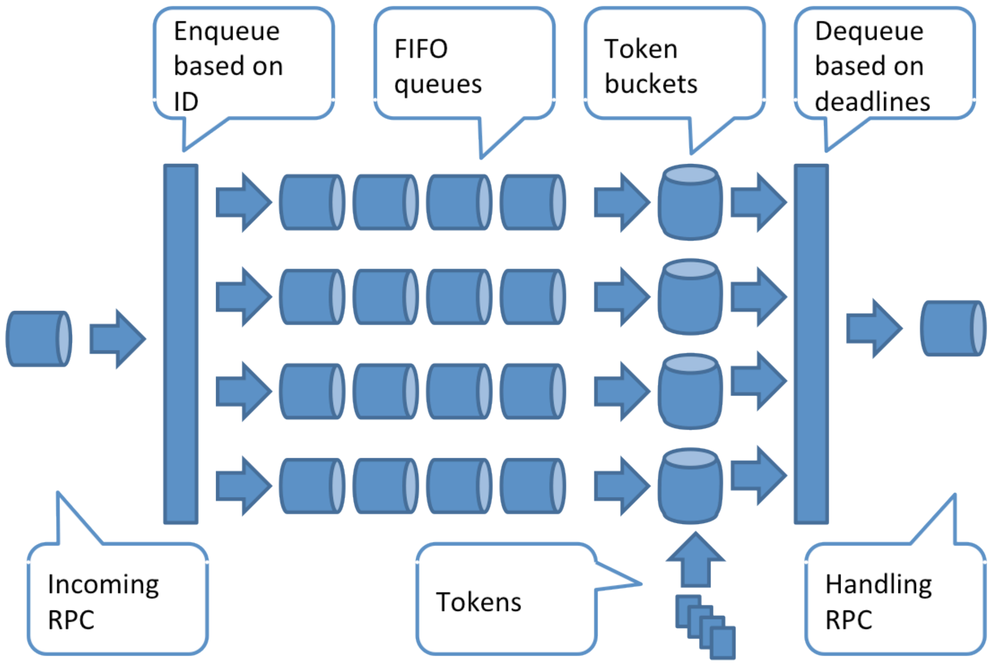
Considering the Lustre file system, the TBF policy has been implemented as follows:

- Non-conformant packets can be treated in various ways in the TBF algorithms. Network traffic control systems normally drop non-conformant packets. But our NRS TBF policy enqueues non-conformant RPC requests, rather than simply drop them.

- Network traffic control systems are able to calculate the size of packets and thus limit the throughput rates precisely in bytes. The NRS TBF policy is facing a different situation. There exist many different kinds of Lustre RPC, and developing a straightforward method to measure the number of tokens that different RPCs consume appears tricky. Thus, the current TBF policy only supports a RPC rate limit, i.e. *one* RPC always consumes *one* token.
- For system administrators, being able to set various types of limits, and for different dimensions, would be highly desirable. For example, we might not only want to limit the RPC rate of each *client,* but also want to limit the RPC rate of each *user*. A powerful QoS mechanism should be flexible enough for this kind of requirements. Network traffic control systems basically stack multiple TBFs (or other kind of traffic control algorithms) layers in order. Today, our TBF policy only supports NID based limits, but we are working on a common layer within the TBF policy that would enable the support other types of RPC limitations, such as limit based on UIDs/GIDs even Job ID.) Further, as soon as the NRS framework supports layered polices, we will be able to define more complex policy within TBF.

2 Design

2.1 Fundamental architecture



GRAPH: Internal Structure of the TBF Policy

When a RPC request arrives, TBF policy puts it to a queue according to its classification. The classification of RPC requests can be based on any of the characteristics owned by the RPC requests, e.g. NID, UID/GID, JobID and their combinations. There are multiple queues in the system, one queue for one category in the classification of RPC requests. The queues are generated automatically at run time. If there is no corresponding queue for an incoming RPC request, a new queue will be created immediately.

Queued requests wait for tokens to be assigned in a FIFO manner. A request will never be dequeued unless it is the first one in the FIFO queue and obtains enough tokens. Every queue has a token bucket to manage tokens.

Conceptually, tokens are inserted into the bucket every **1/r** seconds, where **r** is the token rate set by the administrator. And the bucket can never hold more than **b** tokens. Tokens will be discarded when the bucket is full. In our actual implementation, we don't actually need to insert tokens every **1/r** seconds, which is rather inefficient. Instead, tokens in a bucket are only updated according to the elapsed time and token rate when the queue was actually able to dequeue a RPC request.

In order to enforce the rate limit, the first RPC request in a queue needs to wait until there are enough tokens generated. The queue deadline is the time point when the queue will have accumulated enough tokens to dequeue the RPC request. Queues are sorted in a binary heap according to the deadlines of their first RPC requests. The binary heap waits until it reaches the nearest deadline. Then the RPC request with the smallest deadline will be dropped from the heap and handled by an idle service.

Sometimes, the service is just too busy to handle all of the requests in time, thus deadlines are missed. Note that, if the service cannot satisfy all the scheduled rates, the only thing that happens is that the RPC rates of the queues are *slower* than configured. In this case, because requests in different queues are scheduled according to their deadlines, the queue with higher rates will have an advantage over the queues with lower rates, but none of the queues will be starved.

2.2 Rule-based limit matching

Every queue in the system has a predefined token rate. Instead of manually assigning a taken rate to each queue, queues are automatically matched with predefined rules from which they obtain their respective token rates.

Every rule has a field of associated token rates. Rules are organized as ordered lists. Whenever a queue is newly created, it goes though the list of rules, selects a rule that matches and, finally, obtains its token rate.

A rule can be added to, or removed from the list at run time. Whenever the list of rules is changed, the queues must update their associated rules. Note that, in the actual implementation, and because of performance reason, an associated rule is only updated when a queue actually try to use it.

3 Usage

3.1 Start policy

The format of the start command for a TBS policy is as follows:

```
lctl set_param x.x.x.nrs_policies="tbf [reg|hp] <type>"
```

The argument 'type' is the classification type of RPC requests. Currently, only 'nid' is valid, since only NID based classifications are supported at present:

```
lctl get_param x.x.x.nrs_policies
```

The following commands are valid:

```
lctl set_param ost.OSS.ost_io.nrs_policies="tbf nid"
lctl set_param ost.OSS.ost_io.nrs_policies="tbf reg nid"
lctl set_param ost.OSS.ost_io.nrs_policies="tbf hp nid"
```

3.2 Start rule

The format of the rule start command for a TBS policy is as follows:

```
    lctl set_param x.x.x.nrs_policies="[reg|hp] start <name>
<arguments>..."
```

The 'name' argument is a string that identifies a rule. The format of the 'arguments' is changing according to the type of the TBF policy used. For the NID based TBF policy, the format is as follows:

```
lctl set_param x.x.x.nrs_policies="[reg|hp] start <name> {<nidlist>}
<rate>"
```

The 'nidlist' argument uses the same format used to configure an LNET route. The 'rate' argument is the RPC rate of the rule.

The following commands are valid (Please note that a newly started rule is *prior* to old rules, so the order of starting rules is critical):

```
    lctl set_param ost.OSS.ost_io.nrs_tbf_rule="start loginnode
{192.168.1.1@tcp} 10000"
    lctl set_param ost.OSS.ost_io.nrs_tbf_rule="reg start loginnode
```

```
{192.168.1.1@tcp} 10000"
   lctl  set_param  ost.OSS.ost_io.nrs_tbf_rule="hp  start  loginnode
{192.168.1.1@tcp} 10000"
   lctl  set_param  ost.OSS.ost_io.nrs_tbf_rule="start  computenodes
{192.168.1.[1-128]@tcp} 1000"
   lctl  set_param  ost.OSS.ost_io.nrs_tbf_rule="start  other_clients
{192.168.*.*@tcp} 100"
```

3.3 Change rule

The format of the change rule command for a TBF policy is as follows:

```
lctl set_param x.x.x.nrs_policies=”[reg|hp] change <name> <rate>”
```

The following commands are valid.

```
lctl set_param ost.OSS.ost_io.nrs_tbf_rule="change loginnode 1000"
lctl set_param ost.OSS.ost_io.nrs_tbf_rule="reg change loginnode 1000"
lctl set_param ost.OSS.ost_io.nrs_tbf_rule="hp change loginnode 1000"
```

3.4 Stop rule

The format of the rule stop command for a TBF policy is as follows:

```
lctl set_param x.x.x.nrs_policies=”[reg|hp] stop <name>”
```

The following commands are valid:

```
lctl set_param ost.OSS.ost_io.nrs_tbf_rule="stop loginnode"
lctl set_param ost.OSS.ost_io.nrs_tbf_rule="reg stop loginnode"
lctl set_param ost.OSS.ost_io.nrs_tbf_rule="hp stop loginnode"
```

4 Future work

At this moment, NID only supports NID based limits. We are working on the support of other kinds of limits, e.g. UID/GID or Job ID based limits. We expect all policies to share a common framework, and we expect to keep the user interface compatible with the current version.

Combinations of different types of limits will be possible as soon as the NRS framework supports layered policies. This will enable simultaneous control of I/O traffic and I/O load at various levels within the I/O subsystem.