Server Driven Quality of Service for Network IO

Cluster File Systems, Inc.
Sep 10, 2006, v1.2

Peter J. Braam & Eric Barton

1. Problem Description

In certain environments IO requests from client nodes to servers must take into consideration the origin to achieve appropriate throughput qualities at clients.

2. Requirements

3. Summary

The driver has two subsystems. There is a Local Request Scheduler (LRS) which is responsible for receiving incoming Lustre RPC requests and scheduling them to a service handling entity. The second subsystem is an epoch hander (EH) which provides synchronization and scheduling policy data reduction service to LRS instances.

The LRS is further specialized to allow scheduling policies based on epochs. It may chose to prioritize requests differently depending on the current epoch – a flexible concept of time-slice.

Epochs are of particular interest when they are global among multiple LRS instances. This allows the LRS instances and EH to agree when to start the next scheduling epoch and how to process requests during that epoch

4. Local Request Scheduler (LRS)

A local request scheduler is implemented as a code module running on a server exporting the methods described below. The APIs described here are its abstract methods. They are described from the point of view of someone implementing a scheduler.

4.1 Functional Specification

```
typedef void (lrs_epoch_start_cb_t)(__u64 epoch_number,
                                void *reduced_data,
                                unsigned int reduced_data_length,
                                void *arg);

struct local_request_scheduler *lrs
{
      void (*lrs_incoming_request)(struct local_request_scheduler *lrs,
                              struct ptlrpc_request *req);
      int (*lrs_get_next_request)(struct local_request_scheduler *lrs,
                              lrs_request_callback_t *req_ready_cb,
                              void *req_ready_arg);
      int (*lrs_init)(struct local_request_scheduler *lrs,
                        struct epoch_handler *eh);
      void (*lrs_shutdown)(struct local_request_scheduler *lrs);
      void (*lrs_fini)(struct local_request_scheduler *lrs);
      static lrs_epoch_start_cb_t *lrs_start_epoch;
      void *lrs_data;
}
```

```
int lrs_init(struct local_request_scheduler *lrs,
             struct epoch_handler *eh);
```

This function initializes and starts the scheduler running. After initialization, it calls the event handler's `next_epoch()` method to indicate that it is ready to start its first epoch. When the epoch handler calls back to signal the start of the epoch it will now be "in synch" with its peers.

```
void lrs_shutdown(struct local_request_scheduler *lrs);
```

This procedure tells the scheduler that shutdown has started. After this, there will be no further calls to `lrs_incoming_request()`. The scheduler can be assured that all currently buffered requests will be consumed by calls to `lrs_get_next_request()` after which further calls should be completed immediately, passing the callback a NULL request argument. At this time, the scheduler must tell its epoch handler to disengage from its peers by calling its event handler's `last_epoch()` method.

```
void lrs_fini(struct local_request_scheduler *lrs);
```

The procedure tells the scheduler that all activity has ceased and all resources can be freed.

```
void lrs_incoming_request(struct local_request_scheduler *lrs,
                          struct ptlrpc_request *req);
```

This procedure accepts an incoming request and places it into the scheduler for future dispatch. It may not fail, it may not block, and it may not call into LNET. Only the following fields of the request structure may be referenced…

`rq_peer` to identify the sender
`rq_list` to add the request to any queues maintained by the scheduler.

```
typedef int (lrs_request_callback_t)(struct ptlrpc_request *req,
                                     void *arg);
```

```
int lrs_get_next_request(struct local_request_scheduler *lrs,
                         lrs_request_callback_t *req_ready_cb,
                         void *req_ready_arg);
```

This function asks the scheduler for a new request. When the scheduler has one available, it runs the callback function passing it the request and the opaque callback argument. It may only fail in truly exceptional circumstances – the caller is liable simply to try again in a short while (XXX what does this mean – perhaps make this void?)

The callback function that is passed in may not block (? – can the callback function run in the context of the caller if a request is available?)

```
static void (*lrs_start_epoch)(struct local_request_scheduler *lrs,
unsigned long epoch, void *epoch_data, unsigned epoch_data_len);
```

This is a private method. It instructs the lrs to start scheduling requests in the next epoch. It must be followed by a call to eh_next_epoch within the eh_timeout.

4.2 Use cases

In the use cases below, all procedures will be called in thread context.

lrs_init

Several LRS modules can be provided. Each will export a `struct _local_request_scheduler`. A configuration option to Lustre will specify which request scheduler will be used.

The service initialization function starts the request scheduler by calling this function.

lrs_shutdown

The thread shutting down services calls this during service teardown, after all service request buffer MDs have been unlinked.

lrs_fini

This is called at the end of service teardown, after all service threads have exited.

lrs_incoming_request

~~Please note that this method is not allowed to block because the caller might be holding spinlocks which could serialize all network communications. However it is still OK to do memory allocations which could potentially block for memory to be freed since this avoids excessive reliance on atomic allocation.~~

Called each time the event handler of the LNET event queue associated with incoming requests runs to indicate a new request has arrived. Although this is guaranteed to be in thread context, blocking here will hang all network communications for the duration. So it's OK to do non-atomic memory allocations which might block at moments of extreme memory pressure, but not to wait for external devices, and certainly not to wait for other network communications.

lrs_get_next_request

~~Please also note that this method may not block indefinitely – they should complete at least within the lustre timeout under all circumstances.~~

~~This procedure may not block (why not?) or call into LNET – why not, because the caller ….. (this must be a requirement imposed by the use case).~~

This function is called by a service thread in its main loop when it is ready to block for a new request to be available for processing. The callback argument would typically be used to wake up the sleeping service thread, which will exit if the request passed back is NULL.

This function is not allowed to block apart from for memory as described above – a future lustre implementation may use an asynchronous I/O model with 1 thread per CPU, so blocking here would hang request servicing.

lrs_start_epoch

~~Please also note that this method may not block indefinitely – they should complete at least within the Lustre timeout under all circumstances.~~

This procedure is a private method for the request scheduler which it passes as an argument to `eh_next_epoch()`. The epoch handler calls this function when the next epoch event is delivered.

4.3 Examples

A fifo scheduler would implement `lrs_incoming_request()` as a list insertion at the head of a list and `lrs_get_request()` as a function that removes, a request from the tail of the same list when the list is not empty.

A scheduler that doesn't care about epoch functionality can use the local epoch handler, which completes all the epoch methods (first, next, last) immediately.

A single LRS scheduler cares only about epochs on one node, and may be useful, for example, during initial testing of the scheduler with one server. This can work with a trivial local epoch handler. The local epoch handler would simply reduce policy data in eh_next_epoch and call the lrs_epoch_start callback, without waiting or synchronization.

A scheduler that does care about global epoch functionality must also provide its epoch handler with a reduction function that combines scheduler policy data from multiple LRS instances. Such a scheduler might provide its next choice of client groups to schedule for and preferred epoch length when it calls the epoch handler's eh_`next_epoch()` method. When this is reduced and effectively broadcast via the epoch callback, every scheduler can start serving requests only from the preferred group. When there are no more requests from that group, or the preferred epoch length has expired each scheduler will once more call eh_`next_epoch()`. Note that it may continue to serve requests from this group until the eh_`next_epoch()` callback signals that all its peers are also ready for the next epoch.

4.4 Logic Specification

5. Global Epoch Handler (EH)

The epoch handler distributes LRS policy data and coordinates when to advance to the next epoch.

The epoch handler implements a non-blocking distributed reduction. Each LRS signals readiness for participation in the next epoch by calling its EH's eh_`next_epoch()` method. When all responsive LRS-s have done this and all failed LRS's have been removed from participation (evicted), every functioning LRS's lrs_epoch_start callback is called with the reduced policy data. This signals the start of the next epoch - effectively broadcasting the global scheduling decision.

Note that it is up to the scheduler policy whether to continue scheduling requests after eh_next is called or whether to, for example, generate a quiescent phase in preparation of the next epoch.

The implementation details of the EH are not specified here, but there is a possible implementation using a Lustre lock server.

5.1 Functional Specification

The methods specified below are only those that are immediately relevant to the LRS and they are described from the point of view of someone implementing a scheduler. Other methods will be specified later.

```
struct epoch_handler {
      unsigned int eh_timeout;
      void (*eh_combine)(void *combined_data,
                 void *data a, unsigned int data_length_a,
                 void *data b, unsigned int data_length_b);
      void (*eh_swab)(void *data, unsigned int data_length);
      void (*eh_next_epoch_(struct event_handler *eh,
                    _s64 epoch_number,
                    void *data,
                    unsigned int data_length,
                    epoch_callback_t *callback,
                    void *arg);
```

```
unsigned int eh_timeout;
```

The EH timeout is specified by the ~~LSR~~ LRS in seconds. It is used to detect unresponsive schedulers which may have become stuck or crashed. The timeout starts to expire when the first ~~LSR~~LRS signals it is ready for the next epoch. Any ~~LSR~~LRS which has not signaled that it too is ready when the timeout expires is evicted from the set of EH participants and is ignored until it rejoins correctly.

```
void eh_combine(void *combined_data,
                void *data a, unsigned int data_length_a,
                void *data b, unsigned int data_length_b);
```

This procedure is provided by the LRS to combine 2 sets of LRS policy data. The function performed on the data should be associative (grouping independent) and commutative (order independent). `combined_data` is only guaranteed to be as large as the maximum of the 2 supplied data lengths.

```
void eh_swab(void *data, unsigned int data_length);
```

This procedure is provided by the LRS to convert LRS policy data that was received from a different-endian peer to the local byte-order.

```
__s64 epoch_number;

#define EH_FIRST_EPOCH        0
#define EH_EPOCH_ERROR(e)     ((e) < 0)
```

An epoch number is a 64-bit signed integer. It never wraps. Valid epoch numbers start from 1 and are positive. A negative epoch number can be interpreted as a unix error number.

```
void eh_next_epoch(struct event_handler *eh,
                   _s64 epoch_number,
                   void *data,
                   unsigned int data_length,
                   lrs_epoch_start_cb_t *lrs_epoch_start_cb,
                   void *arg);
```

`epoch_number` may be EH_FIRST_EPOCH if the LRS does not know the current epoch number (e.g. on the first call ever, or after eviction XXX why not send the last epoch after an eviction – finally the LRS doesn't KNOW it was evicted) . Otherwise it must be the current epoch number. This may be used to assert correct functionality or allow looser synchronization in future implementations.

`data` is a pointer to a local buffer containing `data_length` bytes of LRS policy data that is opaque to the EH. Normally all LRS-s will pass the same sized buffer. This buffer may be read or overwritten at any time between calling `eh_next_epoch()` and receiving the completion callback.

There may be restrictions on `data_length` – it may have to be less than some maximum size either statically defined or declared when the EH is initialized. Not meeting such a restriction could trigger an assertion failure. In any event, LRS implementers may not rely on it being larger than 4096.

`lrs_epoch_start_cb()` is a completion callback to make when the next epoch starts. It is passed the current epoch number on success or a negative error number on failure. A buffer containing the reduced LRS policy data is passed back via `reduced_data` and its size via `reduced_data_length`. This size is the minimum across all participants so it cannot be larger than what was supplied locally. Note that this buffer may in fact be the original buffer passed to `eh_next_epoch()`.

The functional specification of this function is described in the state chart below.

```
void eh_last_epoch(struct event_handler *eh,
                   long epoch_number);
```

The LRS calls this procedure to disengage from the EH. The correct `epoch_number` must be passed, but return is immediate.

5.3 State machine

An ~~eh~~ EH is in one of the following states:

EH_IDLE
Transition to: (1) this is the initial state, no eh_next_epoch call has been made yet.
(2) from EH_CLOSING when all LRSs were evicted or called eh_last_epoch
Transition from: (1) to EH_WAITING an eh_next_epoch call is made

EH_CLOSING
Transition to: (1) from EH_WAITING when an eh_next_epoch is received.
Transition from: (1) to EH_WAITING when all participants have made the eh_next_epoch or eh_last_epoch call and non-responsive participants have been evicted and some participants remain for which the lrs_epoch_start_cb is run.
(2) to EH_IDLE when no participants remain after the timeout and evictions and eh_last_epoch calls.

EH_WAITING
Transition to: (1) from EH_IDLE or EH_WAITING when lrs_epoch_start_cb is called
Transition from: (1) to EH_CLOSING when an eh_next_epoch is received.

5.4 Use cases

eh_next_epoch()

The LRS calls `eh_next_epoch()` to indicate it is ready to join in the next epoch.

Unresponsive LRS

An LRS which is not *responsive* will be deemed to have failed and will be evicted from the EH's set of participants. Note that the timeout which determines responsiveness bounds the "raggedness" of the EH epoch barrier rather than the length of any individual epoch.
A second failure of responsiveness arises when an lrs_epoch_start_cb cannot be delivered within a short time.

Failing EH entity

The epoch callback is guaranteed to happen within (short) finite time after the epoch timeout has expired. If this occurred because the local EH could not synchronize with its peers, this is signaled by returning a negative error code rather than a (positive) epoch number in the lrs_epoch_start callback. The LRS should take whatever action it deems sensible to handle outstanding requests while it remains disconnected from its peers (including ignoring them). Its eh_next_epoch message and subsequent lrs_epoch_start_cb will be delivered by the EH.

5.5 Examples

Examples of a scheduling epoch can be the processing of requests from a certain zone, requests from a certain Lustre network or it can be a round robin pass through all zones, or all Lustre networks. This is determined by the LRS policy.

6. Implementation issues