High level design of a Lustre Network Request Scheduler

By Nikitas Angelinas <nikitas_angelinas@xyratex.com>
Date: 2011/04/11
Revision: 1.0

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document.]

This document presents a high level design (HLD) of a Network Request Scheduler for Lustre. The main purposes of this document are: (i) to be inspected by Lustre architects and peer designers to ascertain that high level design is aligned with Lustre architecture and other designs, and contains no defects, (ii) to be a source of material for detailed level design (DLD) and optionally Active Reviews of Intermediate Design (ARID) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of Lustre customers, architects, designers, developers, and management personnel.

0. Introduction

[This section succinctly introduces the subject matter of the design. 1--2 paragraphs.]

The Network Request Scheduler (NRS) is a software component that operates within the PTLRPC layer and in conjunction with layers that make use of PTLRPC services, such as the OST layer in OSS nodes and the MDD layer in MDS nodes. Its main purpose is to influence the execution of RPC requests by altering the ordering these are disposed at from PTLRPC services, in order to achieve an *effect*. In a rather common case, the aim is to obtain increased overall throughput for OSS nodes, by reordering bulk OST_READ and OST_WRITE RPC requests in a way that is likely to be easier for the underlying OS disk I/O scheduler to optimize, in order to minimize costly seek times for OSTs that employ rotating media[1]. Alternatively, NRS behaviour can be tailored to fit more specialized workloads, such as prioritizing filesystem clients, or providing guarantees relating to throughput seen by some clients (see [2]). The present document depicts a server-side request scheduler, i.e. one that operates on OSS and MDS nodes only, with no modifications required to client nodes, although the latter would be a valid design option in a particular implementation, and could perhaps extend the potential of NRS deployments (for an example of this, see [2]).

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the a any project-wide glossary or similar construct that may exist are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

---

1. On OSS nodes that employ solid state memory devices, a different scheduling algorithm could be used, or NRS functionality could be disabled.

- A *policy* is the generic course of action that is taken by a particular NRS adaptation., and is characterized by the nature of the *effect*.
- A *scheduling algorithm* is a subcomponent implementing a particular policy.
- The *effect* is the end goal of a particular policy implementation; this can optionally be a quantifiable measure.

Additionally, some definitions that are related but may not all directly apply to the content of the present document, can be found in [0].

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of development. References to the appropriate RA and RI documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.][2]

[r.nrs.non-disruptive]: the inclusion of NRS should not in any way disrupt the operation of the cluster.

[r.nrs.scheduler-various]: the core NRS framework should allow for various scheduling algorithms to be used [4] (and optionally to be substituted at run-time).

[r.nrs.scheduler-improve-throughput]: the main scheduling policy to be used should aim to improve overall throughput seen at the OSS where NRS is deployed [4].

[r.nrs.scheduler-fair]: the main scheduling policy to be used should cater for fairness amongst I/O requests [4].

[r.nrs.scalability]: the feature should result in no reduction in the server-side RPC handling rate.

[r.nrs.scheduler-network-unfairness]: the fair scheduling policy may optionally try to compensate for any observed network unfairness, by selectively prioritizing clients in order to, in effect, cause I/O operations to take place between the server and specific client nodes, in a way that balances network utilization.

[r.nrs.scheduler-client-prioritization]: a policy should be implemented to cater for prioritization amongst filesystem clients [4].

---

2. RA and RI phases have not been formally undertaken for the task. The requirements listed here stem either from architectural descriptions or prudence. As such, they are not intended as inputs to an (semi) automatic requirements tracking mechanism, but are rather listed here to assist in manual requirements tracking throughout appropriate phases of the project. Some of the requirements listed are also optional for the design, something which may not be in line with established policies.

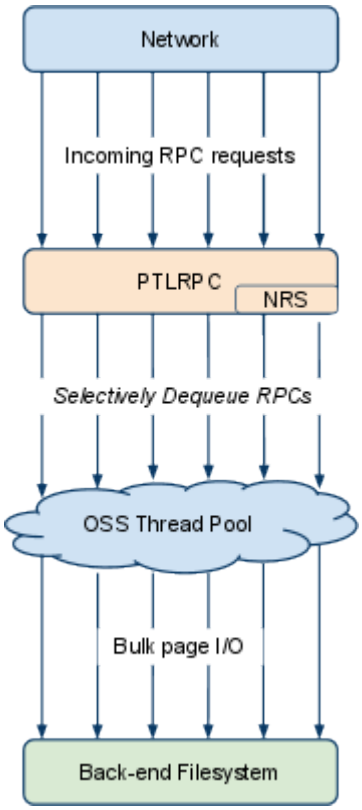
3. Design highlights

[This section briefly summarises key design decisions that are important for understanding of the functional and logical specifications and enumerates topics that a reader is advised to pay special attention to.]

A thread-less software component is inserted inside the PTLRPC layer; NRS allows for different methods (policies in NRS parlance) to be implemented that can influence the manner in which RPC requests are handled by a PTLRPC service. Each policy registers itself with the core NRS framework, and defines what RPC types it provides special handling for; for RPCs which are not directly manipulated by the registering policy, the policy denotes at registration time a secondary policy to be used for these RPCs. PTLRPC service threads also denote what policy/policies they are willing to handle RPCs for at service initialization time, and are used to implement that particular policy (policies) by handling the denoted types of RPCs, such that the desired *effect* is achieved.

4. Functional specification

[This section defines a functional structure of the designed component: the decomposition showing *what* the component does to address the requirements.]



The schema above, which has been adapted from [1], conceptually depicts the generic idea behind an NRS policy that is applied to an OSS node in order to provide with reordering of the manner in which OST_READ and OST_WRITE requests are dispatched. As requests arrive from the network via the OSS PTLRPC ost_io service, normally, they would be queued by one of the ost_io service threads in the RPC servicing queue for the service, and would at a latter point be dequeued and dispatched by one of the same service threads, in order to progress the two-phase bulk I/O operation.

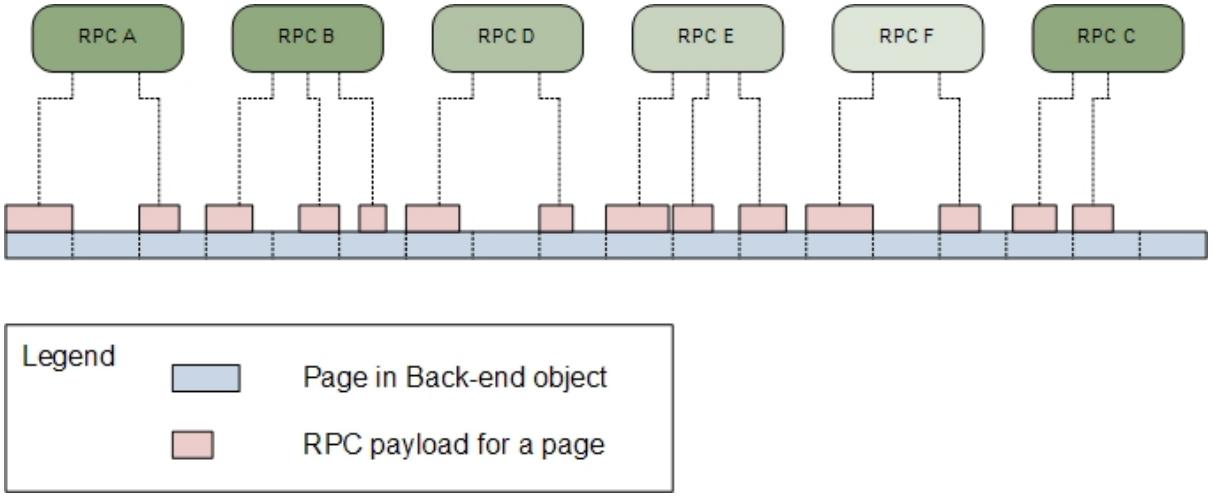
The NRS policy intercepts the normal FIFO manner in which OST_READ and OST_WRITE RPC requests are handled, and instead rearranges the order these are dispatched in, in order to achieve the desired *effect*. As mentioned elsewhere in this document, in the generic case, the targeted effect will be an increase in overall throughput seen in the operation of the OSS. In order to achieve this, the NRS policy will examine the page ranges involved in I/O operations, and attempt to dispatch requests in a manner that is likely to be easier for the underlying disk I/O scheduler to re-arrange, in order to present a workload to the storage media that will result in reduced overall seek times, in cases where rotating media are employed[3].

As with disk I/O schedulers, the aspect of fairness amongst I/O requests will need to be catered for by an NRS algorithm which essentially performs sorting operations amongst the ordered set or a subset of the assembled I/O requests. A problem arises in that, as the NRS algorithm is selectively dissociating requests in a way that requests in the current working set are dispatched in some order different to the order of arrival, additional requests that continue to arrive and are added to the working set may cause some requests to be *unfairly* delayed, thus introducing the potential for *starvation,* unless this is catered for by the algorithm.

In an example scenario where a given NRS scheduling algorithm is to serve requests (assembled in queue structures) on a per-object basis and in ascending order according to file offset (as done in [3], the schema below depicts a number of requests pertaining to an object in the back-end filesystem. Assuming PTLRPC requests $\{A, B, C\}$ have been queued for service and comprise the working set of requests for the object, the NRS algorithm would start serving 'RPC A' before it moves to 'RPC B'; but if, before the algorithm is finished serving 'RPC B' additional requests D, E and F arrive for service from the network, a perhaps naive algorithm would in all cases prefer to serve these requests before it handles 'RPC C'. Since additional requests can continue arriving beyond only $\{D, E, F\}$ before 'RPC C' receives any service time, it is possible for 'RPC C' to be delayed for long periods of time, potentially leading to a starvation situation. An NRS scheduling algorithm that aims to apply to the average workload in the general case, should thus also cater for fairness amongst the set of I/O requests to be serviced.

---

3. Since head seeks in a disk drive involve movement of mechanical parts (the actuator arm carrying the read/write head), they are very time-consuming relative to other activity that occurs during a data transfer; in OS kernels, it is the aim of I/O scheduler to minimize seek times, using algorithms that employ I/O request sort and merge operations, and other techniques.



In order to accommodate the requirements of different workloads, the NRS component will allow for the ability to make use of different NRS *scheduling algorithms,* each implementing an NRS *policy*. Whereas the NRS algorithm that will apply to a generic workload will aim to achieve increased average throughput across the OSS, other workload characteristics can be catered for by using different algorithms. In example, I/O requests originating from some of the cluster clients can be given higher priority during dispatching [4]. A similar usage of NRS which demonstrates further possibilities that may arise if additional information is passed from client nodes to the OSS, can be seen in [2], where an NRS implementation has been used to provide with a QoS-aware mechanism across small-scale clusters, thus fulfilling guarantees regarding the throughput witnessed by a client, that are imposed upon the storage cluster as a result of a higher-level Service License Agreement between the end-user and the storage service provider.

5. Logical specification

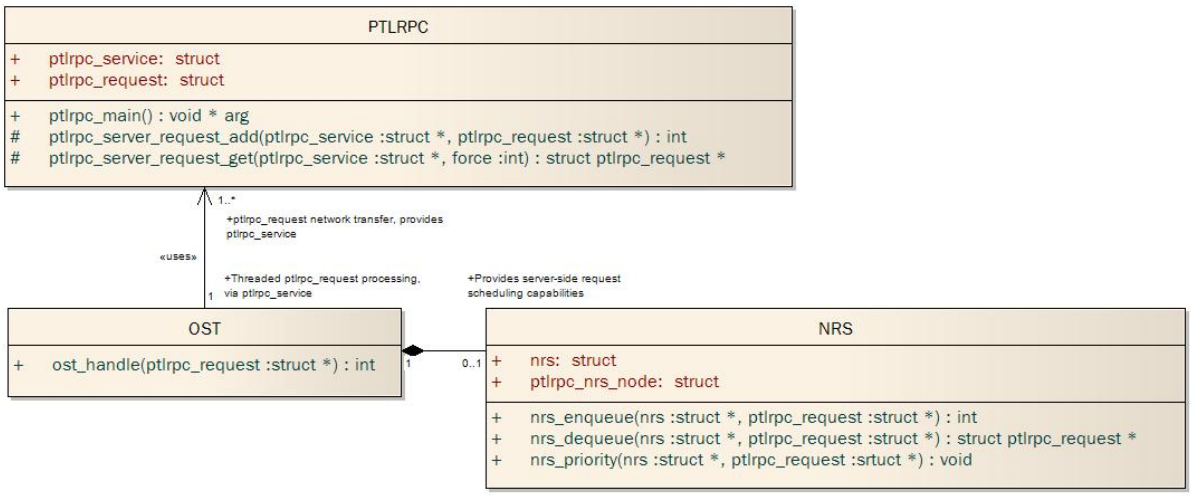
[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

5.a. Core NRS Framework

TODO: Change this to reflect actual operations used.

The NRS component needs to intervene with the normal flow of RPC handling in server nodes. An example of how this can be achieved in OSS nodes can be seen in [3]. The prototype NRS implementation hooks into the RPC handling paths in order to influence the queueing and dequeuing of RPCs. It is assumed that a production-level implementation will use similar hooks at the same points in the path; as the paths are common to server nodes irrespective of type, a null-policy option will exist that can be used on server types which may not wish to make use of NRS, in which case the effects of the option would be optimized-away by the compiler.

The partial class diagram below denotes part of the additional interface the NRS component introduces to OST, in the prototype NRS implementation in [3]. A traditional function pointer table can be used to allow for different scheduling algorithms to be hosted using the core NRS framework, via a registration interface that is invoked at OSS startup time. Operations *nrs_enqueue()* and *nrs_dequeue()* can be used from within *ptlrpc_main()* in PTLRPC service threads to tailor the operation of *ptlrpc_server_request_add()* and *ptlrpc_server_request_get()*; this is necessary as these are likely code paths where NRS scheduling algorithms could utilize to implement their undertakings.



The *nrs_priority()* operation allows an NRS scheduling algorithm to intervene with the handling of the high priority RPC queue. This is necessary as there may be instances where an RPC that is being handled by an NRS policy is moved into the high priority RPC queue for the service from a racing context when *ptlrpc_hpreq_reorder_nolock()* is invoked from *ldlm_server_blocking_ast()*, in cases where the RPC is blocking an LDLM lock cancellation; a particular NRS algorithm may need to compensate for occurrences of the above situation by e.g. removing a request from its internal queueing structures.

The core NRS framework will allow for different scheduling algorithms to become operational on demand. This can either occur according to user preference, as signified by interaction with a pseudo-filesystem or similar interface, or depending on internal to the core NRS framework logic, that may yield a preference to a particular scheduling algorithm, depending on measurement of the *effect*, if applicable to a particular policy or algorithm.

5.b. Fair scheduling policy

A fair scheduling algorithm needs to be produced that will cater for the generic case and will aim to provide an increase in overall throughput rates seen across OSS nodes; this will also act as a means to test the operation of the core framework in non-trivial setups. Matters of prime importance that should be considered for algorithm design are best captured by [r.nrs.scheduler-

improve-throughput], [r.nrs.scheduler-fair], [r.nrs.scheduler-fair-scalability] and [r.nrs.scheduler-client-prioritization].

It is assumed that an adaptation of the Object-Based Round Robin (OBRR) algorithm presented in [1] will allow for fulfilling the first three requirements to an extent. OBRR divides incoming requests in an OSS node into per-object sets, and maintains these in assorted order according to file offset. Sets that are non-empty and ready to be serviced are added to a global queue that is handled by OSS threads in FIFO order in [1], however it is likely that a more CPU cache-friendly structure could be devised[4]. The implementation in [1] maintains two separate subsets per object for OST_READ and OST_WRITE requests, however it is assumed that a single set may prove to be even more efficient, as read and write requests can be interleaved in the manner they are dispatched from rotating storage media. As an object is a separate file in the back-end filesystem, the back-end filesystem allocator will use preallocation strategies which will aim to place related blocks in adjacent tracks on the rotating medium. It is thus expected that dispatching requests that belong to the same object in a non-interleaved manner, will result in reduced overall seek times from rotating media.

However, concerns have been expressed that OBRR may not constitute an optimal scheduling algorithm for fair scheduling, and that one that would take into consideration the particular user on each client which initiates an I/O request could perhaps allow for better overall locality of reference during I/O accesses, thus yielding better overall performance. This could be implemented in the form of a User-Based Round Robin (UBRR) scheduling algorithm, which deviates from OBRR in that the object-based sets are replaced by client-based sets[5]. As it is perhaps not easy to quantify and ascertain a priori which algorithm would yield the best performance, it may be required to pursue both implementations in order to obtain measurable evidence of their effectiveness.

However positive the effects of using OBRR, UBRR or UBVS (see relevant footnote under the present section) may be to the overall throughput figures, serving requests on a per-entity set basis in an RR or VS manner may introduce the potential for starvation of a request, as shown in *section 4,* depending on the order of request dispatching for every per-entity set. The problem is potentially introduced in cases where the dispatching order is not a function of the temporal order of request arrivals.

Thus, potential ways to address the issue of starvation may involve introducing characteristics into an NRS algorithm that would preclude the possibility of starvation ever occurring, in return for an anticipated less efficient algorithm. One potential implementation could place requests on per-entity sets and serve requests in each set in FIFO order (in an RR or VS fashion between sets). This would naturally cater for fairness amongst the requests, and exploit some of the on-disk locality of blocks, as requests belonging to the same file or client will be issued in approximal order.

However, leastwise in the case of OBRR, it would be preferential if spatial request ordering according to file offset is also taken into account, by e.g. dispatching requests by offset order in a file. A potential such implementation, could involve *bounding* the number or aggregate size of

---

4. As each per-object set will contain more than one request, it will be accessed by more than one OSS ost_io thread in evey RR quantum. Caching efefcts may negatively affect performance if every thread needs to modify the same per-object set in order to remove requests as they are handled.

5. It is worth noting that an adaptation of UBRR, where the fixed RR quantum is replaced by a variable-size slice, thus giving rise to a User-Based Variable Slice (UBVS) algorithm, would also allow for meeting requirement [r.nrs.scheduler-client-prioritization].

I/O requests that take place on a single object set per RR quantum. By setting a relatively small number as the limit, it is expected that the algorithm will yield a small increase in overall performance, but without the possibility of delaying any of the assembled requests for large amounts of time; this could potentially be achieved by only allowing up to the specified number, or aggregate I/O size of requests to be assembled in OBRR data structures, and queueing the remaining of the requests per object into the normal RPC service queue used by ost_io threads. Although this adaptation of OBRR will not yield the most efficient implementation possible, it is expected to provide with a useful solution that may assist in establishing a working and bug-free core NRS framework, and serve as the basis for implementing more efficient NRS algorithms thereafter.

An OBRR implementation that takes into consideration the relative offsets of requests for each object, and exhibits *unbounded* sorting behavior, could cater for fairness by the addition of *deadline* values associated with requests, in a similar manner as used in disk I/O schedulers. The NRS algorithm would serve requests from per-object sets as normal, but interrupt its normal flow of operations once a request's deadline becomes expired; it could then serve any requests with expired deadlines and then continue to execute the current RR quantum. As is pointed out in [1], however, a pitfall may be introduced when making use of deadlines of fixed nominal values, by the fact that an OSS server in a real-life cluster can become arbitrarily loaded with requests from thousands of clients; a *static* deadline value would be difficult to determine; too small a value would yield high numbers of expired requests in larger setups, while too large a value and deadlines would fail to adequately yield requests that should be regarded as suffering from starvation. To this end, [1] proposes a *dynamic* deadline heuristic that can operate alongside the static deadline value, which involves benchmarking of the server node's capabilities at server setup time. A more elegant solution could perhaps involve substituting both deadline values with a single heuristic that makes use of the adaptive timeouts mechanism in OSS nodes or is otherwise a function of the total request load across the OSS node, in order to produce a dynamic deadline value which is a function of temporal server load.

Additional features that could potentially improve the behaviour and performance of the fair NRS scheduling algorithm could include **[a]** using reduced weightings when determining dynamic deadline values for read requests, as these will generally cause the calling application to block, whereas write requests follow an asynchronous completion path, **[b]** taking into consideration the presence of pages in the OSS page cache, i.e. an OST_READ operation could be dispatched and completed without incurring any seek time in cases where all the required pages exist in the cache; such operations could be treated differently by the NRS algorithm, by e.g. being dispatched at the start of each RR quantum[6] **[c]** adding an additional heuristic to replace the strictly RR manner in which per-object sets of requests are served in OBRR, in favour of a more dynamic timeslice allocation strategy, akin to what would be performed in UBVS (see relevant footnote under the present section), **[d]** the dispatching of RPCs from high priority request queues can be deferred until the current set of requests is dispatched, such that the disposal of a set of requests is not interrupted mid-stream, **[e]** in addition to the above, a given scheduling algorithm could also respond to observed unfairness introduced by the network, by requesting bulk I/O transfers from specific clients, thus making more informed decisions about how requests should be dispatched; in a rather more involved scenario, clients could also transfer additional information within each I/O request, that would help the algorithm identify related requests (see [0], *'Offset Stream Consistency'*).

---

6. This would also apply to other algorithms

5.b.a OBRR with a request number limit

TODO: Need to change all this to show the results of the HLDINSP, and the method Nathan suggested.

TODO: Does this need a copyright message or similar, to show it's Xyratex property?

The first fair scheduling policy that will be implemented will use an OBRR algorithm, applied on OSS nodes only and will handle OST_READ and OST_WRITE requests, while making use of existing functionality to provide handling of other types of RPCs; the existing functionality will be wrapped into a default scheduling policy, that can also be used as the secondary one by other policies. The OBRR policy will implement a proc-adjustable limit on the maximum number of requests OBRR is allowed to reorder per object grouping; this will allow for a simpler implementation that should provide a measurable increase in overall performance while circumventing issues related to request starvation, and will serve as a platform for implementing more ambitious scheduling policies in the future.

In this implementation of OBRR, there is an object load descriptor for every object which has had requests arriving that pertain to its data. The object load descriptor keeps track of information that is used in carrying out reordering operations on an object's RPCs, and maintains internal OBRR state pertaining to the object. The following steps describe how object load descriptors are used in OBRR to coordinate the handling RPCs:

1. Once the first request for an object is handled by a service thread for pre-processing and addition to the RPC servicing queue, an object load descriptor is allocated and added to the RPC servicing queue, which amongst other things keeps track of the request. A hash table entry is created that uses the object ID as the hash key, and is used to resolve to the object load descriptor during future OBRR operations.

2. When a second request for the object is handled for pre-processing, it obtains the object load descriptor, and uses the information held therein to sort-add itself into the RPC servicing queue with respect to the file offset of the operation it carries and the file offset of the operation carried by the first request; operations are hence left sorted in ascending file offset on each per-object grouping in the RPC servicing queue.

3. Subsequent requests are also sort-added to the object grouping in an identical manner, so long as the RR limit for the object has not been reached; this allows OBRR to conveniently circumvent any issues that could potentially give rise to a request starvation situation. As requests from the RPC servicing queue are handled by the PTLRPC servicing threads, the RR quantum for the object is not replenished, in order to avoid introducing the potential for request starvation.

4. A subsequent request for the object that is handled for pre-processing when the RR quantum has been reached, causes the object load descriptor to be reset, and is used for another cycle of request reordering that starts with the newly-handled request, at the end of the RPC servicing queue. Re-using object load descriptors is a safe act, as requests from previously-sorted per-object groupings can be serviced from PTLRPC service threads without further information provided by OBRR.

The following depicts a high-level representation of the OBRR load descriptor data structure:

```
struct obrr_load_desc {
...
cfs_spinlock_t ld_lock;
obd_id ld_oid;
cfs_hlist_node_t ld_list;
struct ptlrpc_request *ld_index;
nrs_rr ld_max_reqs;
nrs_rr ld_sorted_reqs;
...
}

struct ptlrpc_request {
...
+ void *req_nrs_data;
...
}
```

TODO: figure, was: Figure 1: Red-Black tree holding OBRR-handled RPCs for an object

TODO figure, was: Figure 2: Object load descriptors serviced in RR fashion

The obrr_state structure tracks the operational state of OBRR, including information regarding any currently handled RPCs; the RR quantum size can be specified in either I/O size or number of requests and is adjustable via a proc interface.

```
typedef __u64 nrs_rr;

struct obrr_state {
...
nrs_rr st_rr_quantum;
cfs_hash_t *st_object_hash;
enum nrs_policy_state st_state;
enum obrr_status st_status;
...
}
```

An increase in an RR quantum using the supplied proc interface takes effect immediately, whilst a decrease in the RR limit takes effect on the current request sorting cycle as long as the new RR

quantum does not specify a value that is smaller than the number of the currently sorted RPCs for the object; alternatively, the new quantum takes effect from the next sorting cycle for the object, i.e. a decrease in an RR quantum does not affect any currently sorted RPCs.

5.c. Client prioritization scheduling policy

Client prioritization could probably be implemented exclusively on OSS nodes to an extent, however some modifications would be required to the client nodes in order to provide with metrics to drive decisions upon which client prioritization could be tailored; an example of how this can be achieved can be seen in [2]. A similar method could be used to offer prioritization, whereby client nodes inform OSS nodes of the relative weighting of a given client. Algorithm UBVS (see relevant footnote in *section 5.b.*), or an adaptation of UBVS that replaces the per-user sets with per-client (export) sets, thus giving rise to a Client-Based Variable Slice (CBVS) algorithm could be used to implement this policy, whereby the size of the variable slice is a function of the client weightings.

5.1. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

[i.nrs.non-disruptive]: this requirement is fulfilled by allowing the core NRS framework to provide the ability to NRS scheduling algorithms to cater for fairness amongst requests, and for interacting with the high priority RPC queue; I/O access semantics relating to Partial Order Preservation (see [0]) are adhered to by virtue of the use of LDLM in Lustre.

[i.nrs.scheduler-various]: this requirement is met by allowing for different scheduling algorithms to operate that make use of the core NRS framework; if required, substituting algorithms at run-time can be driven by any external or internal stimulus, and can occur by moving assembled requests between sets in the data structures of different NRS algorithms.

[i.nrs.scheduler-improve-throughput]: the fair NRS scheduling algorithm aims to provide with increased overall throughput on each OSS, by selectively rearranging the order in which I/O requests will be dispatched, on a per-entity basis, and optionally, but preferentially, by taking into account the spatial locality of the file offsets of requests.

[i.nrs.scheduler-fair]: the fair NRS scheduling algorithm will cater for fairness amongst requests by **[a]** either treating requests in each per-entity set in a FIFO manner with respect to the request arrival times, **[b]** by bounding the degree to which it handles requests on a per-entity basis, **[c]** or by providing a dynamic deadline heuristic that will ensure expired requests are serviced close to their expiration time.

[i.nrs.scheduler-fair-scalability]: by making use of techniques that fulfil [r.nrs.scheduler-fair], and by placing requests in different sets on a per-entity basis and operating only on these small subsets

of the whole set of I/O requests per RR quantum, the algorithm should be equally efficient when operating in environments with a large number of clients, imposing a large number of requests, in a way that RPC rate at the server-side is not compromised with increasing server load. In case UBRR is used to implement the fair scheduling policy, making use of *unbounded* sorting behaviour and a dynamic deadline heuristic, the dynamic deadline value could also be a function of the overall load imposed by each client node user on the server (this is captured by [r.nrs.scheduler-unbounded-fair] in *section 5.4*, 'Refinement').

[i.nrs.scheduler-network-unfairness]: the core NRS framework could optionally cater for this requirement by taking into account request arrival times, during their transit from clients.

[i.nrs.scheduler-client-prioritization]: this requirement is met by the client prioritization scheduling policy, which is implemented by the UBVS or CBVS scheduling algorithms, optionally with modifications on the client nodes.

5.2. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, *etc*.) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

NRS does not have any dependencies that are not fulfilled by the existing filesystem components.

5.3. Security model

[The security model, if any, is described here.]

The core NRS framework and NRS scheduling algorithms will not implement any security-related operations, unless these aim to fulfil specific requirements that may relate to the problem domain a particular NRS policy applies against, and have not been foreseen in this document; the filesystem will continue to rely on currently supported security-related features.

5.4. Refinement

[This sub-section enumerates design level requirements introduced by the design. These requirements are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

[r.nrs.operations]: the core NRS framework should allow for scheduling algorithms to provide operations which can be used to intervene with the normal flow of requests in an OSS.

[r.nrs.obrr]: the fair NRS scheduling policy should use an OBRR or UBRR approach, unless there is indication that different algorithms would yield a more efficient or scalable implementation.

[r.nrs.threadless]: the core NRS framework should most probably be based on a thread-less implementation.

[r.nrs.scheduler-unbounded-fair]: if the fair NRS scheduler is to implement *unbounded* sorting behaviour, then fairness amongst requests should be catered for, preferably by means of a heuristic that would yield dynamic deadline values.

[r.nrs.scheduler-multiple]: the fair NRS scheduling policy implementation could allow for many of the different design aspects to be accommodated under a single scheduling algorithm instance, and come into effect on demand.

6. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behaviour from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

6.1. Core NRS Framework

6.1.a. States, events, transitions

An NRS scheduling policy can be in any of the following states:

- *ACTIVE*, denotes that the policy is being used to handle any newly arriving RPCs, and to optionally control the SECONDARY policy that will handle RPCs that the active policy does not intend to handle directly; only one policy is allowed to be in the ACTIVE state at any given point.
- *SECONDARY*, denotes that a policy in the ACTIVE state has denoted that this policy is to be used to handled any RPC types not supported by the active policy. It is anticipated that default scheduling behaviour presented in the form of an NRS scheduling algorithm will serve adequately as the secondary policy, for most policies expected to be implemented at this point. Only one policy is allowed to be in the SECONDARY state at any given point.
- *AVAILABLE*, denotes that the policy is in a state valid for transitioning to either ACTIVE or SECONDARY, but is not being used to handle any RPCs at the moment. More than one policies can be in the AVAILABLE state at any given point.
- *DISABLED*, denotes that although the policy has been registered with the NRS core, it is not in a state where it can be used legitimately, for some reason. More than one policies can be in the DISABLED state at any given point.

```
enum nrs_pol_svc_state {
NPS_INVALID,
NPS_ACTIVE,
NPS_SECONDARY,
NPS_AVAILABLE,
NPS_DISABLED,
NPS_NR
}
```

<State diagram to be included here optionally>

[This sub-section enumerates state machine states, input and output events and state transitions incurred by the events with a table or diagram of possible state transitions. UML state diagrams can be used here.]

6.1.b. State invariants

[This sub-section describes relations between parts of the state invariant through the state modifications.]

6.1.c. Concurrency control

[This sub-section describes what forms of concurrent access are possible and what forms on concurrency control (locking, queuing, *etc.*) are used to maintain consistency.]

The core NRS framework gives rise to concurrency control issues pertaining to the handling of the scheduling policies; structures representing different scheduling policies are kept in a lock-protected list.

TODO: Change : Policies transitioning from the OPERATIONAL to the PHASING_OUT state dispatch any already queued RPCs before transitioning to the AVAILABLE state; how dispatching of the RPCs occurs is determined by the policy in the PHASING_OUT state; the policy can for example either complete the handled RPCs itself, or delegate their handling to its secondary scheduling policy. The data structures keeping track of the state of scheduling policies are lock-protected; a linked list should suffice. Only one instance of each scheduling policy may be operational at a given point.

6.2. Fair scheduling policy (OBRR w/ limit on requests per object)

The following pertains to the use of OBRR with a hard limit on the number of RPCs that OBRR is allowed to handle for each object at any given point; this will help simplify algorithm design, before more ambitious implementations can be pursued.

6.2.a. States, events, transitions

If OBRR has an nrs_policy state != OPERATIONAL, it can not accept any incoming RPCs.

6.2.b. State invariants

6.2.c. Concurrency control

Concurency control issues that arise from the use of OBRR with a request number limit, are handled by use of existing locking primitives inside PTLRPC services; most of the work this scheduling algorithm performs will occur in code paths where current locks are held, namely cfs_spinlock_t::srv_lock and cfs_spinlock_t::srv_rq_lock; any internal to OBRR state will also be lock-protected if required. The aforementioned lock protection will be enough to serialise allocation and usage of object laod descriptors and handling of the object load descriptor hash table; OBRR does not introduce any new execution contexts, nor does it introduce the potential for additional races between existing executions contexts apart from the aforementioned.

7. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

7.1. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]

Scenario	[usecase.nrs.enqueue]
Relevant quality attributes	usability
Stimulus	an OST_READ/OST_WRITE request arrives via a PTLRPC service
Stimulus source	an application has made an I/O operation
Environment	normal filesystem operation

Artifact	a ptlrpc_request is handled by an ost_io OSS thread
Response	the ptlrpc_request is handled by the NRS scheduling algorithm enqueue operation
Response measure	• Scalability of the algorithm with regards to working set size • Optionally, depending on the DLD, the ordering of requests in per-entity sets according to offset
Questions and issues	Need to exactly find out the possible arrangement of pages that a bulk I/O request can refer to; can an I/O request pertain to disparate pages, belonging to different extents, and to what degree?

Scenario	[usecase.nrs.dequeue]
Relevant quality attributes	usability, scalability
Stimulus	an ost_io thread is handling the dissociation of requests from a per-entity set
Stimulus source	normal process scheduler operation
Environment	normal filesystem operation
Artifact	an ost_io OSS thread is scheduled
Response	the NRS dequeue operation of the scheduling algorithm dispatches a request
Response measure	• Selective dispatching of requests • Fairness amongst requests where applicable
Questions and issues	It is assumed that any I/O access semantics are maintained by observance of LDLM locks, and that NRS is free to sort amongst the whole set of queued requests. This query also pertains to [usecase.nrs.hp-pc] below.

Scenario	[usecase.nrs.hp-rpc]
Relevant quality attributes	usability, scalability, integrity
Stimulus	a ptlrpc_request is added to the high-priority rpc queue via a racing LDLM AST
Stimulus source	a contending LDLM lock has been requested
Environment	normal filesystem operation
Artifact	ptlrpc_hpreq_reorder_nolock() is called from the context of an ost_io OSS thread
Response	the NRS scheduling algorithm performs the necessary cleanup operations from its internal data structures

Response measure	• Integrity of OSS and NRS data structures is maintained • Assembled and queued RPC requests are conserved
Questions and issues	

[UML use case diagram can be used to describe a use case.]

7.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to Byzantine failures (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

8. Analysis

8.1. Scalability

[This sub-section describes how the component reacts to the variation in input and configuration parameters: number of nodes, threads, requests, locks, utilization of resources (processor cycles, network and storage bandwidth, caches), *etc*. Configuration and work-load parameters affecting component behavior must be specified here.]

8.2. Other

[As applicable, this sub-section analyses other aspects of the design, *e.g.*, recoverability of a distributed state consistency, concurrency control issues.]

8.2. Rationale

[This sub-section describes why particular design was selected; what alternatives (alternative designs and variations of the design) were considered and rejected.]

8.2.a Core NRS framework

It is assumed that varying workloads and environment characteristics will require different scheduling algorithms to be served best; it is preferable to implement a generic framework that will allow for different scheduling algorithms to be used.

Using entity-based fair scheduling algorithms allows for better scalability of the implementation, as the requests pertaining to an entity are expected to be a small subset of the overall request set.

8.2.b Fair scheduling policy (OBRR w/ limit on requests per object)

An RCU primitive could be used instead of a read/write spinlock; the RCU version may not perform well when an object experiences a large number of OST_WRITE RPCs while its RR quantum is being serviced, but will allow for multiple OST_WRITE RPCs to be added to the rbtree at the same time; some experimentation with different workloads may be necessary after the core NRS framework has been delivered.

9. Deployment

9.1. Compatibility

[Backward and forward compatibility issues are discussed here. Changes in system invariants (event ordering, failure modes, *etc.*)]

9.1.1. Network

9.1.2. Persistent storage

9.1.3. Core

Concerns have been expressed by project stakeholders that the prototype implementation in [3] may not be able to adequately cope with recovery-related operations in the filesystem, although the reasons this may hold are not clear; it would be prudent to ensure that the DLD does not disrupt the operation of any of the filesystem mechanisms.

[Interface changes. Changes to shared in-core data structures.]

9.2. Installation

[How the component is delivered and installed.]

10. References

[References to all external documents (specifications, architecture and requirements documents, *etc.*) are placed here. The rest of the document cites references from this section. Use Google Docs bookmarks to link to the references from the main text.]

[0] NRS Architectural Description from Oracle
[1] A Novel Network Request Scheduler for a Large Scale Storage System
[2] ETG Xyratex IRMOS whitepaper
[3] Server Side Request Scheduler, Oracle Bugzilla Bug 13634
[4] Morpheus Architecture

11. Inspection process data

[The tables below are filled in by design inspectors and reviewers. Measurements and defects are transferred to the appropriate project data-bases as necessary.]

Note: 'NRS' below is for 'T0.10PF.10pf.NRS' of Morpheus.

11.1. Logt

	Task	Phase	Part	Date	Planned time (min.)	Actual time (min.)	Comments
Nathan Rutman	NRS	HLDINSP					
Alexey Lyashkov	NRS	HLDINSP					
Andrew Perepechko	NRS	HLDINSP					

11.2. Logd

No.	Task	Summary	Reported by	Date reported	Comments
1	NRS				
2	NRS				
3	NRS				
4	NRS				
5	NRS				
6	NRS				
7	NRS				
8	NRS				
9	NRS				