High level design of a parity de-clustering algorithm

By Nikita Danilov <nikita.danilov@clusterstor.com>

Date: 2010/07/06
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 parity de-clustering algorithms and its demonstration program. The main purposes of this document are: (i) to be inspected by Mero architects and peer designers to ascertain that high level design is aligned with Mero architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of Mero customers, architects, designers and developers.

0. Introduction

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

This specification describes layout mapping functions of Mero parity de-clustered layouts. The purpose of such a function is to prescribe how file data are redundantly stored over a set of storage objects while maintaining desirable fault-tolerance characteristics.

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the Mero Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

This document includes by reference all definitions from the SNS Repair HLD [1]. Additionally,

- in the context of this document only layouts of a single storage pool, that will be called simply *the pool* are considered;
- number of target storage objects allocated in the pool for a layout will be denoted as P;
- number of data units in a parity group of a layout will be denoted as N;
- number of parity units in a parity group will be denoted as K (note that K is also equal to the number of spare units in a group);
- for a given $i \leq K$, union of i-th spare units from all parity groups in the pool is called an i-*th spare slot* (of the pool). A spare slot is used to repair a device failure;
- a *failure vector* is defined as an ordered collection of pool storage device identifies [D_1, D_2, ..., D_k], where D_i is an identify of a pool storage device failed at the moment and for $i \leq K$ repair of device D_i uses spare slot i. When no devices are failed, the failure vector is empty ([]). When the failure vector has more than K entries, the pool is dud (see [1]);

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.]

- [r.pdec.N+K]: a parity group width and a number of parity units in it can be specified for a parity de-clustered layout, where N and K are specified as pool parameters;
- [r.pdec.P]: a parity de-clustered layout with a specified parity group parameters can be implemented on top of a given number of target objects;
- [r.pdec.valid]: produced parity de-clustered layout valid in that it provides required degree of fault tolerance by never mapping two units of the same parity group to the same device;
- [r.pdec.unit-size]: size of data and parity units used by a layout are specifiable as layout parameters;
- [r.pdec.space.uniform]: a parity de-clustered layout consumes space on pool drives uniformly as object size grows infinitely;
- [r.pdec.repair.uniform]: an amount of data that has to be read from a storage device A to repair a layout when a storage device B fails is equal for all possible (A, B) pairs assuming sufficiently high percentage of storage drives is occupied;
- [r.pdec.degraded]: given a parity de-clustered layout and a failure vector it is possible to efficiently determine how a client should execute IO requests under given failures (*how*-s include a possible requirement to re-fetch the layout). Additionally, SNS repair process must be able to efficiently determine repair data-flow for a given layout;
- [r.pdec.formula]: parity de-clustered layouts can be efficiently generated from a layout formula by substituting parameters.

For the sake of completeness below is a standard list of requirements for a parity de-clustered layout as identified in [0]:

1. Single failure correcting. No two stripe units in the same parity stripe may reside on the same physical disk. This is the basic characteristic of any redundancy organization that recovers the data of failed disks. In arrays in which groups of disks have a common failure mode, such as power or data cabling, this criteria should be extended to prohibit the allocation of stripe units from one parity stripe to two or more disks sharing that common failure mode.
2. Distributed reconstruction. When any disk fails, its user workload should be evenly distributed across all other disks in the array. When replaced or repaired, its reconstruction workload should also be evenly distributed.
3. Distributed parity. Parity information should be evenly distributed across the array. Every data update causes a parity update, and so an uneven parity distribution would lead to imbalanced utilization (hot spots), since the disks with more parity would experience more load.
4. Efficient mapping. The functions mapping a file system's logical block address to physical disk addresses for the corresponding data units and parity stripes, and the appropriate inverse mappings, must be efficiently implementable; they should consume neither excessive computation nor memory resources.
5. Large write optimization. The allocation of contiguous user data to data units should correspond to the allocation of data units to parity stripes. This insures that whenever a user performs a write that is the size of the data portion of a parity stripe and starts on a parity stripe boundary, it is possible to execute the write without pre-reading the prior contents of any disk data, since the new parity unit depends only on the new data.
6. Maximal parallelism. A read of contiguous user data with size equal to a data unit times the number of disks in the array should induce a single data unit read on all disks in the array (while

requiring alignment only to a data unit boundary). This insures that maximum parallelism can be obtained.

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.]

Present parity de-clustered layout is based on combinatorial layouts (see [5]), as opposed to so called block design layouts ([2]). The key observation behind this decision is that while block design based layouts strictly conform to the uniformity requirements mentioned above under all circumstances, they are difficult to build and pose artificial constraints on admissible values of (N, K, P). Combinatorial layouts, on the other hand, are easy to build for any (N, K, P) (provided $N + 2 \cdot K \leq P$), at the expense of only guaranteeing uniformity asymptotically, *i.e.*, for a sufficiently large number of units and objects.

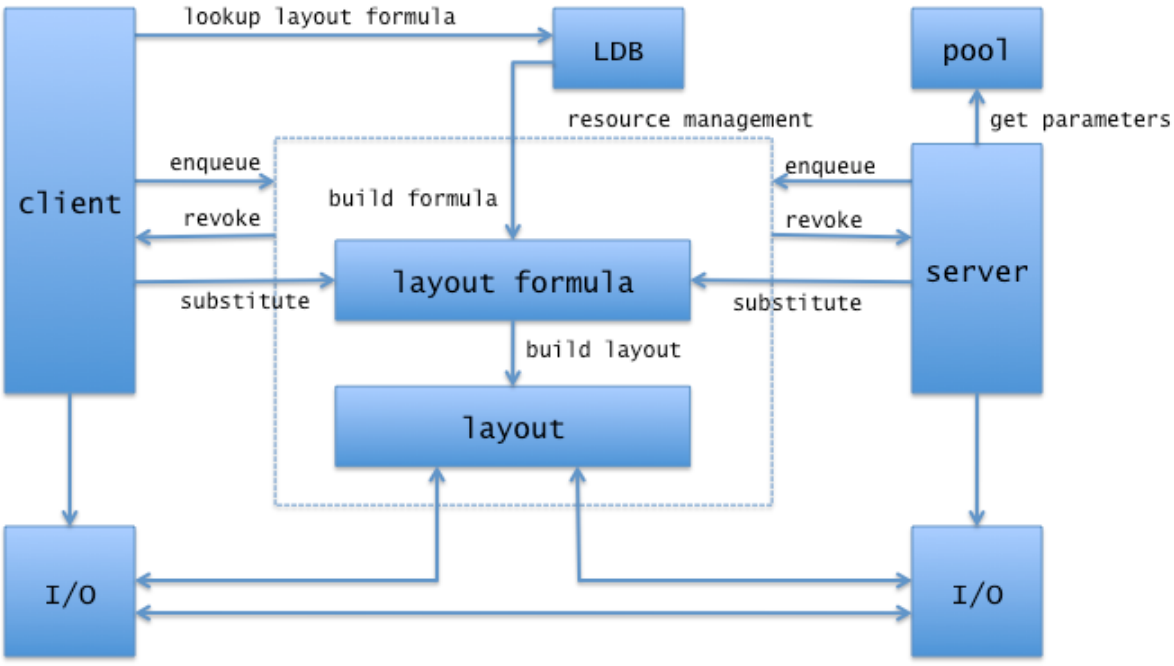
4. Functional specification

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

Three major external interfaces are described in this specification:

- a layout formula interface that generates a parity de-clustered layout by substituting parameters into a layout formula;
- a layout interface that plugs layout into client and server IO sub-systems, and
- a parity de-clustering demonstration program to analyze and simulate behavior of parity de-clustered layouts.

Interaction between components using first two interfaces is expressed by the following diagram:



A client fetches a layout from a Layout Data-Base (LDB) by sending a lookup query. LDB query typically returns a layout formula that can be instantiated into a layout by substituting parameters such as object fid and current pool failure vector to obtain a layout. Note that parameters are typically managed by the resource framework: cached on clients and revoked on resource usage conflicts. Layout formulae and layouts are resources too.

A file layout is used by a client to do IO against a file. The most basic layout IO interface is a mapping of file's logical namespace into storage objects' namespaces.

A server uses layouts for SNS repair.

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. Layout mapping function

The key question of a parity de-clustered layout design is a method for construction of a layout mapping function satisfying given requirements. Let's outline one such method, starting with an

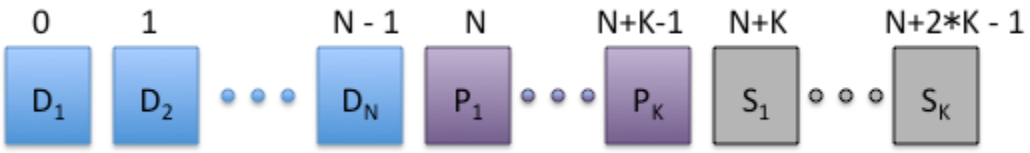
informal description. Precise definition of this method is given at the end of this sub-section.

First, some assumptions. A unit size is number of bytes in a data, parity or spare unit of a layout, denoted as U. For each layout the pool contains P objects (target objects). A target object is logically divided into *frames* of layout unit size. To specify a layout it is enough to specify how data, parity and spare units map to the target frames.

Some notation. For any natural number k, let S(k) denote the set {0, ..., k - 1}. Data units in a given parity group are indexed by S(N). Parity units and spare units are indexed by S(K). Target objects for a layout are indexed by S(P). Target frames in a target object are indexed by the set R = {0, 1, ...}. *Target address-space*, the set of all frames in target objects is $T = R \times S(P)$.

We shall use *g* to denote a parity group (number), *u* for a unit (number) in a parity group, *p* for a target object (number) and *r* for a frame (number).

Let's fix an enumeration of units within a parity group:



where D_i is a data unit, P_i is a parity unit and S_i is a spare unit. With this enumeration, units of a parity group can be identified with the set S(N + 2·K). Parity groups can be indexed by the set G = {0, 1, ...}

To specify a layout mapping for an object one has to present a function

$$f: G \times S(N + 2 \cdot K) \rightarrow T$$

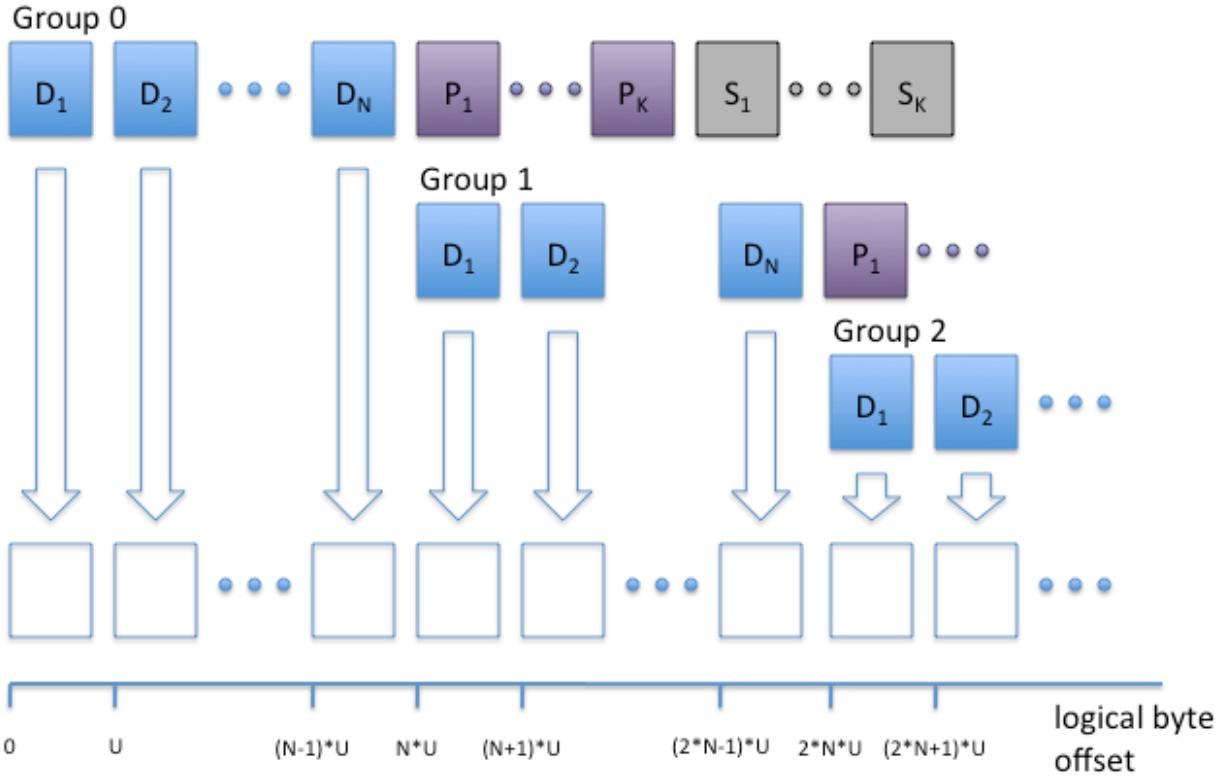
mapping pairs (parity-group-number, unit-number) to pairs (target-object-frame, target-object-number) and also a function

$$m: \mathbb{N} \rightarrow G \times S(N + 2 \cdot K) \times S(U)$$

mapping logical byte offset in the object to the (parity-group-number, unit-number, byte-offset-in-unit) triples. In all cases considered in this document,

```
m(offset) {
        group_size = U*(N+2*K);
        group = offset / group_size;
        unit = (offset - group*group_size) / U;
        unit_offset = offset - group*group_size - unit*U;
        return (group, unit, unit_offset);
}
```

That is, data units from consecutive parity groups are consecutive in the file:



A well-formed layout doesn't map different units to the same target unit, which means that function f has right inverse:

$$h{:}T \rightarrow G \times S(N + 2 \cdot K)$$

such that h(f(g, u)) = (g, u). An ability to efficiently calculate h is essential for SNS repair.

For example, standard RAID5 with K = 1, $P = N + 2$ and position of parity unit in a parity group g given by $s(g) = g\ (N + 1)$ (that is, forward parity rotation) has a layout function that can be defined as

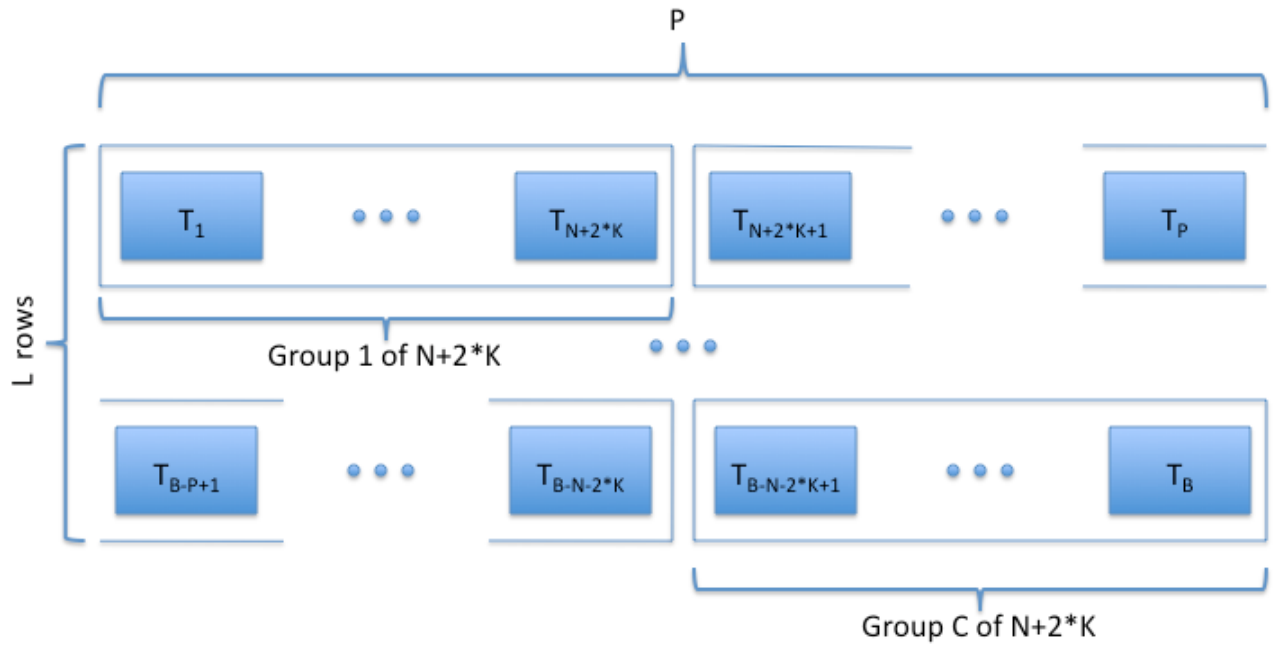
missing equation

Layout specified by this function can be illustrated by the following table:

group	pool device (total of N+2)								
	0	1	2	...	k	...	N - 1	N	N + 1
0	f(0, N)	f(0, N+1)	f(0, 0)	...	f(0, k - 2)	...	f(0, N-3)	f(0, N-2)	f(0, N-1)
1	f(1, 0)	f(1, N)	f(1, N+1)	...	f(1, k - 2)	...	f(1, N-3)	f(1, N-2)	f(1, N-1)
2	f(2, 0)	f(2, 1)	f(2, N)	...	f(2, k - 2)	...	f(2, N-3)	f(2, N-2)	f(2, N-1)
...									

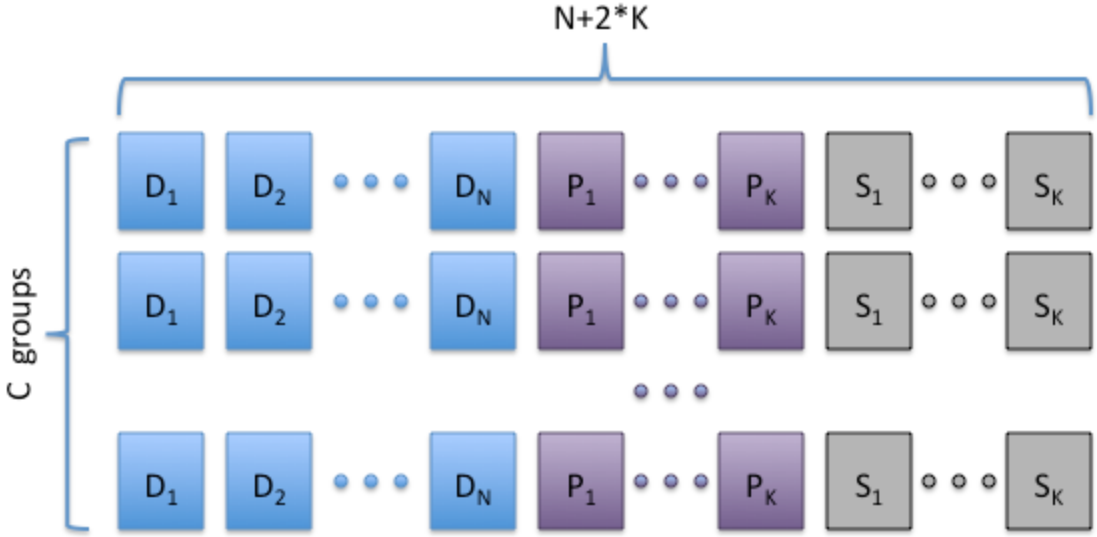
The parity de-clustered layout mapping is constructed in the following way:

- `select B that is multiple of N + 2·K and P. For example, B can be taken as the least common multiple of N + 2·K and P. Let L be B/P and C be B/(N + 2·K);`
- order address space $R \times S(P)$ lexicographically (such that (r, 0) is followed by (r, 1)) and split resulting sequence into *tiles* of B units;
- a tile can be seen as an $L \times P$ matrix with L rows, P columns each. Tile ω contains pairs (r, p) pairs with $r/L = \omega$:

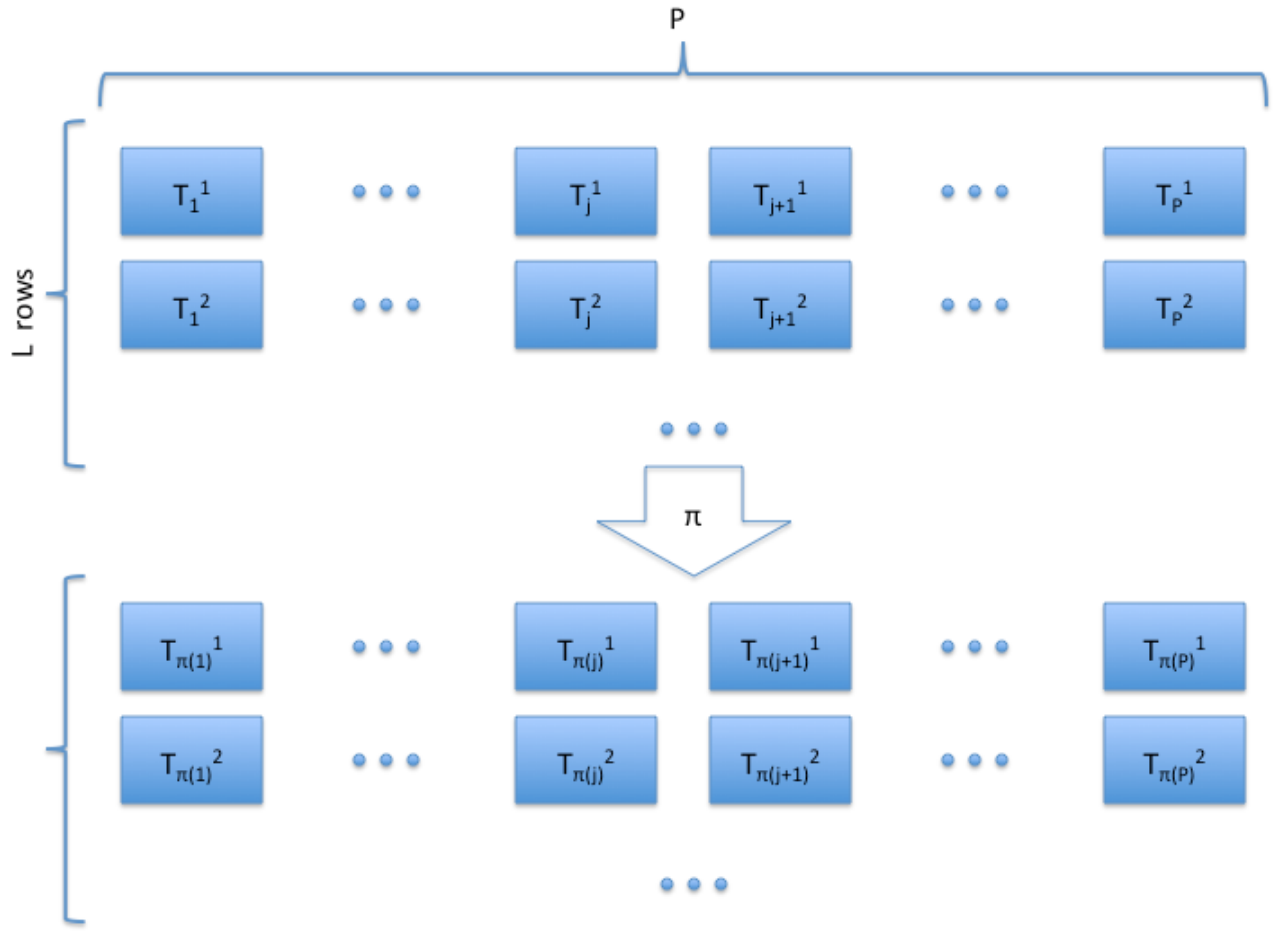


- alternatively a tile can be seen as a $C \times (N + 2 \cdot K)$ matrix (thanks to the selection of B);
- exactly C parity groups map to each tile. A parity group number g goes into a tile g/C. Tile number ω contains parity groups from $\omega \cdot C$ to $\omega \cdot C + C - 1$;
- considering a tile ω as a $C \times (N + 2 \cdot K)$ matrix, map units of each of C parity groups in the following way:
  - units of the group $\omega \cdot C + j$ map to j-th row of the matrix;
  - data units map to the first N target frames of the row;
  - parity units map to the next K frames in the row;

    - spare units map to the last K frames in the row;
- this process tiles the whole tile and produces valid layout fragment (in the sense that no two units of the same parity group are mapped to the same target object):



- select for this tile a (pseudo-random) permutation π of S(P);
- apply this permutation to each tile row:



- produced layout fragment is still valid (because the same permutation is applied to each row).

This construction can be precisely formalized with the help of two auxiliary families of function, which can be called "XDR functions for matrices":

- an "encoding" function $e_A: \mathbb{N} \times S(A) \to \mathbb{N}$, returning a number an element of a matrix with A columns and arbitrary number of rows has in the lexicographical ordering: $e_A(r, c) = c + A \cdot r$ and
- a "decoding" function $d_A: \mathbb{N} \to \mathbb{N} \times S(A)$, returning the row and column of the element of a matrix having a given number in the lexicographical ordering, $d_A(x) = (x/A, x \bmod A)$.

These functions are inverse of each other and with their help conversion between matrices of arbitrary shape can be easily made.

The last needed piece of data is a function assigning a permutation to each tile. Let's denote a permutation for a tile number ω as π_ω.

Now the parity de-clustering layout function can be implemented:

```
f(parity_group g, parity_unit u) {
          /* determine which tile the parity group p is in */
          (ω, j) = d_C(g);
         /* convert C × (N + 2 · K) matrix to L × P matrix */
          (r, t) = d_P(e_{N+2·K}(j, u));
          /* apply the permutation */
          t = π_ω(t);
          /* translate back from tile to target address space */
          r = e_L(ω, r);
          return (r, t);
}
```

The inverse function can be trivially constructed by replacing all steps in f's pseudo-code with their inverses:

```
h(frame r, target_object t) {
          (ω, r) = d_L(r);
          t = π_ω^{-1}(t);
          (j, u) = d_{N+2·K}(e_P(r, t));
         g = e_C(ω, j);
          return (g, u);
}
```

5.B. Permutation selection

To discharge [p.dec.repair.uniform] requirement a layout mapping function should shuffle tile columns so that on average the same number of parity groups contains units on a given pair of target objects. To this end columns of each tile should be re-ordered according to a permutation selected "randomly" for each tile. Let's first look at the magnitude of numbers involved in this process.

There are P! permutations of the set S(P). The following table shows number of bits in P! for various values of P:

P	$log_2\ (P!)$
10	22
20	61
57	256
100	524
300	2041
1000	8529
3000	30331

10000	118457

It's obvious that it is impractical, in general case, to assume that every layout has associated with it some kind of a "cookie" usable for selecting a permutation, because the size of this cookie would be prohibitively large even for modestly sized pools, given that systems with 100000's of storage devices are considered [4].

The solution is to use a relatively small sized (e.g., 64 or 128 bit) cookie as a layout formula parameter. For a given tile number, this cookie together with a tile number used to seed a fast pseudo-random number generator. The PRNG is then used to generate a sequence of numbers long enough to identify a tile permutation. The advantage of this approach is that a sequence thus produced can be interpreted as an encoding of a permutation lexicographic number in *factorial number system*, which can be used to quickly apply the permutation. With this approach only very small fraction of total permutations space is ever used by layout functions, but this fraction is scattered uniformly across the total space of all permutations (provided a PRNG with good properties is used), resulting in desired layout mapping function features.

5.C. Failure vector

Failure vector is an attribute associated with a layout and checked on every IO by the server. IO can continue only when failure vector supplied by a client matches current pool failure vector. On mismatch, the IO request is rejected and the client has to re-fetch the layout, including the new failure vector.

Given a failure vector and a mapping from layout target objects to the storage devices these objects are located on, a client is able to determine which of target object are failed and to which spare slots degraded mode IO requests should be redirected.

As SNS repair succeeds in re-constructing lost data, it changes pool failure vector, forcing layout updates on clients.

Note that in the current recovery strategy (NBA) clients never write to the pool while the latter is being repaired. All writes go to another pool, but failure vector still affects reads.

5.1. Conformance

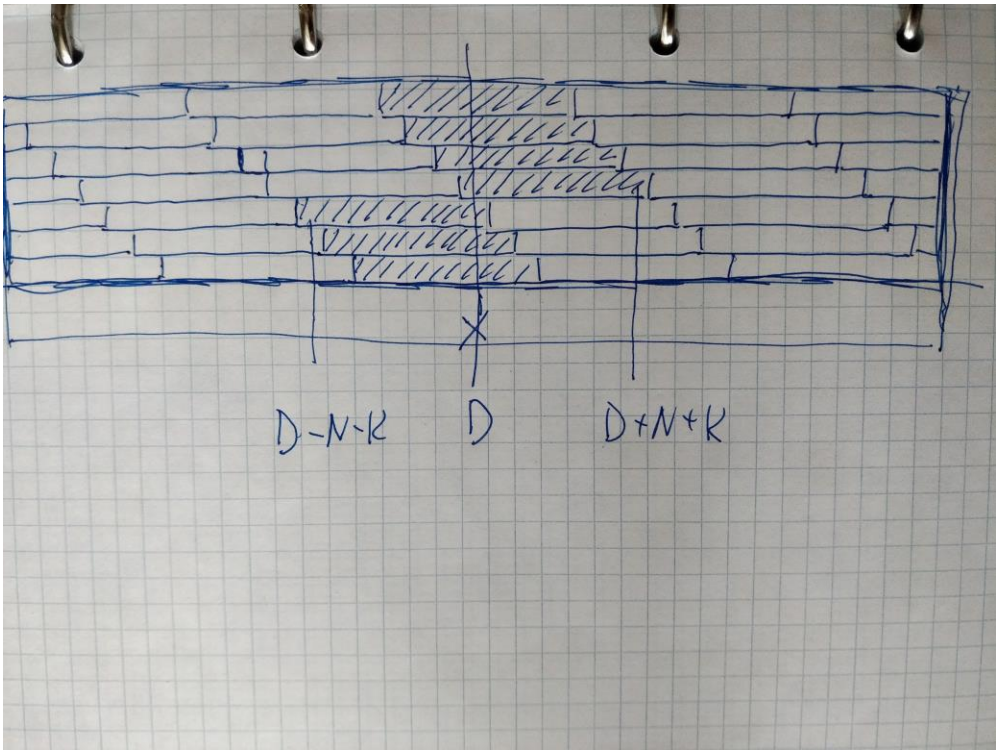
[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

- [r.pdec.N+K], [r.pdec.P]: by the construction of a layout mapping function above, any combination of N, K and P satisfying $N + 2 \cdot K \leq P$ is permitted;
- [r.pdec.valid]: one has to prove that from

it follows that $t\ t'$. Turning to the layout mapping function pseudo-code and observing that π_ω is injection, it is enough to prove that , which is obvious, because d_P is injection and $\in_{N+2\cdot K}$ is injection as a function of second argument.

- [r.pdec.unit-size]: layout mapping function maps units to units, any unit size can be used;
- [r.pdec.space.uniform]: described layout mapping function consumes tiles in target address space one by one. This means that difference between space consumption on different target objects can be at most L units. This guarantees that space consumption is uniform as file sizes grows infinitely. If, additionally, target objects are allocated on randomly selected storage devices, space consumption on pool storage devices would be uniform as number of files grows infinitely;
- [r.pdec.repair.uniform]: assuming that PRNG used to build tile column permutations is uniform, the distribution of columns in $L \times P$ mapped by permutations to a given pair of target objects is the same for all pairs of target objects. This means that the amount of data that has to be read from a target object O_1 to repair target object O_2 is equal for all possible (O_1, O_2). Together with the uniform distribution of target objects over pool storage devices this provides [r.pdec.repair.uniform];

Without permutations the situation would look like this when a disk D is failed:



So all the target objects needed to repair the failed one would be located nearby only. That

would make unacceptably not-uniform I/O load pattern for repair.

- [r.pdec.degraded]: on a client side, layout mapping function with failure vector efficiently compute the location of parity and spare units, guiding degraded mode IO. For SNS repair, layout mapping function (f) together with inverse layout mapping function (h) allow efficient enumeration of units of a parity group mapped to a given target frame, determining repait data-flow;
- [r.pdec.formula]: described parity de-clustered layouts can be generated from a "formula" (N, K, P, B) by substituting a per-file datum (*e.g.*, a file identifier) to seed a permutation function.

Conformance to the requirements from the literature:

1. Single failure correcting. Holds.
2. Distributed reconstruction. Holds asymptotically.
3. Distributed parity. Holds asymptotically
4. Efficient mapping. Holds.
5. Large write optimization. Holds.
6. Maximal parallelism. Holds at the tile granularity: if IO size is no less than a tile size, all pool storage devices can be utilized to serve the IO.

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.]

5.3. Security model

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

5.4. Refinement

[This sub-section enumerates design level requirements introduces 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.]

6. State

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

The layout mapping functions described by this specification have no side-effects are update no state. Maintenance of other state (layout formulae, layouts, failure vectors, *etc.*) is done by resource management and SNS repair sub-systems.

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.component.name]
Relevant quality attributes	[*e.g.*, fault tolerance, scalability, usability, re-usability]
Stimulus	[an incoming event that triggers the use case]
Stimulus source	[system or external world entity that caused the stimulus]
Environment	[part of the system involved in the scenario]
Artifact	[change to the system produced by the stimulus]
Response	[how the component responds to the system change]
Response measure	[qualitative and (preferably) quantitative measures of response that must be maintained]
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.]

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

[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] Mark Holland , Garth Gibson, *Parity Declustering for Continuous Operation in Redundant Disk Arrays*, 1992
[1] SNS Repair HLD
[2] Block design
[3] Factorial number system
[4] Mero Data Organization
[5] Thomas J. E. Schwarz , Jesse Steinberg , Walter A. Burkhard, *Permutation Data Layout (PDDL) Disk Array Declustering*

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.]

11.1. Logt

	Task	Phase	Part	Date	Planned time (min.)	Actual time (min.)	Comments
Jay	parity-declustering	HLDINSP	1		180		
		HLDINSP	1		180		

11.2. Logd

No.	Task	Summary	Reported by	Date reported	Comments
1	parity-declustering				
2	parity-declustering				
3	parity-declustering				
4	parity-declustering				