High Level Design of SNS Repair

By Huang Hua <hua.huang@clusterstor.com>,
Nikita Danilov <nikita.danilov@clusterstor.com>
Date: 2010/02/21
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 SNS repair of Mero M0. The main purposes of this document are: (i) to be inspected by M0 architects and peer designers to ascertain that high level design is aligned with M0 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 M0 customers, architects, designers and developers.

High level design of SNS repair
- 0. Introduction
- 1. Definitions
- 3. Design highlights
- 4. Functional specification
  - 4.0. Overview
  - 4.1. Failure type
  - 4.2. Redundancy level
  - 4.3. Triggers of SNS repair
  - 4.4. Control of SNS repair
  - 4.5. Concurrency & priority
  - 4.6. Client I/O during SNS repair
  - 4.7. Repair throttling
  - 4.8. Repair logging
  - 4.9. Device-oriented repair
  - 4.10. SNS repair and layout
- 5. Logical specification
  - Copy machine initialization
  - Copy machine operation
    - Persistent state
    - Resource management
    - Transactions
    - Concurrency control

storage-in agent A on a storage device D:
collecting agent A:
network-out agent A:
network-in agent A:
Pool machine
Server state machine
Scalable IO
5.1. Conformance
5.2. Dependencies
5.3. Security model
5.3.1. Network
5.3.2. Servers
5.3.3. Clients
5.3.4. Others
5.3.5. Issues
5.4. Refinement
6. State
6.1. Pool machine states, events, transitions
6.2. Pool machine state invariants
6.3. Server machine states, events, transitions
6.4. Server state machine invariants
6.5. Concurrency control
7. Use cases
7.1. Scenarios
7.2. Failures
8. Analysis
8.1. Scalability
8.2. Other
8.2. Rationale
9. Deployment
9.1. Compatibility
9.1.1. Network
9.1.2. Persistent storage
9.1.3. Core
9.2. Installation
10. References
11.1. Logt
11.2. Logd

0. Introduction

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

See HLD template document for important note on terminology used by the high level design specifications.

Redundant striping is a proved technology to achieve higher throughput and data availability. SNS (a.k.a Server Network Striping) applies this technology to networked devices, and achieves the similar goals as local RAID.

In the presence of storage and/or server failure, SNS repair will reconstruct lost data from surviving data reliably and quickly, without major impact to the product systems. This document will present the HLD of SNS repair.

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the M0 Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

Repair is a scalable mechanism to reconstruct data or meta-data in a redundant striping pool. Redundant striping stores a *cluster-wide object* in a collection of *components*, according to a particular *striping pattern*.

The following terms are used to discuss and describe Repair:

- *storage devices* are attached to *data servers*;
- *storage objects* provide access to storage device contents by means of a linear name-space associated with an object;
- some of the objects are *containers*, capable of storing other objects. Containers are organized into a hierarchy;
- a *cluster-wide object* is an array of bytes (the object's linear name-space, called *cluster-wide object data*) and accompanying meta-data, stored in containers and accessed through at least read and write operations (and potentially other operations of POSIX or similar interface). Index in this array is called an *offset*; A cluster-wide object can appear as a file if it is visible in file system namespace.
- a cluster-wide object is uniquely identified by an identifier, called a *fid*;
- a cluster-wide object *layout* [4], roughly speaking, is a map, determining where on the storage a particular element of its byte array is located. For the purposes of the present discussion only *striping layouts*, that will be called simply layouts, are needed. There are other layouts for encryption, deduplication, *etc.* that look quite differently. In addition to the cluster-wide object data, a cluster-wide object layout specifies where and how redundancy information is stored;
- a cluster-wide object layout specifies a location of its data or redundancy information as a pair (component-id, component-offset), where component-id is a fid of a *component* that is stored in a certain container (Layouts introduce constraints on fids of components of a given cluster-wide object, but these are not important for the present specification.);
- a component is an object, and, like a cluster-wide object, an array of bytes (*component data*) identified by a (component) offset plus some meta-data;
- a component has *allocation data*—a meta-data specifying where in the container component data are stored;
- cluster-wide object layouts, considered in this document, are piecewise linear mappings in the sense that for every layout there exists a positive integer S, called a (layout) *striping unit size*, such that the layout maps any extent `[p*S, (p + 1)*S - 1]` onto some `[q*S, (q + 1)*S - 1]` extent of target, which is some component. Such an extent in a target is called a *striping unit* of the layout. The layout mapping within a striping unit is increasing (this characterizes it uniquely);

- in addition to placing object data, striping layouts define the contents and placement of *redundancy information*, used to recreate lost data. Simplest example of redundancy information is given by RAID5 parity. In general, redundancy information is some form of check-sum over data;
- a striping unit to which cluster-wide object or component data are mapped is called a *data unit*;
- a striping unit to which redundancy information is mapped is called a *parity unit* (this standard term will be used even though redundancy information might be something different than parity);
- A striping layout belongs to a *striping pattern* (N+K)/G if it stores K parity units with redundancy information for every N data units and units are stored in G containers. Typically G is equal to the number of storage devices in the pool. Where G is not important or clear from the context, one talks about N+K striping pattern (which coincides with the standard RAID terminology);
- a *parity group* is a collection of data units and their parity units. We only consider layouts where data units of a parity group are contiguous in the source. We do consider layouts where units of a parity group are not contiguous in the target (parity declustering). Layouts of N+K pattern allow data in a parity group to be reconstructed when no more than K units of the parity group are missing;
- for completeness, it should be mentioned that meta-data objects have associated layouts, not considered by this specification. Meta-data object layouts are described in terms of meta-data *keys* (rather than byte offsets) and also based on redundancy, typically in the form of mirroring. Replicas in such mirrors are not necessary (and usually aren't) byte-wise copies of each other, but they are key-wise copies;
- components of a given cluster-wide object are normally located on the servers of the same pool. Sometimes, *e.g.*, during migration, a cluster-wide object can have a more complex layout with components scattered across multiple pools;
- a layout is said *to intersect* (at the moment) with a storage device or a data server if it maps any data to the device or to any device currently attached to the server, respectively;
- a *pool* is a collection of storage, communication and computational resources (server nodes, storage devices and network interconnects) configured to provide IO services with certain fault-tolerance characteristics. Specifically, cluster-wide objects are stored in the pool with striping layouts with such striping patterns that guarantee that data are accessible after a certain number of server and storage device failures. Additionally, pools guarantee (by means of SNS repair described in this specification) that a failure is repaired in a certain time.

Examples of striping patterns:

- K = 0. RAID0, striping without redundancy;
- N = K = 1. RAID1, mirroring;
- K = 1 < N. RAID5;
- K = 2 < N. RAID6.

A cluster-wide object layout owns its target components. That is, no two cluster-wide objects store data or redundancy information in the same component object.

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 an d 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.]
These requirements are already listed in the SNS repair requirement analysis document [1]:

- [r.sns.repair.triggers] Repair can be triggered by a storage, network or node failure.
- [r.sns.repair.code-reuse] Repair of a network array and of a local array is done by the same code.
- [r.sns.repair.layout-update] Repair changes layouts as data are reconstructed.
- [r.sns.repair.client-io] Client IO continues as repair is going on.
- [r.sns.repair.io-copy-reuse] The following points of variability are implemented as policy applied to the same underlying IO copying and re-structuring mechanism:
    - do client writes target the same data that are being repaired or client writes are directed elsewhere?
    - does repair reconstruct only a sub-set of data (*e.g.*, data missing due to a failure) or all data in the array?

  That is, the following use cases are covered by the same IO restructuring mechanism:

	same layout	**separate layouts**
missing data	in-place repair	NBA
all data	migration, replication	snapshot taking

Here "same layout" means that client IO continues to the source layouts while data restructuring is in progress and "separate layout" means that client IO is re-directed to a new layout at the moment when data restructuring starts.

"Missing" data means that only a portion of source data is copied into a target and "all data" means that all data in the source layouts are copied.

While data restructuring is in progress affected objects have composite layouts showing what parts of object linear name-space have already been re-structured. Due to the possibly on-going client IO against an object, such a composite layout can have a structure more complex than "old layout up to a certain point, new layout after".

- [r.sns.repair.priority] Containers can be assigned a *repair priority* specifying in what order they are to be repaired. This allows to restore critical cluster-wide objects (meta-data indices, cluster configuration data-base, *etc.*) quickly, decreasing the damage of a potential double failure.
- [r.sns.repair.degraded] Pool state machine is in degraded mode during repair. Individual layouts are moved out of degraded mode as they are reconstructed.
- [r.sns.repair.c4] Repair is controllable by an advanced C4 settings: can be paused, aborted, its IO priority can be changed. Repair reports its progress to C4.
- [r.sns.repair.addb] Repair should produce ADDB records of its actions.
- [r.sns.repair.device-oriented] Repair uses device-oriented repair algorithm, as described in [3].
- [r.sns.repair.failure.transient] Repair survives transient node and network failures.
- [r.sns.repair.failure.permanent] Repair handles permanent failures gracefully.
- [r.sns.repair.used-only] Repair should not reconstruct unused (free) parts of failed storage.

There are also some overall requirements from the Summary Requirement Table [2], but none of them are relevant to SNS repair.

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 design structures SNS repair implementation as a composition of two sub-systems: a generic data restructuring engine (*a copy machine*, described in the logical specification) and an SNS repair specific part.

A copy machine is a scalable distributed mechanism to restructure data in multiple ways (copying, moving, re-striping, reconstructing, encrypting, compressing, re-integrating, *etc.*) that can be used in variety of scenarios, some enumerated in the following text. An SNS specific part of repair interacts with sources of repair relevant events (failures, recoveries, administrative commands, client IO requests), constructs a copy machine suitable for SNS repair and controls its execution.

Following topics deserve attention:

- all issues and questions mentioned in the requirements analysis document [1] must be addressed;
- pool state machine must be specified precisely;
- repair state machine must be specified precisely;
- handling of transient and permanent failures during repair must be specified precisely;
- interaction between repair and layouts must be specified;
- definitions must be made precise;
- details of iteration over objects must be specified;
- details of interaction between repair and DTM must be specified;
- redundancy other than N+1 (N+K, K > 1) must be regarded as a default configuration;
- multiple failures and repair in the presence of multiple failures must be considered systematically;
- repair and re-balancing must be clearly distinguished;
- reclaim of a distributed spare space must be addressed (this is done in a separate Distributed Spare design documentation);
- locking optimizations.

4. Functional specification

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

4.0. Overview

When a failure is detected and the system decides to do SNS repair, the repair is able to concurrently read data from multiple storage devices, aggregate them, transfer them over the network and place them into distributed spare space. The whole process can utilize the system resources with full bandwidth. If another failure happens during this process, it is reconfigured with new parameters and starts repair again, or fails gracefully.

4.1. Failure type

- *Transient failure*. Transient failure includes a short network partition or a node crash followed by reboot quickly enough. Formally, a transient failure is a failure that was healed before the system decided to declare the failure permanent. Transient network failures are handled transparently by the rpc and networking layer (resend). Transient node failures are handled by the DTM (recovery). Data or meta-data stored on media drive is not damaged.
- *Permanent failure*. Permanent failure means permanent damage to media drives, and there is no way to recover the data physically from the drive. Data will have to be reconstructed from redundant information living in surviving drives, or restored from archival backups.
- For SNS repair purposes, we only talk about permanent failure of storage devices or nodes. The C4 and/or SNS repair manager can distinguish the two types of failures from each other.
- Failure detections will be done by various components, e.g. liveness.

4.2. Redundancy level

- A pool using N+K striping pattern can recover from at most K drives failures. System can reconstruct lost units from the surviving unit. K can be selected so that a pool can recover from a given number K_d or device failures and a given number K_s of server failures (assuming uniform distribution of units across servers).
- The default configuration will always have $K > 1$ (and $L > 1$) to insure the system can tolerate multiple failure.
- More detailed discussion on this can be found at: Reliability Calculations and Redundancy Level. and in the Scalability analysis section below.

4.3. Triggers of SNS repair

- When a failure of storage, network or node is detected by various components(e.g. liveness layer), it will be reported to components which are interested in the failure, including pool machine and C4. Pool machine will decide whether to trigger a SNS repair.
- Multiple SNS repairs can be running simultaneously.

4.4. Control of SNS repair

- Running and queued SNS repair can be listed upon query by management tools.
- Status of individual SNS repair can be retrieved upon query by management tools: estimated progress, estimated size, estimated time left, queued or running or completed, etc.
- Individual SNS repair can be paused/resumed.

- A fraction of resource usage can be set to individual repair by management tools. These resources include disk bandwidth, network bandwidth, memory, CPU usage and others. System has a default value when SNS repair initiated. This value can be changed dynamically by management tools.
- Resource usage will be reported and collected at some rate. These information will be used to guide the future repair activities.
- Status report will be trapped asynchronously to C4 while a repair is started, completed, or failed, or progressed.

4.5. Concurrency & priority

- To guarantee that sufficient fraction of system resource are used, we (i) guarantee that only a single repair can go on a given server pool and (ii) different pools do not compete for resources.
- Every container has a repair priority. A repair for failed container has the priority derived from the container.

4.6. Client I/O during SNS repair

- From client's point of view, the client I/O will be served while SNS repair is going on. Some performance degradation may be experienced, but this should not lead to starvation or indefinite delays.
- Client I/O to surviving containers or servers will be handled normally. But SNS repair agent will also read from or write to the containers while SNS repair is going on.
- Client I/O to failed container (or failed server) will be directed to proper container according to the new layout, or data will be served by retrieving from other containers and computing from parity/data unit. This depends on implementation options. we will discuss this later.
- When repair is completed, client I/O will restore to its normal performance.

4.7. Repair throttling

- SNS manager can throttle the repair according to system bandwidth, user control. This is done by dynamically changing the fraction of resource usage of individual repair or overall.

4.8. Repair logging

- SNS repair will produce ADDB records about its operations and progress. These records include, but not limited to, {start, pause, resume, complete} of individual repair, failure of individual repair, progress of individual repair, throughput of individual repair, etc.

4.9. Device-oriented repair

- Agent iterates components over the affected container, or all containers which have surviving data/parity unit in the need-to-reconstruct parity group. These data/parity unit will be read and

sent to proper agent where spare space lives, and used to re-compute the lost data.

4.10. SNS repair and layout

- SNS manager gets an input set configuration and output set configuration as the repair initiated. These input/output set can be described by some form of layout. SNS repair will read data/parity from the devices described with the input set and reconstruct missing data. In the process of reconstruction object layouts affected by the data reconstruction (i.e., layouts with data located on the lost storage device or node) are transactionally updated to reflect changed data placement. Additionally, while the reconstruction is in progress, all affected layouts are switched into a degraded mode so that clients can continue to access and modify data.

  Note that the standard mode of operation is a so called "*non-blocking availability*" (NBA) where after a failure a client can immediately continue writing new data without any IO degradation. To this end a client is handed out a new layout where it can write to. A cluster-wide object has, after this point, a composite layout: some parts of object linear name-space are laid accordingly to the old layout and other parts (ones where clients write to after a failure)—a new one. In this configuration, clients never write to the old layout, while its content is being reconstructed.

  The situation where there is a client-originated IO against layouts being reconstructed is possible because of

  - reads have to access old data even under NBA policy and
  - non-repair reconstructions like migration or replication.

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.]

The main sub-components of SNS repair component are:

- *pool machine*: a replicated state machine [5] responsible for organising IO with a pool. External events such as failures, liveness layer and management tool call-backs incur state transitions in a pool machine. A pool machine, in turn, interacts with entities such as layout manager and layout IO engines to control pool IO. A pool machine uses quorum based consensus to function in the face of failures;

- *copy machine*: a replicated state machine [5] responsible for carrying out a particular instance of a data restructuring. A copy machine is used to move, duplicate or reconstruct data for various scenarios including

  - SNS repair;
  - replication;

    - migration;
    - snapshotting;
    - backup;
    - proxy reintegration;
    - cache reintegration, including client-server cached IO and container migration.

  In the case of an SNS repair, a copy machine is created when a pool machine transitions into degraded state. A copy machine creates an ensemble of storage, network and collecting agents that collaborate to reconstruct pool data. Agent completion events cause copy machine to interact with the layout manager to indicate that data have been reconstructed. Client IO forces copy machine to reconstruct data out of order to serve client requests. On additional failures pool machine reconfigures the copy machine to continue the repair. Eventually, copy machine transfers its parent pool into either normal or failed mode and self-destructs;

- *storage-in agent*: a state machine, associated with a storage device. A storage-in agent, created as part of a copy machine, asynchronously reads data from the device and directs them to a collecting, network-out or storage-out agent. A storage-in agent is responsible for limiting the fraction of storage bandwidth consumed by a copy machine and for acquiring DLM locks. A storage-in agent executes a loop, described below, iterating over containers stored on the device and over component objects stored in the containers;

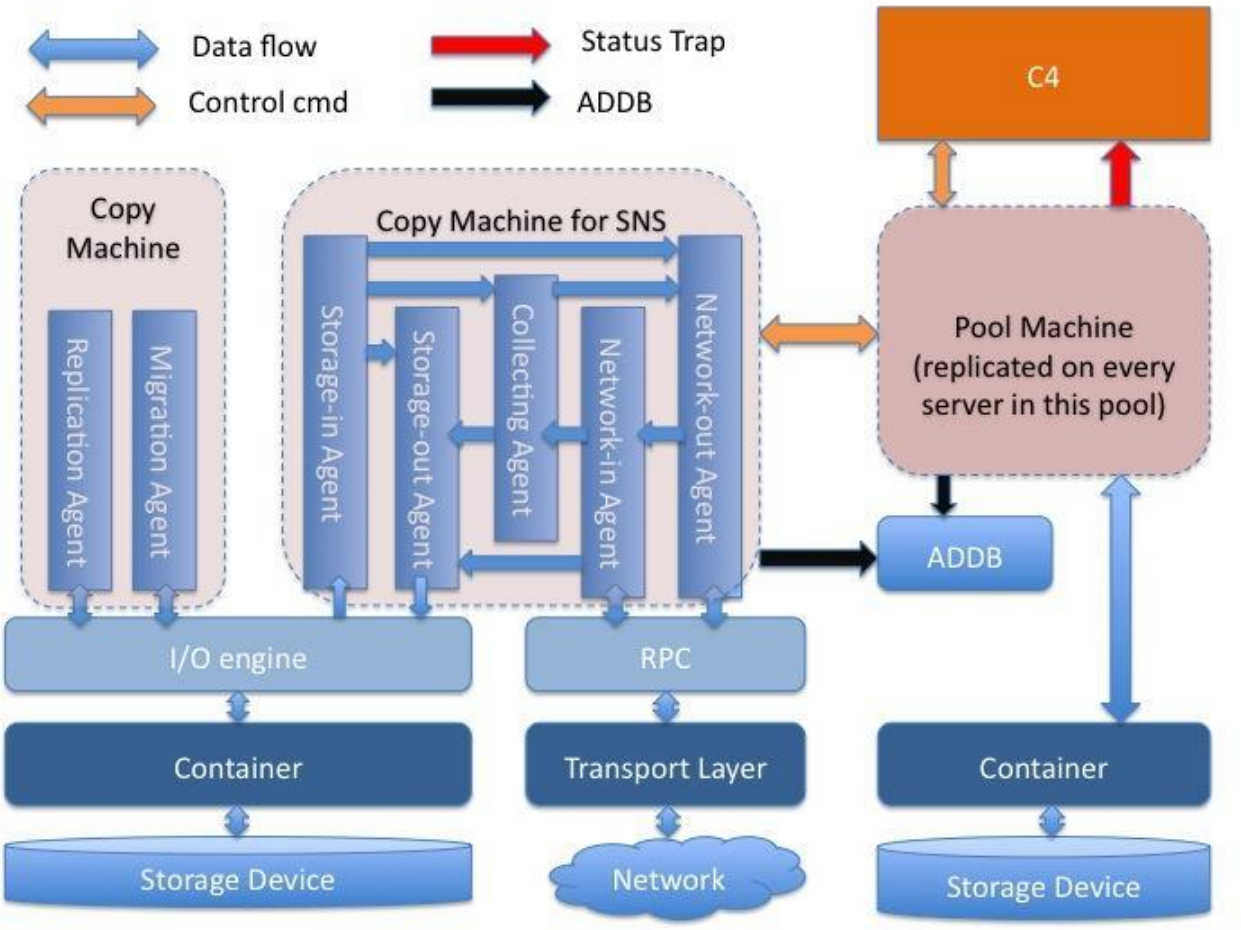
- *storage-out agent*: a state machine, associated with a storage device. A storage-out agent, created as part of a copy machine, collects data from other agents (storage-in, network-in or collecting ones) and submits them to the storage. A storage-out agent is responsible for limiting the fraction of storage bandwidth consumed by a copy machine;

- *network-in agent*: a state machine, associated with a network interface. A network-in agent, created as part of a copy machine, asynchronously reads data, sent by other agents across the network and directs them to a collecting or storage-out agent. A network-in agent is responsible for limiting the fraction of network bandwidth consumed by a copy machine;

- *network-out agent*: a state machine, associated with a network interface. A network-out agent, created as part of a copy machine, collects data from other agents (storage-in or collecting ones) and asynchronously submits them to the RPC layer. A network-out agent is responsible for limiting the fraction of network bandwidth consumed by a copy machine;

- *collecting agent*: a state machine, created as part of a copy machine, typically associated with a server node, collecting data from other agents, reconstructing data by using transformation function associated with the copy machine and forwarding data to other agents. A collecting agent is responsible for limiting the fraction of processor used by a copy machine;
- *copy machine buffer pool*: a buffer pool limiting the amount of memory copy machine consumes on a given node.

The following diagram depicts these sub-components of SNS repair:



Various configurations of agents are possible. In the simplest case, storage-in and storage-out agents are located in the same address space and directly exchange data through a pool of shared buffers (this is a scenario of local data duplication, *e.g.*, for the purpose of a snapshot creation). Next in complexity is a configuration where storage-in agents, associated with a storage devices on a node, deliver data into buffers allocated by a network-out agent running on the node. The latter sends data to a set of network-in agents across the network. The network-in agent forwards data to a collecting agent, running on the same node. This collecting agent, collects and optionally reconstructs the data, and sends them to the storage-out agent running on the same node. This is a scenario of a network repair. More complex arrangements with multiple layers of collecting are useful for re-integrating large persistent caches of proxy clusters or spreading data reconstruction across multiple processors.

It is assumed that all state machines are endowed with a common set of features that won't be specifically mentioned:

- integration with ADDB[1]. State transitions and distribution of times spent in particular states are logged to the ADDB;

[1][u.machine.addb]

- persistency[2]. State machines use services of local and distributed transaction managers to recover from node failures. After a restart, persistent state machine receives a *restart event*, that it can use to recover its lost volatile state;
- discoverability[3]. State machines can be discovered by the management tool;
- queuing[4]. A state machine has a queue of incoming requests.

Copy machine initialization

A copy machine parameters are:

- an input set description. An input set consists of data and meta-data that the copy machine has to re-structure. Examples of input set description are:

    - data in a given container;
    - data in all cluster-wide objects, having a component in a given container;
    - data on a given storage device;
    - data in all cluster-wide objects, having a component in a container on a given storage device;
    - data on a given data server;
    - data in all cluster-wide objects, having a component in a container residing on a given data server;
    - data in a client or proxy volatile or persistent cache;
    - data from the operation records in a given segment of FOL;
    - data from all files in a given file-set.

  Examples above are for *data* input sets specifying sources of data reconstruction process. Similarly, meta-data reconstruction uses copy machine operating on a meta-data input set. Meta-data input set describes a collection of meta-data in one or more meta-data containers:

    - a collection of meta-data records in a certain meta-data table;
    - a collection of meta-data records stored on a certain storage device;
    - a collection of meta-data records pertaining to operations on objects in a given file-set;
    - a collection of meta-data records in a certain segment of FOL, *etc.*

  Present specification deals mostly with data reconstruction.
  It is possible, for a given input set description, to efficiently estimate at what storage devices and at what servers the data from the set are located at the moment. A copy machine uses this information to start agents. The estimation must be conservative: it must include all the servers and devices where data from the set are, but may also include some other servers and devices. The assumption here is that estimation can be made simpler and cheaper at the expense of creating extra agents that will do nothing. The "at the moment" qualification of estimation is for the possibility of container migration. There are multiple possible strategies

[2][u.machine.persistency]

[3][u.machine.discoverability]

[4][u.machine.queuing]

to deal with the latter:

- "lock" the container as part of estimation, so that migration is not possible while a copy machine uses the container;
- implement container migration as a 2-phase protocol[5] where container users (in this case a copy machine) are first asked a permission to move a container;
- implement agents as part of container, migrated together with the later. When migration of a container completes, all its agents get *migration event*[6]. This is basically the same solution as the previous one, but implemented as part of a container framework.

In the case where a container is migrated by physically moving a storage device around, migration cannot be prevented and any solution of migration problem must deal with this (possibly by failing the copy machine).

It is easy to see that an input set description is, essentially, a layout [4] of some kind. Its difference from a more typical file layout is that a file layout domain is a linear file offset name-space, whereas input set layout domain can be more complex: it addresses contents of multiple objects and also their redundancy information.

Input set specifies how data can be found in a certain container or a set of containers. In a typical case of SNS repair, input set specifies how data can be found in the device container (*i.e.*, on a storage device). Other possibilities include input set referring to data containers or objects. These cases capture the cases where input data are compressed, encrypted or de-duplicated;

- priority assignment. A priority is assigned to every container in the input set, determining the order of restructuring;

- an output set description. Similarly to the input set description, an output set description indicates where re-structured data will be stored at. Examples of output set descriptions are:

  - a container;
  - a distributed spare space of a pool;
  - a file or file-set;

  All the notes about input set descriptions apply to the output set descriptions.

- an aggregation function. An aggregation function maps input set description layout onto output set description layout domain. Examples of possible aggregation functions are:

  - identity function (for replication, migration, backup);
  - map striping units of a striping group in the input set layout to a striping unit in the output set layout (for repair);
  - map a byte extent of a file in a client data cache onto an equally sized extent in one of the file's component objects;
  - function mapping a block of input set data to its hash (de-duplication).

[5][u.container.migration.vote]

[6][u.container.migration.call-back]

Aggregation function specifies which parts of input set have to be accumulated before output data can be produced. Hence, aggregation function defines the restructuring data-flow and graph of agent-to-agent connections. A set of input parts that aggregation function maps to the same place in the output set is called *an aggregation group*. For example, striping units of the same parity group form aggregation group for repair, all blocks sharing the same de-duplication hash value form aggregation group for de-duplication;

- a transformation function. A transformation function changes aggregated data or meta-data before they are placed into output set. Examples of possible transformation functions are:

    - identity function (for replication, migration, backup);
    - XOR or higher Reed-Solomon codes for N+K striping layout repair;
    - compression or encryption algorithm.

    Transformation function produces output data from a collection of all input data that aggregate function accumulates at a given point of output set layout domain. All transformations that we consider can be calculated by applying some function to each new portion of input set as it arrives, that is, it is never necessary to buffer all aggregated input data before calculating output.

    An important optimisation is an opportunistic aggregation and transformation of data. For example, when a parity group having multiple units on the same server is reconstructed, XOR of these units can be calculated by the collecting agent on the server, without sending individual units over the network.
- resource consumption thresholds:

    - memory;
    - processor cycles;
    - storage bandwidth;
    - network bandwidth;

    Resource thresholds can be adjusted dynamically.
- a set of call-backs that copy machine calls at certain conditions:

    - progress notifications:
        - enter: called when the copy machine starts re-structuring a part of its input set;
        - leave: called when the copy machine has completed re-structuring a part of its input set.

            A copy machine invokes notification call-backs at different granularities: when it starts or stops processing

            - the whole input set;
            - on a server;
            - on a storage device;
            - in a container;
            - a layout;
            - an aggregation group.

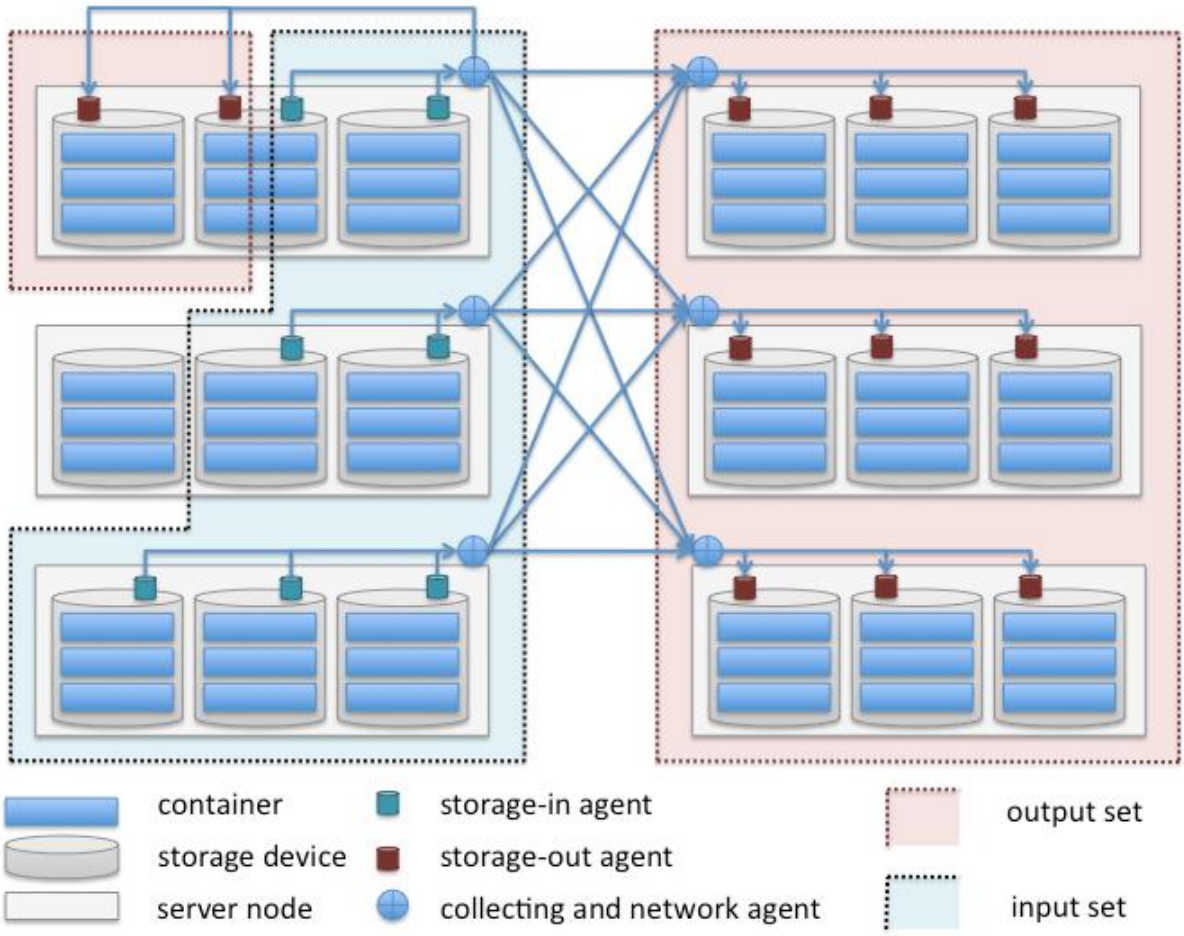


Figure: a copy machine data-flow

Copy machine operation

To describe the operation of a copy machine, the following definitions are convenient:

- *copy packet*: a chunk of input data traversing through the copy machine. A copy packet is created by a storage-in agent or a collecting agent and encapsulates information necessary to route the data through the machine's agents;
- *next-agent(packet, agent)*: a function returning for a copy packet identity of the agent that has to process this packet next after the given agent. next-agent function determines the routing of copy packets through the copy machine. next-agent function is determined by the aggregation function;
- *aggregation-group(P)* is an aggregation group to which packet P belongs;
- *queue(P, A)* is a function adding copy packet P to the incoming queue of next-agent(P, A), assuming that this queue is local, *i.e.*, that A and next-agent(P, A) are located on the same node.

The figure below shows a typical route of a copy packet P through the copy machine:

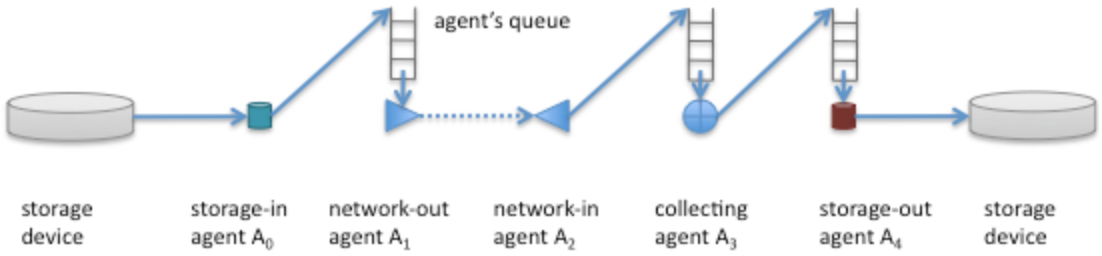


Figure: a possible route of a copy packet through a copy machine.

On this diagram, next-agent(P, A_i) = A_{i+1}. In the following only configurations where every pipe-line starts with a storage-in and ends in a storage-our agent are considered. This can be easily generalized.

Persistent state

A copy machine persistent state records processed parts of input set. In a typical case where input set description is defined as a meta-data iterator[7] of some kind (like it is in all the examples given above), persistent copy machine state consists of extents in the iterator position space[8]. Multiple disjoint processed extents can appear in input set because of the out-of-order processing of user IO requests. A natural persistent state structure is a B-tree of such extents[9], keyed by starting position of an extent, with merge on insertion and split on deletion done as part of transaction undo. *Note*: or maybe without merging to avoid the complications of undo action failing. *End note.*

Each storage device, participating in data restructuring, holds its own copy of copy machine persistent state. This provides a natural level of copy machine fault tolerance, assuring that restructuring can proceed as long as data to restructure are available.

Resource management

A copy machine deals with distributed resource consumption problem. For example, limitations on a fraction of storage device throughput that the copy machine can consume constrain the rate at which copy packets can be sent to the node. Such constraints are addressed by the generic M0 resource management infrastructure[10]. Server nodes declare[11] resources (memory, disk bandwidth, processor

[7][u.md.iterator]

[8][u.md.iterator.position]

[9][u.TREE.SCHEMA.ADD] ST

[10][u.resource.generic]

[11][u.resource.declare]

bandwidth) allocated for the particular copy machine instance. Other nodes grab[12] portions of declared resources and cache[13] resources to submit copy packets.

Note: on top of resource management infrastructure a more sophisticated queuing mechanism can be build. Define *a packet P pipe-line from agent A* as a sequence of agents A, next-agent(P, A), next-agent(P, next-agent(P, A)), *etc.* until a storage-out agent is reached. For every possible packet pipe-line starting at A, an estimation of the pipe-line throughput can be calculated, based on the local resource limits and estimations obtained through the resource management infrastructure. This throughput can then be used to globally optimize resource consumption by throttling a packet processing at A based on pipe-line the packet is to follow. *End note.*

Transactions

Restructuring of an aggregation group must be a distributed transaction. To achieve this, a copy packet P is tagged with a distributed transaction identifier which is a function of aggregation-group(P), so that all packets of the same group have the same transaction identifiers[14]. Every agent processing the packet, places transaction record in FOL[15] as usual part of transaction processing. In a typical scenario, where packet pipe-line contains only a single storage-out agent, redo-only transactions can be used.

FOL records of packet processing are pruned from the FOL as usual. On the undo phase of recovery, FOL is replayed and each node undoes local part of copy packet processing. On the redo recovery, an agent redoes packet processing, including forwarding the packet to other nodes, if versions match.

Copy machine call-backs are executed as part of the same transaction as the rest of packet processing.

Concurrency control

M0 will support variety of concurrency control mechanisms selected dynamically to optimize resource utilization. Without going into much detail, following mechanisms are considered for controlling access to cluster-wide object data:

- a whole file lock acquired on a meta-data server when cluster-wide object meta-data are fetched. This works only for cluster-wide objects visible in a file system name-space (*i.e.*, for files);
- an extent lock taken on one of the lock servers. A replicated lock service runs on the pool servers. Every cluster-wide object has an associated locking server where locks on extents of object data are taken. Locking server might be one of the servers where object data are stored;

[12][u.resource.grab]

[13][u.RESOURCE.CACHEABLE] ST

[14][u.dtm.tid.generate]

[15][u.fol.record.custom]

- "component level locking" is achieved by taking lock on an extent of object data on the same server where these data are located;
- time-stamp based optimistic concurrency control (see [9]).

Independently of whether a cluster-wide object level locking model[16], where data are protected by locks taken on cluster-wide object (these can be either extent locks taken in cluster-wide object byte offset name-space[17] or "whole-file" locks[18]), or component level locking model, or time-stamping model is used, locks or time-stamps are served by a potentially replicated locking service running on a set of *lock servers* (a set that might be equal to the set of servers in the pool). The standard locking protocol as used by the file system clients would imply that all locks or time-stamps necessary for an aggregation group processing must be acquired before any processing can be done. This implies a high degree of synchronization between agents processing copy packets from the same aggregation group.

Fortunately, this ordering requirement can be weakened by making every agent to take (the same) required lock and assuming that lock manager recognizes, by comparing transaction identifiers, that lock requests from different agents are part of the same transaction and, hence, are not in conflict[19]. Overhead of locking can be amortized by batching and locking-ahead.

storage-in agent A on a storage device D:

- for each repair priority P from highest to lowest:
  - for each container C on D[20], with priority P and containing data from the input set:
    - for each extent E of data from C[21] and from the input set:
      - for each sub-extent E' of E mapped by the aggregation function to the same agent G (G is an agent located on the same node as A):
        - for each chunk H of E' not larger than maximal buffer size accepted by G (maximal RPC size[22], as determined by the RPC layer when G is a collecting agent or maximal storage device transfer size[23], as determined by the storage device layer, when G is a write agent):
          - enqueue asynchronous distributed lock request[24] for H;

---

[16][u.dlm.logical-locking]

[17][u.IO.EXTENT-LOCKING] ST

[18][u.IO.MD-LOCKING] ST

[19][u.dlm.transaction-based]

[20][u.container.enumerate]

[21][u.CONTAINER.KEY-OFFSET] ST

[22][u.rpc.maximal-bulk-size]

[23][u.storage.transfer-size]

[24][u.dlm.enqueue.async], [u.IO.NON-BLOCKING] ST

            - end for each chunk
        - for each sub-extent
    - end for each extent
  - end for each container
- end for each priority
- on lock enqueue completion for a chunk H:
  - put H into ready queue;
  - invoke progress "enter" call-back if necessary
- end on lock enqueue
- on condition:
  - if D serves IO client requests[25], then the fraction of storage device bandwidth consumed by the copy machine agents on D drops below the threshold allotted to D (if device is under-utilised, all unused bandwidth can be used without affecting service times), grab a portion of bandwidth necessary for H and
  - there is a free buffer B in the copy machine buffer pool (grab B), and
  - there is a chunk H in the ready queue, do:
  - submit asynchronous request to read H into B[26]
- end on condition
- on read completion for chunk H into buffer B:
  - create a copy packet P for H;
  - queue(P, A)
- end on read completion

- on packet completion notification for P:
  - release P's resources;
  - invoke progress "leave" if necessary;
  - mark input set portion, corresponding to P as processed in the persistent state;
  - release DLM locks acquired for P
- end on packet completion notification

Nested loops above try to stream data to the target agents while respecting priority and resource consumption thresholds, and to do optimal storage transfers, by doing them in the container offset order[27] and relying on container based read-ahead[28].

storage-out agent on a storage device D:

---

[25][u.storage.utilization]

[26][u.container.async]

[27][u.container.offset-order]

[28][u.container.read-ahead]

- forever:
  - wait:
    - until a packet P is submitted to the agent for writing and
    - if D serves IO client requests[29], then until the fraction of storage device bandwidth consumed by the copy machine agents on D drops below the threshold allotted to D, grab a portion of bandwidth necessary to write P;
  - end wait
  - submit asynchronous request to write P to D[30];
- never
- on write completion for packet P:
  - free P's buffer back to the buffer pool;
  - release P's resources and send P completion notification back through the pipe-line;
- end on write completion
- on a client write request[31]:
  - if
    - the request intersects with the output set, and
    - D is a designated (see below) storage device for the request
  - then
    - create a fake copy packet P representing the request;
    - send P completion notification back through the P's pipe-line
  - end if
- end on client write request

The purpose of creating a fake copy packet for a client write request is to send completion notifications back through the pipe-line. Completion notifications are executed in the context of user IO transaction[32]. A designated storage device is selected so that a user IO triggers only one chain of completion notifications. For example, a designated device can be defined as a device containing the first (according to the natural ordering of striping units within a parity group) non-failed striping unit for the first (in the cluster-wide object offset order) parity group intersecting with the request.

collecting agent A:

- forever:
  - wait:
    - until a packet P is submitted to A;

[29][u.storage.utilization]

[30][u.container.async]

[31][u.storage.intercept]

[32][u.dtm.intercept]

        - until the fraction of processor cycles bandwidth[33] consumed by the copy machine agents drop below the threshold, grab a portion of processor cycles bandwidth necessary apply transformation function to P's contents;
    - end wait
    - if no packets from aggregation-group(P) were processed by A, then
        - apply transformation function to P and use P as aggregation group buffer
    - else
        - find aggregation group buffer for aggregation-group(P);
        - apply transformation function to P and merge P to the aggregation group buffer;
        - free B
    - end if
    - if all parts of aggregation-group(P) are collected
        - queue(P, A)
    - end if
- never
- on packet P completion notification:
    - release P's resources (free aggregation group buffer)
    - send completion notification to all agents from which packets of this aggregation group were received
- end on packet completion notification

network-out agent A:

- forever:
    - wait:
        - until a packet P is submitted to A;
        - until a fraction of network bandwidth[34] consumed by the copy machine is below the threshold, grab a portion of bandwidth necessary to submit P;
    - end wait
    - send asynchronous copypacket(P) RPC to next-agent(P, A)[35]
- never;
- on packet completion notification for P:
    - release P's resources;
    - send completion notification back through the P's pipe-line
- end on packet completion notification

network-in agent A:

- forever:

[33][u.processor.utilization]

[34][u.network.utilization]

[35][u.rpc.async]

    - wait:
        - until a copypacket(P)[36] RPC is received by A;
    - end wait;
    - queue(P, A);
- never

- on packet completion notification for P:
    - release P's resources;
    - send completion notification back through the P's pipe-line
- end on packet completion notification

Pool machine

Pool machine is a replicated state machine [5], having replicas on all pool nodes. Each replica maintains the following state:

```
node          : array of struct { id    : node identity,
                                  state : enum state };
device        : array of struct { id    : device identity,
                                  state : enum state };
read-version  : integer;
write-version : integer;
```

where `state` is `enum { ONLINE, FAILED, OFFLINE, RECOVERING }`. It is assumed that there is a function device-node() mapping device identity to the index in node[] corresponding to the node the device is currently attached to. The elements of the device[] array corresponding to devices attached to non-`ONLINE` nodes are effectively undefined (state transition function does not depend on them). To avoid mentioning this condition in the following, it is assumed that

```
device-node(device[i].id).state == ONLINE,
```

for any index `i` in device[] array, that is, devices attached to non-`ONLINE` nodes are excised from the state.

State transitions of a pool machine happen when the state is changed on a quorum[37] of replicas. To describe state transitions the following derived state (that is not necessary actually stored on replicas) is introduced:
`nr-nodes` : number of elements in node[] array
`nr-devices` : number of elements in device[] array
`nodes-in-state[S]` : number of elements in node[] array with the state field equal to S
`devices-in-state[S]` : number of elements in device[] array with the state field equal to S

[36][u.rpc.pluggable]

[37][u.quorum.consensus]

```
nodes-missing = nr-nodes - nodes-in-state[ONLINE]
devices-missing = nr-devices - devices-in-state[ONLINE]
```

In addition to the state described above, a pool is equipped with a "constant" (in the sense that its modifications are beyond the scope of the present design specification) configuration state including:

- `max-node-failures : integer`, a number of node failures that the pool tolerates;
- `max-device-failures : integer`, a number of storage device failures that the pool tolerates.

A pool is said to be *a dud* (Data possibly Unavailable or Damaged) when more device and node failed in it than the pool is configured to tolerate.

Based on the values of derived state fields, the pool machine state space is partitioned as:

devices-missing \ nodes-missing	0	1 .. max-node-failures	max-node-failures + 1 .. nr-nodes
0	normal	degraded	dud
1 .. max-device-failures	degraded	degraded	dud
max-device-failures + 1 .. nr-device	dud	dud	dud

A pool state with nodes-missing = n and devices-missing = k is said to belong to a *state class* $S_{(n, k)}$, for example, any normal state belongs to the class $S_{(0,0)}$.

As part of changing its state, a pool machine interacts with external entities such as layout manager or client caches. During this interaction multiple failures, delays and concurrent pool machine state transitions might happen. In general it is impossible to guarantee that all external state will be updated by the time the pool machine reaches its target state. To deal with this, pool state contains a version vector, some components of which are increased on any state transition. All external requests to the pool (specifically, IO requests) are tagged with the version vector of the pool state the request issuer knows about. The pool rejects requests with incompatibly stale versions, forcing issuer to re-new its knowledge of the pool state. Separate read and write versions are used to avoid unnecessary rejections. *E.g.*, read requests are not invalidated by adding a new device or a new server to the pool. Finer grained version vector can be used, if necessary.

Additional `STOPPED` state can be introduced for nodes and devices. This state is entered when a node or a device is deliberately temporarily inactivated, for example, to move a device from one node to another or to re-cable a node as part of preventive maintenance. After a device or a node stood in `STOPPED` state for more than some predefined time, it enters `OFFLINE` state.

See details in the State section.

Server state machine

Persistent server state consists of its copy of the pool state.

On boot a server contacts a quorum[38] of pool servers (counting itself) and updates its copy of the pool state. If recovery is necessary (unclean shutdown, server state as returned by the quorum is not OFFLINE), the server changes the pool state (through the quorum) to register that it is recovering. After the recovery of distributed transactions completes, the server changes the pool state to indicate that the server is now in ONLINE state (which must have been the server's pre-recovery state). See details in the State section.

Scalable IO

As described in the Logical specification, at the heart of SNS repair design is a *copy machine*: a mechanism for scalable data re-structuring. The same copy machine code can be used for variety of purposes, some described in certain detail in this document. One particularly important use of copy machine outside of SNS repair proper is a "normal" client IO.

In the simplest case, a client performs a non-cached write operation on a file. A very simple copy machine consisting of a single *user-in agent* reading data from the user space, a network-out agent sending data to the servers and a network-in and storage-out agents on the servers can be used. The input set description in this case consists of a single buffer in the user application address space or `iovec` vector describing a collection of such buffers. Output set description is similarly given by a extent or a vector of extents in the file layout. An obvious generalisation of this is a copy machine with multiple user-in agents copying segments of an input buffer in parallel. This design can utilise throughput of multiple processor cores at the expense of weakening POSIX failure semantics. The same applies for file read, with *user-out* agent(s) copying data to the user space.

Similar copy machines can be employed for cached IO, except that input set description is done in terms of local file layout (based on page cache indexing).

An alternative construction of a copy machine for cached IO is one with an input set description made directly in terms of cached data pages. This copy machine would be able to form copy packets containing data from multiple objects and multiple files. To unify this model with earlier described layout based input set descriptions, a client allocates a number of local containers and populates them with cached data. A copy machine sends out parts of these containers. With this approach, multi-object cached IO and container migration become instances of the same generic data re-structuring process, sharing the copy machine infrastructure.

An important advantage of copy machine based IO is a flexibility in data routing. The description of SNS repair above mentioned the possibility of partial aggregation of data on intermediate servers. In the case of client IO, the same mechanism makes it possible to route data through proxy nodes, including proxy servers and peer-to-peer clients. For example, an owning client can be selected for every parity group in a file with IO requests to this group being routed through this client. In addition

[38] [u.quorum.read]

to utilising client to client network bandwidth, this allows makes client to server operations more efficient because they would more likely be full-stripe ones.

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.sns.repair.triggers] A pool machine registers with health layer its interest[39] in hearing about device[40], node[41] and network[42] failures. When health layer notifies[43] the pool machine about a failure, state transition happens[44] and repair, if necessary, is triggered.
- [i.sns.repair.code-reuse] Local RAID repair is a special case of general repair. When a storage device fails in a way that requires only local repair, the pool machine records this failure as in general case and creates a copy engine to handle the repair. All agents of this machine are operating on the same node.
- [i.sns.repair.layout-update] When a pool state machine enters a non-normal state, it changes its version numbers. Clients attempting to do IO on layouts tagged with old version numbers, would have to re-fetch the pool state. Optionally, requests layout manager to proactively revoke all layouts intersecting[45] with the failed device or node. Optionally, use copy machine "enter layout" progress call-back to revoke a particular layout. As part of re-fetching layouts, clients learn updated list of alive nodes and devices. This list is a parameter to layout[46]. The layout IO engine uses this parameter to do IO in degraded mode[47].
- [i.sns.repair.client-io] Client IO continues as repair is going on. This is achieved by re-directing clients to degraded layouts, so that clients collaborate with the copy machine in repair. After

[39][u.health.interest]

[40][u.health.device]

[41][u.health.node]

[42][u.health.network]

[43][u.health.call-back]

[44][u.health.fault-tolerance]

[45][u.layout.intersects]

[46][u.layout.parameter.dead]

[47][u.layout.degraded-mode]

copy machine notifies pool machine of processing progress (through the "leave" progress call-back), repaired parts of layout[48] are upgraded;

- [i.sns.repair.io-copy-reuse] The following table gives input parameters to the copy machines implementing required shared functionality:

	layout setup	aggregation function	transformation function	"enter layout" progress call-back	"leave layout" progress call-back
in-place repair		aggregate striping units	recalculate lost striping units	layout moved into degraded mode	upgrade layout out of degraded mode
NBA repair	original layout moved into degraded mode, new NBA layout created for writes	aggregate striping units	recalculate lost striping units		update NBA layout
migration	migration layout created	no aggregation	identity		discard old layout
replication	replication layout created	no aggregation	identity		nothing
snapshot taking	new layout created for writes	no aggregation	identity		nothing

- [i.sns.repair.priority] Containers can be assigned a *repair priority* specifying in what order they are to be repaired. Prioritization is part of the storage-in agent logic.
- [i.sns.repair.degraded] Pool state machine is in degraded mode during repair: described in the pool machine logical specification. Individual layouts are moved out of degraded mode as they are reconstructed: when copy machine is done with all components of a layout, it signals layout manager that layout can be upgraded (either lazily[49] or by revoking all degraded layouts).
- [i.sns.repair.c4]
    - Repair is controllable by advanced C4 settings: can be paused, its IO priority can be changed. This is guaranteed by dynamically adjustable copy machine resource consumption thresholds.
    - Repair reports its progress to C4. This is guaranteed by the standard state machine functionality.
- [i.sns.repair.addb] Repair should produce ADDB records of its actions: this is a part of standard state machine functionality50.

[48] [u.LAYOUT.EXTENT] ST

[49] [u.layout.lazy-invalidation]

- [i.sns.repair.device-oriented] Repair uses device-oriented repair algorithm, as described in *On-line Data reconstruction in Redundant Disk Arrays* dissertation: this follows from the storage-in agent processing logic.
- [i.sns.repair.failure.transient] Repair survives transient node and network failures. After failed node restarts or network partitions heals, distributed transactions, including repair transactions created by copy machine are redone or undone to restore consistency. Due to the construction of repair transactions, recovery also restores repair to a consistent state, from which it can resume.
- [i.sns.repair.failure.permanent] Repair handles permanent failures gracefully. Repair updates file layouts with at the transaction boundary. Together with copy machine state replication, described in the Persistent state sub-section, this guarantees that repair can continue in the face of multiple failures.
- [i.sns.repair.used-only] Repair should not reconstruct unused (free) parts of failed storage: this is a property of a container based repair design.

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.]

- layouts
  - [r.layout.intersects]: it must be possible to efficiently find all layouts intersecting with a given server or a given storage device;
  - [r.layout.parameter.dead]: a list of failed servers and devices is a parameter to a layout formula;
  - [r.layout.degraded-mode]: layout IO engine does degraded mode IO if directed to do so by the layout parameters;
  - [r.layout.lazy-invalidation]: layout can be invalidated lazily, on a next IO request;
- DTM
  - [r.fol.record.custom] : custom FOL record type, with user defined redo and undo actions can be defined;
  - [r.dtm.intercept]: it is possible to execute additional actions in the context of a user-level transaction;
  - [r.dtm.tid.generate]: transaction identifiers can be assigned by DTM users;
- management tool
- RPC
  - [r.rpc.maximal.bulk-size]
  - [r.network.utilization]: an interface to estimate network utilization;
  - [r.rpc.pluggable]: it is possible to register a call-back to be called by the RPC layer to process a particular RPC type;
  - health and liveness layer:
  - [r.health.interest], [r.health.node], [r.health.device], [r.health.network] it is possible to register interest in certain failure event types (network, node, storage device) for certain system components (e.g., all nodes in a pool);
  - [r.health.call-back] liveness layer invokes a call-back when an event on interest happens;
  - [r.health.fault-tolerance] liveness layer is fault-tolerant. Call-back invocation is carried through the node and network failures;

    - [r.rpc.streaming.bandwidth]: optimally streamed RPCs can utilize at least 95% of raw network bandwidth;
    - [r.rpc.async]: there is an asynchronous RPC sending interface;
- DLM
    - [r.dlm.enqueue.async]: a lock can be enqueued asynchronously;
    - [r.dlm.logical-locking]: locks are taken on cluster-wide objects;
    - [r.dlm.transaction-based]: lock requests issued on behalf of transactions. Lock requests made on behalf of the same transaction are never in conflict;
- meta-data:
    - [u.md.iterator]: generic meta-data iterators, suitable for input set description;
    - [u.md.iterator.position]: meta-data iterators come with a totally ordered space of possible iteration positions;
- state machines:
    - [r.machine.addb]: state machines report statistics about their state transitions to ADDB;
    - [r.machine.persistence]: state machine can be made persistent and recoverable. Local transaction manager invokes restart event on persistent state machines after node reboots;
    - [r.machine.discoverability]: state machines can be discovered by c4;
    - [r.machine.queuing]: a state machine has a queue of incoming requests;
- containers:
    - [r.container.enumerate]: it is possible to efficiently iterate through the containers stored (at the moment) on a given storage device;
    - [r.container.migration.call-back]: a container notifies interested parties in its migration events;
    - [r.container.migration.vote]: container migration, if possible, includes a voting phase, giving interested parties an opportunity to prepare for the future migration;
    - [r.container.offset-order]: container offset order matches underlying storage device block ordering enough to make container offset ordered transfers optimal;
    - [r.container.read-ahead]: container do read-ahead;
    - [r.container.streaming.bandwidth]: large-chunk streaming container IO can utilize at least 95% of raw storage device throughput;
    - [r.container.async]: there is an asynchronous container IO interface;
- storage:
    - [r.storage.utilization]: an interface to measure a utilization a given device for a certain time period;
    - [r.storage.transfer-size]: an interface to determine maximal efficient request size of a given storage device;
    - [r.storage.intercept]: it should be possible to intercept IO requests targeting a given storage device;
- SNS:
    - [r.sns.trusted-client] (constraint): only trusted clients can operate on SNS objects;
- miscellaneous:
    - [r.processor.utilization]: an interface to measure processor utilization for a certain time period;
- resource management:
    - [r.resource.generic]: resource management infrastructure is generic enough to implement copy machine resource limits;
- quorum:

- [r.quorum.consensus]: quorum based consensus mechanism is needed;
- [r.quorum.read]: read access to quorum decisions is needed;

5.3. Security model

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

5.3.1. Network

It is assumed that messages exchanged over the network are signed so that a message sender can be established reliably. Under this condition, nodes cannot impersonate each other.

5.3.2. Servers

Present design provides very little protection against a compromised server. While compromised storage-in or network agents can be detected by using striping redundancy information, there is no way to independently validate the output of a collecting agent or check that storage-out agent in fact wrote the right data to the storage. This issue is, in general, unavoidable, as long as output set can be non-redundant.

If we restrict ourselves to the situations where output set is always redundant, quorum based agreement can be used to deal with malicious servers in the spirit of [7]. Replicated state machine design of a copy machine lends itself naturally to a quorum based solution.

Deeper problem is due to servers collaborating in distributed transactions. Given that transaction identifiers used by the copy machine are generated by a known method, a server can check that server-to-server requests it receives are from well-formed transactions and a malicious server cannot wreak chaos by initiating malformed transactions. What is harder to counter is a server *not* sending requests that it must send according to the copying algorithm. We assume that the worst thing that can happen when a server delays or omits certain messages is that corresponding transaction will eventually be aborted and undone. Unresponsive server is evicted from the cluster and pool handles this as a server failure. This still doesn't guarantee progress, because the server might immediately re-join the cluster only to sabotage more transactions.

The systematic solution to such problems is to exploit already present redundancy in the input set. For example, when a layout with N+K (where K > 2) striping pattern is repaired after a single failure, N+K-1 survived striping units are gathered from each parity group. Collecting agent uses additional units to check in three ways that every received unit matches the redundancy and uses majority in case of mismatch. This guarantees that a single malign server can be detected. Obviously, RAID-like striping patterns can be generalized from fail-stop failures to Byzantine failures. It seems likely, that, as is typical for agreement protocols, N+K pattern with K > 2*F would suffice to handle up to F arbitrary failures (including usual fail-stop failures).

5.3.3. Clients

The fundamental difference between a server and a client is that the latter cannot, in general, be replicated, because it runs arbitrary code outside of M0 control. While well-formedness of client

supplied transactions and client liveness can be checked with some effort, there is no obvious way to verify that a client calculates redundancy information correctly, without sacrificing system performance to a considerable degree. It is, hence, posited that SNS operations, including client interaction with repair machinery, can originate only from the trusted clients[50].

5.3.4. Others

SNS repair interacts with and depends on variety of core distributed M0 services including liveness layer, lock servers, distributed transaction manager and management tool. Security concerns for such services should be addressed generically and are beyond the scope of the present design.

5.3.5. Issues

It is in no way clear that the analysis above is any close to exhaustive. A formal security model is required[51].

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.]

- copy machine
  - input set
    - estimate servers
    - estimate devices
    - efficiently enumerate all containers an input set has on a given device
    - input set *vs.* container migration
    - efficient check of whether given container contains data from an input set
    - efficiently enumerate all data extents common between an input set and a container
    - processed extents of an input set can be efficiently and compactly coded by an agent
  - agents creation is inexpensive
  - resource thresholds can be adjusted dynamically, agents should be notified about new thresholds

[50][u.sns.trusted-client]

[51][u.SECURITY.FORMAL-MODEL] ST

    - agents track fraction of device throughput consumed
    - distribute resource utilization quotas among agents
    - a buffer from the copy machine buffer pool can be obtained without dead-lock:
        - buffer user is not allowed to wait for other agent progress;
        - storage device IO and network RPCs do not involve copy machine operations (*e.g.*, a storage device cannot be a T1 export);
        - collecting agent violates these rules by keeping aggregation group buffer while waiting for the rest of the group to arrive.

      Alternatively, buffers can be preallocated.
    - buffer pool:
        - it is possible to wait for a free buffer;
    - an agent can wait until a buffer is submitted to it;
    - transformation function: it is possible to estimate how much its application costs in processor cycles;
    - aggregation groups can be determined efficiently;
    - construction of next-agent() function from aggregation function;
    - completion of layout repairing has to be detected;
    - redo and undo actions for agents must be defined and registered with DTM;
    - collecting agent must be able to efficiently determine from which agents it (would have) received packets for a given aggregation group so that completion notifications can be sent back;
    - lock requests by different storage-in agents must be of the same size, so that DLM could match them;
- pool machine:
    - device-node function: mapping between device identifier and node identifier is an implicit part of pool state.

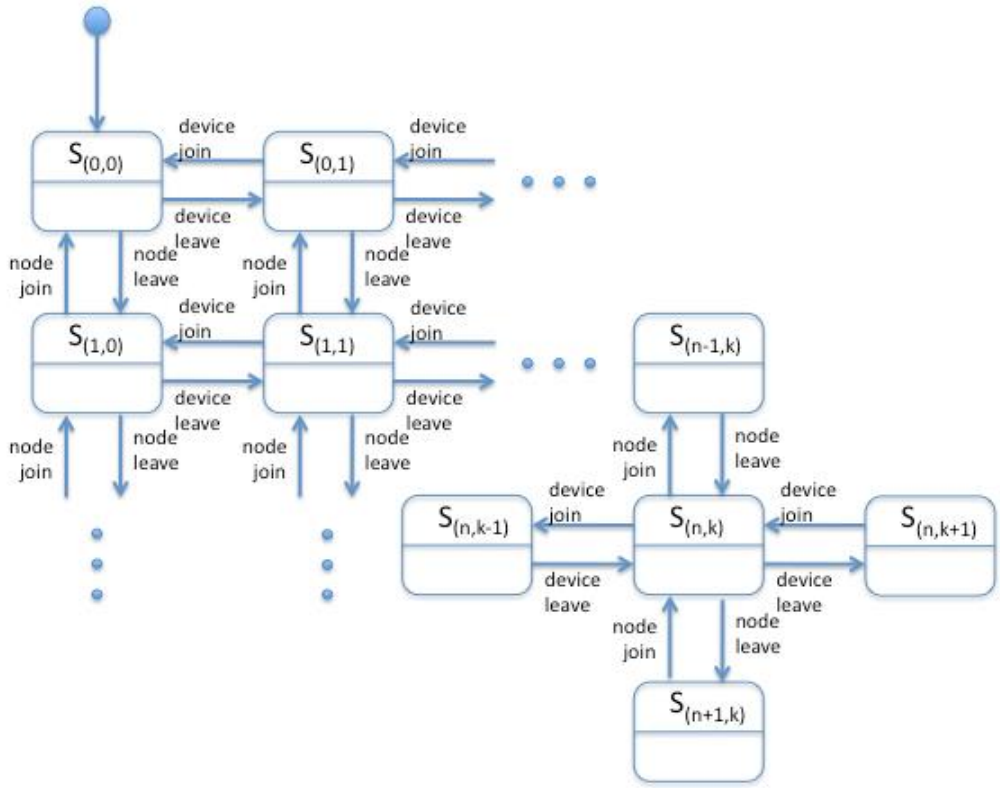
6. State

[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.]

6.1. Pool machine states, events, transitions

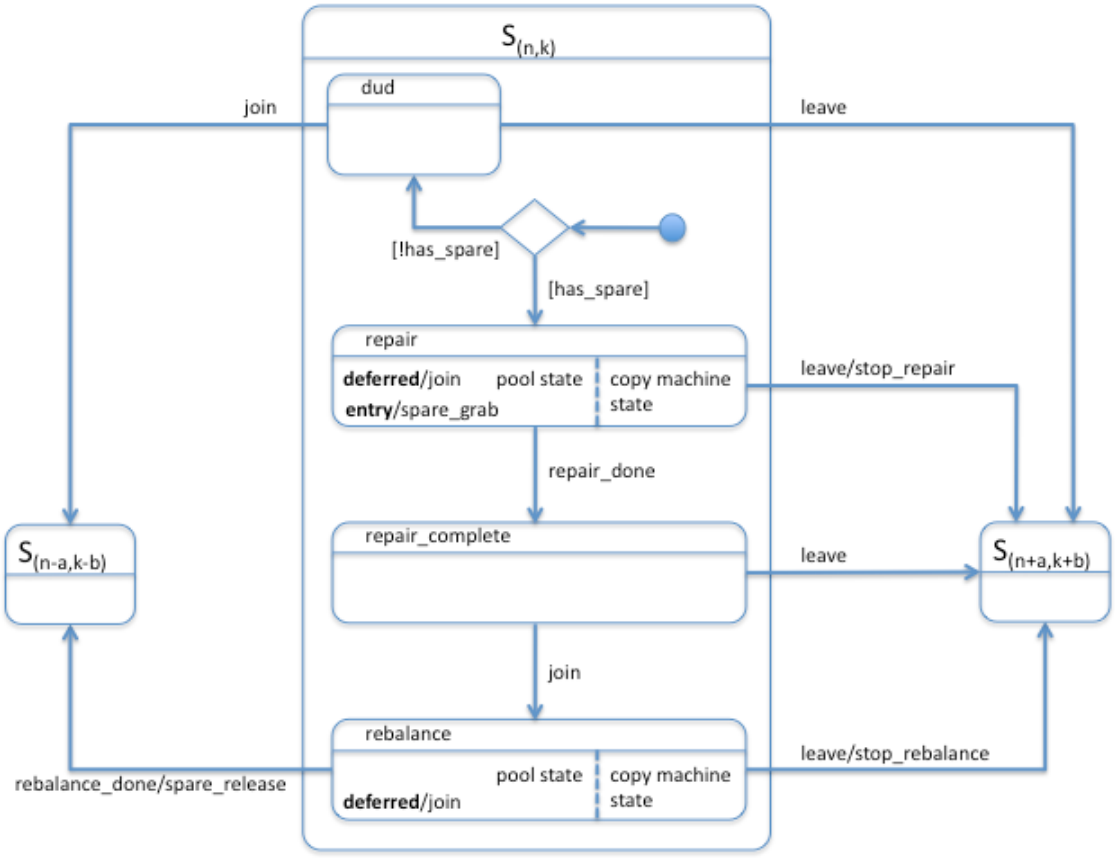
[This sub-section enumerates state machine states, input and output events and state transitions incurred by the events with a table or diagram of possible state transitions. UML state diagrams can be used here.]

Pool machine states can be classified into state classes $S_{(n, k)}$. "Macroscopic" transitions between state classes re described by the following state machine diagram:



Here *device leave* is any event that increases *devices-missing* field of pool machine state: planned device shutdown, device failure, detaching a device from a server, *etc*. Similarly, device join is any event that decreases *devices-missing*: addition of a new device to the pool, device startup, *etc*. The same, *mutatis mutandis* for *node leave* and *node join* events.

Within each state-class the following "microscopic" state transitions happen:



Where *join* is either *node_join* or *device_join* and *leave* is either *node_leave* or *device_leave*; and *spare* means distributed spare space.

Or, in the table form:

	has_spare_space	!has_spare_space	repair_done	rebalance_done	join	leave
choice pseudo-state	repair /spare_grab(); start_repair_machine()	dud	impossible	impossible	impossible	impossible
repair	impossible	impossible	repair_complete	impossible	defer, queue	$S_{(n+a, k+b)}$ /stop_repair()
repair_complete	impossible	impossible	impossible	impossible	rebalance	$S_{(n+a, k+b)}$
rebalance	impossible	impossible	impossible	$S_{(n-a, k-b)}$	defer, queue	$S_{(n+a, k+b)}$

				/spare_release()		/stop_rebalance()

Recall that a pool state machine is replicated and its state transition is, in fact, a state transition on a quorum of replicas. Impossible state transitions happening when a replica receives an unexpected event are logged and ignored. It's easy to see that every transition out of $S_{(n, k)}$ state class is either directly caused by a *join* or *leave* event or directly preceded by such an event.

Events:

- storage device failure;
- node failure;
- network failure;
- storage device recovery;
- node recovery;
- network recovery;
- media failure;
- container transition;
- client read;
- client write;

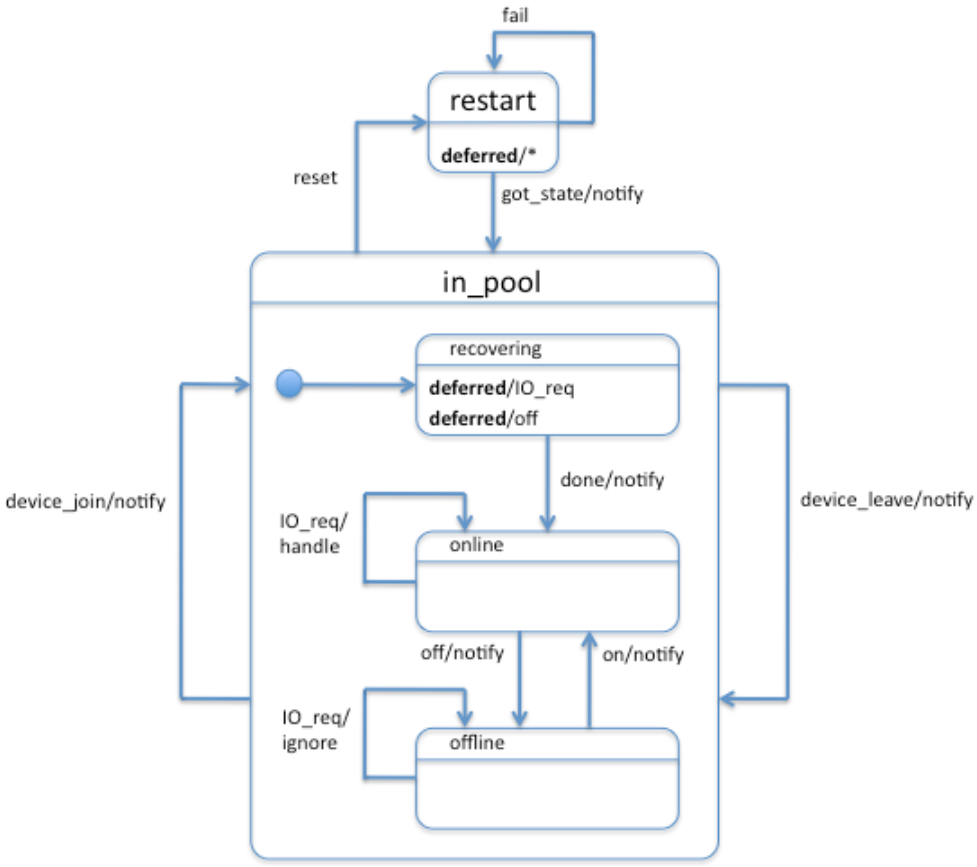
6.2. Pool machine state invariants

In a state class $S_{(n, k)}$ the following invariants are maintained (for a replicated machine a state invariant is a condition that is true on at least some quorum of state replicas):

- if n <= max-node-failures and k <= max-device-failures, then exactly F(n, k)/F(max-node-failures, max-device-failures) of pool (distributed) spare space is busy, where the definition of F(n, k) function depends on details of striping pattern is used by the pool (described elsewhere). Otherwise the pool is a dud and all spare space is busy;
- (this is not, technically speaking, an invariant) version vector part of pool machine state is updated so that layouts issued before previous cross-class state transition can be invalidated if necessary;
- no repair or rebalancing copy machine is running when a state class is entered or left.

6.3. Server machine states, events, transitions

State transition diagram:



Where

- *restart* state queries pool quorum (including the server itself) about pool machine state (including the server state);
- *notify* action notifies replicated pool machine about changes in the server state or in the state of some storage device attached to the server.

In the table form:

	restart	in_pool.recovering	in_pool.online	in_pool.offline
got_state	in_pool.recovering /notify	impossible	impossible	impossible
fail	restart	impossible	impossible	impossible
done	impossible	online/notify	impossible	impossible
off	impossible	impossible	offline/notify	impossible
on	impossible	impossible	impossible	online/notify
IO_req	defer	defer	online/process	offline/ignore

device_join	defer	defer	online/notify	offline/notify
device_leave	defer	defer	online/notify	offline/notify
reset	restart	restart	restart	restart

6.4. Server state machine invariants

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

Server state machine: no storage operations in OFFLINE state.

6.5. 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.]

All state machines function according to the Run-to-completion (RTC) model where each state transition is executed completely before next state transition is allowed to start. Queuing[52] is used to defer concurrently incoming events.

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.]

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

Scenario	usecase.repair.throughput-single
Business goals	high availability
Relevant quality attributes	scalability
Stimulus	repair invocation
Stimulus source	node, storage or network failure, or administrative action
Environment	server pool

[52][u.machine.queuing]

Artifact	repair data reconstruction process running on the pool
Response	repair utilises hardware efficiently
Response measure	repair can utilize at least 90 percent of a raw hardware bandwidth of any storage device and any network connection it uses, subject to administrative restrictions. This is achieved by: ● the streaming IO done by storage-in and storage-out agents, together with the guarantee that large-chunk streaming container IO can consume at least 95% of raw storage device bandwidth[53], ● streaming network transfers done by network-in and -out agents, together with the guarantee that optimal network transfers can consume at least 95% of raw network bandwidth[54] , ● assumption that there is enough processor cycles to reconstruct data from redundant information without the processor being bottleneck. More convincing argument can be made by simulating the repair.
Questions and issues	

Scenario	usecase.repair.throughput-total
Business goals	high availability
Relevant quality attributes	scalability
Stimulus	repair invocation
Stimulus source	node, storage or network failure, or administrative action
Environment	a pool
Artifact	repair data reconstruction process running on the pool
Response	repair process utilizes storage and network bandwidth of as many pool elements as possible, even if some elements have already failed and are not being replaced.
Response measure	fraction of pool elements participating in the repair, as a function of a number of failed units. This is achieved by distributed parity layouts, on average uniformly spreading

[53][u.container.streaming.bandwidth]

[54][u.rpc.streaming.bandwidth]

	parity groups across all devices in a pool. See [3] for details.
Questions and issues	

Scenario	usecase.repair.degradation
Business goals	maintain an acceptable level of system performance in degraded mode
Relevant quality attributes	availability
Stimulus	repair invocation
Stimulus source	node, storage or network failure, or administrative action
Environment	a pool
Artifact	repair process competing with ongoing IO requests to the pool
Response	fraction of the total pool throughput consumed by the repair at any moment in time is limited
Response measure	Fraction of total throughput consumed by the repair at any moment is lower than the specified limitation. This is achieved by: repair algorithm throttles itself to consume no more than a certain fraction of system resources (storage bandwidth, network bandwidth, memory) allocated to it by a system parameter. The following agents will do the throttle respectively according to its parameters: ● storage-in agent, ● storage-out agent, ● network-in agent, ● network-out agent, ● collecting agent Additionally, storage and network IO requests are issued by repair with a certain priority, controllable by a system parameter.
Questions and issues	

Scenario	usecase.repair.io-copy-reuse
Business goals	flexible deployment

Relevant quality attributes	re-useability
Stimulus	local RAID repair
Stimulus source	storage unit failure on a node
Environment	M0 node with a failed storage unit
Artifact	local RAID repair
Response	local RAID repair uses the same algorithms and the same data-structures as network array repair
Response measure	a ratio of code shared between local and network repair. This is achieved by: ● the same algorithm and data structures will be used to do the parity computing, data reconstructing, resource consumption limitation, *etc.*
Questions and issues	

Scenario	usecase.repair.multiple-failure
Business goals	system behaves predictably in any failure scenario - Failures beyond redundancy need to have a likelihood over the lifetime of the system (e.g. 5-10 years) to achieve a certain number of 9's in data availability/reliability. The case where not an entire drive fails beyond the redundancy level (media failure) is considered elsewhere.
Relevant quality attributes	fault-tolerance
Stimulus	a node, storage or network failure happens while repair is going on
Stimulus source	hardware or software malfunction
Environment	a pool in degraded mode
Artifact	more units from a certain parity group are erased by the failure than a striping pattern can recover from
Response	repair identifies lost data and communicates the information about data loss to the interested parties.
Response measure	data loss is contained and identified. This is achieved by proper pool state machine transition: ● When more units from a certain parity group are detected to be failed than a stripe patter can tolerate, the pool machine transits to DUD, and ● Internal read-version and write-version of the state

	machine will be increased, and pending client I/O will get error.
Questions and issues	

Scenario	usecase.repair.management
Business goals	system behavior is controllable by C4, but normally automatic. Will be reported by C4, some parameters are somewhat tunable (in the advanced-advanced-advanced box)
Relevant quality attributes	observability, manageability
Stimulus	a control request to repair from a management tool
Stimulus source	management tool
Environment	an array in degraded mode
Artifact	management tool can request repair cancellation, change to repair IO priority
Response	repair executes control request. Additionally, repair notifies management tool about state changes: start, stop, double failure, ETA.
Response measure	Control requests are handled properly, correctly and timely. Repair status and events are reported to C4 properly, correctly and timely. This is achieved by the commander handler in the SNS repair manager and its call-backs to C4.
Questions and issues	

Scenario	usecase.repair.migration
Business goals	
Relevant quality attributes	re-usability
Stimulus	migration of a file-set from one pool to another
Stimulus source	administrative action or policy decision (*e.g.*, space re-balancing)
Environment	normal system operation

Artifact	a process to migrate data from the pool starts
Response	• Migrate data according to its policy correctly, under the limitation of resource usage. • data migration re-uses algorithms and data-structures of repair
Response measure	• Data is migrated correctly. • A ratio of code shared between migration and repair These are achieved by: • Using the same components and algorithm with repair, but with different integration.
Questions and issues	

Scenario	usecase.repair.replication
Business goals	
Relevant quality attributes	re-usability
Stimulus	replication of a file-set from one pool to another
Stimulus source	administrative action or policy decision
Environment	normal file system operation
Artifact	a process to replicate data from the pool
Response	data replication re-uses algorithms and data-structures of repair
Response measure	a ratio of code shared between replication and repair. This is achieved by using the same components and algorithm with repair, but with different integration.
Questions and issues	

Scenario	usecase.repair.resurrection (optional)
Business goals	
Relevant quality attributes	fault tolerance
Stimulus	a failed storage unit or data server comes back into service

Stimulus source	
Environment	a storage pool in a degraded mode
Artifact	repair detects that reconstructed data are back online
Response	depending on the fraction of data already reconstructed various policies can be selected: ● abort the repair and copy all data modified since reconstruction back to the original, restore the layout to its original one. ● abandon original data and continue repair.
Response measure	Less time and resource should be used, regardless to continue the repair or not.
Questions and issues	If we choose to restore layouts to its original state, it is a potentially lengthy process (definitely not atomic) and additional failures can happen while it is in progress. It requires scanning already processed layouts and reverting them to their original form, freeing spare space. This is further complicated by the possibility of client IO modifying data stored in the spare space before roll-back starts. So, resurrection will be marked as "optional" feature to be implemented later.

Scenario	usecase.repair.local
Business goals	
Relevant quality attributes	resource usage
Stimulus	local RAID repair starts
Stimulus source	storage unit failure on a node
Environment	M0 node with a failed storage unit
Artifact	local RAID repair
Response	local RAID repair uses copy machine buffer pool to exchange data. No network traffic is needed.
Response measure	No network traffic in the repair. This is achieved by running storage-in agent, collecting agent, and storage out agent on the same node, and exchanging data through buffer pool.
Questions and issues	

Scenario	usecase.repair.ADDB
Business goals	better diagnostics
Relevant quality attributes	ADDB
Stimulus	repair
Stimulus source	SNS repair
Environment	Running SNS repair in M0
Artifact	Diagnostic information are logged in ADDB
Response	SNS repair log status, state transition, progress etc. in ADDB for better diagnostic in the future.
Response measure	The amount of ADDB records useful for later diagnostic. This is achieved by integrating ADDB infrastructure tightly into SNS repair, and producing ADDB records correctly, efficiently, timely.
Questions and issues	

Scenario	usecase.repair.persistency
Business goals	availability
Relevant quality attributes	local and distributed transactions
Stimulus	SNS
Stimulus source	SNS
Environment	node in M0
Artifact	State machines survive from node failures
Response	State machines use services of local and distributed transaction managers to recover from node failures. After a restart, persistent state machine receives a restart event, that it can use to recover its lost volatile state.
Response measure	State machines survive from node failures. This is achieved by using replicated state machine mechanism.
Questions and issues	

Scenario	usecase.repair.priority
Business goals	availability
Relevant quality attributes	container
Stimulus	SNS repair initialization
Stimulus source	Pool machine
Environment	a running SNS repair
Artifact	Priority of SNS repair assigned.
Response	A priority is set for every container included in the input set, and the SNS repair will be executed by this priority.
Response measure	SNS repairs are executed with higher priority first. This is achieved by a looping from the highest priority to the lowest one to initiate new repairs.
Questions and issues	

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.]

See Logical specification and State section for failure handling and Security model sub-section for Byzantine failures.

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.]

Major input parameters, affecting SNS repair behavior are:

- number of storage devices attached to a server;
- number of servers in the pool;
- storage device bandwidth;
- storage device capacity;
- storage device space utilization;
- server-to-server network bandwidth;
- processor bandwidth per server;
- frequency and statistical distribution of client IO with the pool;
- frequency and distribution of permanent device failures;
- frequency and distribution of permanent server failures;
- frequency and distribution of transient server failures (restarts);
- mean time to replace a storage device;
- mean time to replace a server;
- fraction of storage bandwidth used by repair;
- fraction of storage bandwidth used by re-balancing;
- fraction of network bandwidth used by repair;
- fraction of network bandwidth used by re-balancing;
- degradation in visible client IO rates;
- pool striping pattern: (N+K)/G.

Major SNS repair behavior metrics, affected by the above parameters are:

- mean time to repair a storage device;
- mean time to repair a server;
- mean time to re-balance to a new storage device;
- mean time to re-balance to a new server;

To keep this section reasonably short, a number of simplifying assumptions, some of which can be easily lifted, are made:

- a pool consists of N_D devices attached to N_S servers (the same number of devices on each server);
- every cluster-wide object is N+K striped across all servers and devices in the pool using parity de-clustering;
- device size is S_D (bytes);
- average device utilization (a fraction of used device space) is U;
- device bandwidth is B_D (bytes/sec);
- server-to-server network bandwidth is B_S (bytes/sec);
- server processor bandwidth is B_P (defined as a rate at which RAID6 redundancy codes can be calculated, bytes/sec);
- fractions of respectively storage, network and processor bandwidth dedicated to repair are A_S, A_N and A_P.

Let's consider a steady state of repair in a pool with F_S failed servers and F_D failed devices (F_D includes all devices on failed servers), assuming at most one unit is lost in any parity group. Define G_D, G_S, G_P and G_O as rates (bytes/sec) at which repair reads data from a device, sends data from a given server, computes redundancy codes and writes data to a device respectively.

Every one of N_S - F_S survived servers has on average (N_D - F_D)/(N_S - F_S) devices attached and from each of these data are read at the rate G_D. Assuming that none of this data are for "internal

consumption" (that is, assuming that no parity group has a spare space unit on a server where it has data or parity units), servers sends out all these data, giving

$$G_D \cdot \frac{N_D - F_D}{N_S - F_S} = G_S$$

Every server fans data out to every other survived server. Hence, every server receives data at the same G_S rate. Received data (again assuming no "internal consumption") are processed at G_P rate, giving

$$G_S = G_P$$

Redundancy codes calculation produces a byte of output for every N bytes of input. Finally, reconstructed data are uniformly distributed across all the devices of the server and written out, giving

$$G_P \cdot \frac{1}{N} = G_O \cdot \frac{N_D - F_D}{N_S - F_S}$$

Steady state rates are subject to the following constraints:

$$G_D + G_O \leq A_D \cdot B_D$$

$$G_S \leq A_S \cdot B_S$$

$$G_P \leq A_P \cdot B_P$$

To reconstruct a failed unit in a parity group, N of its N + K - 1 units, scattered across N_D - 1 devices have to be read, meaning that to reconstruct a device an $N/(N_D - 1)$ fraction of used space on every device in the pool has to be read, giving

$$MTTR_D = \frac{U \cdot S_D}{G_D} \cdot \frac{N}{N_D - 1}$$

as a mean time to repair a device. To minimize $MTTR_D$, G_D has to be maximized. From the equations and inequalities above, the maximal possible value of G_D is obviously

$$G_D = min\left(A_S \cdot B_S \cdot \chi, A_P \cdot B_P \cdot \chi, \frac{A_D \cdot B_D}{1 + 1/N}\right)$$

where $\chi = \frac{N_S - F_S}{N_D - F_D}$.

Let's substitute vaguely reasonable data:

Parameter	Value	Unit	Explanation

U	1.0		
S_D	2.0e12	bytes	`2TB drive`
B_S	4.0e9	bytes/sec	IB QDR
B_P	8.0e9	bytes/sec	check-sum throughput
B_D	7.0e7	bytes/sec	`ST31000640SS`
A_S	1		
A_D	1		
A_P	1		

See this spreadsheet for details.

For a small configuration with $N_S = 1$, $N_D = 48$:

- 4+2 striping: G_D is 56MB/sec, $MTTR_D$ is 3800 sec;
- 1+1 striping (mirroring): G_D is 35MB/sec, $MTTR_D$ is 1215 sec.

For a larger configuration with $N_S = 10$, $N_D = 480$:

- 10+3 striping: G_D is 64MB/sec, $MTTR_D$ is 787 sec;
- 1+1 striping: G_D is 35MB/sec, $MTTR_D$ is 119 sec.

In all cases, storage IO is the bottleneck.

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.]

- flow control. Should network-in agents drop incoming copy packets when node runs out of resource limits?
- pipeline based flow control.
- a dedicated lock(-ahead) agent can be split out of storage-in agent for uniformity.

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.]

[1] SNS Repair Requirement Analysis.
[2] Summary Requirement Table.
[3] *On-Line Data Reconstruction In Redundant Disk Arrays*, Mark Holland, dissertation.
[4] On Layouts.
[5] State machine replication, Wikipedia.
[6] traceability memo.
[7] *Practical Byzantine Fault Tolerance*, Miguel Castro, Barbara Liskov
[8] HLD Inspection Trail for SNS Repair.

[9] *Scalable Concurrency Control and Recovery for Shared Storage Arrays*, Khalil Amiri, Garth A. Gibson, Richard Golding.

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	Actual time	Comments

					(min.)	**(min.)**	
Peter	SNS repair	HLDINSP	1		180		
Andy	SNS repair	HLDINSP	1		180		

11.2. Logd

No.	**Task**	**Summary**	**Reported by**	**Date reported**	**Comments**
1	SNS repair	use cases must show *how* the design deals with the use case	nikita	2010.03.18	
2	SNS repair				
3	SNS repair				
4	SNS repair				
5	SNS repair				
6	SNS repair				
7	SNS repair				
8	SNS repair				
9	SNS repair				