This document summarizes SNS and local RAID discussions in the C2 architecture team. The goal is to provide a starting point for design level discussions and resource estimation. Note that this documents is not a substitute for a list of requirements or use-cases. Intended audience includes T1 team and C2 architects.

Overview.

SNS (Server Network Striping) and local RAID are mechanisms to achieve data redundancy or to increase IO throughput. Assuming knowledge of RAID, SNS is, roughly speaking, a network RAID, where a file is stored in multiple component files, allocated on separate data servers. The term *striping* will be used to refer to both SNS and local raid. Striping includes non-redundant (RAID0) as well as redundant (*e.g.*, RAID5) patterns.

Items.

- Striping is, generally, two-level. SNS stores a file in a collection of component files, using logical offsets within component files to determine where file data go. A component file is stored by a C2 back-end in a *container*, which is a thin abstraction layer on top of device. A container can itself be striped over other containers (or de-duped, encrypted, compressed, *etc.*).

- The same algorithms and the same data-structures are used for SNS and local RAID.

- The key piece of meta-data used to implement striping (and all other forms of file storage in C2) is *layout* (see On Layouts). A layout is a map that transforms an offset in a file into a location where data at this offset reside. For example, an SNS file layout transforms a file offset into a `(component-file-id, offset)` pair, where offset is a logical offset within the component file. A component file has layout too, and this layout can be used to transform the offset into location. In other words, layouts are traversed until final data location is found.

- As we shall see, a layout can be rather complex and storing large layout as an attribute of every file can be prohibitively expensive (compare with Lustre LOVEA story). To address this, a notion of *layout formula* is introduced. A layout formula takes some parameters and produces a layout as result. Instead of storing a file layout together with the file, the file references a layout formula shared by a large number of files. To obtain file layout, a few parameters, including fid, are substituted into the formula. For example, instead of layout `L = {RAID0 over component files with fids CF0, CF1, ..., CF127 }` a formula `LFun(n, f, s) = {RAID0 over component files with fids f, f + 1*s, f + 2*s, ..., f + (n-1)*s}` can be used. A particular file layout can then be described as `LFun(128, fid0, step0)`.

- Design of layout formulae is a critical condition of efficient striping implementation. The class of formulae must conform to the following requirements:
  - all standard RAID patterns must be supported compactly;
  - parity declustering must (see below) be supported;

    - formulae should work efficiently with small FLDB (Fid Location Data-Base) and CLDB (Container Location Data-Base), in the face of object migration.

- Concurrency control. Concurrent conflicting operations on a striped file must preserve consistency. Striping with redundancy introduces dependencies between data updates: parity and corresponding data must be updated atomically. T1 does not provide concurrent access to exported block devices, but still have some concurrency, *e.g.*, during recovery.

- Meta-data can be striped too (this is traditionally called *clustered meta-data*). This is outside of T1 scope.

- Failures. Network partition between a client and one of the data servers can happen. Partition between servers can happen. Partitions can be asymmetric. Data server can crash and restart losing all its volatile storage. Data server can crash forever. Data server can lose one or several of its drives. There can be latent media failures. Striping maintains data availability in the presence of certain combinations of the above failures. The strategy here relies on three mechanisms:
    - when a server becomes inaccessible or non-responsible for any reason, NBA (non-blocking availability) sub-system changes layouts of affected files so that all following writes use different set of servers to stripe data onto. This includes revoking layouts used by clients and updating layout data-base. NBA is based on liveness detection that allows quick detection of server unavailability instead of relying on timeouts;
    - if a server or a drive is lost permanently, then parallel to NBA actions, a *rebuild* is started. Rebuild restores lost data using redundancy, see below for details;
    - if a server restarts after a crash, distributed transaction manager restores consistency.

- Transactions. In the presence of redundancy, updates to data and parity blocks are dependent and has to be applied atomically, both in concurrency control sense (*i.e.*, other clients see either all or none of dependent updates, this aspect was briefly described above) and in failure sense (*i.e.*, after a failure either all or none of dependent updates are present in the system). Required atomicity is achieved by DTM (distributed transaction manager) by combination of caching, write-ahead logging, epochs and undo-based recovery, as described in the corresponding documentation.

- Rebuild. Rebuild restores lost data using redundancy. For each unit of lost information (a block in case of local RAID and a striping unit in case of SNS) other units from the corresponding parity group are read and the lost unit is recovered from them. The servers are said to be in a *degraded mode*, while rebuild is going on. Rebuild has to achieve following goals:
    - restore lost data quickly to reduce a possibility of a second failure during rebuild (for usual RAID5-style redundancy, second failure leads to unrecoverable data loss);
    - utilize throughput of as many servers and drives as possible;
    - keep fraction of total servers throughput consumed by rebuild low, to allow system to meet its performance goals while in degraded mode.

  The last two goals are clearly at odds and, as it turns out, place severe constraints on permitted layouts of redundantly striped files.

- Distributed spare. A lost unit of data, recalculated by rebuild, has to be stored somewhere. A distributed spare is a space reserved on each drive for purpose of storing rebuilt data. Allocation

of distributed spare blocks to rebuilt data is a non-trivial task, because of the basis requirements of no two stripe units of the same parity groups ever being placed on the same drive (or server). When a replacement for a failed drive or node is added to the system, contents of distributed spare are copied out (are they? Alternatively one can rely on space rebalancing mechanism to move some data to a new disk and use thusly freed space as a new spare.).

- Parity declustering. The method to keep rebuild overhead low is to stripe a file over more servers or drives than there are units in a parity group. In RAID5 every parity group contains blocks on every drive, which means that during rebuild every block of every survived drive has to be read. If, on the other hand, a file is striped with parity group of size G (*e.g.*, G - 1 data blocks and 1 parity block) over C servers (or drives), then to recover a particular lost unit of data, reads on only G - 1 out of survived C - 1 servers have to be done. That is, (G - 1)/(C - 1) fraction of total throughput of C servers is consumed by rebuild. To achieve the companion goal of utilizing the throughput of as many servers as possible, parity groups must be scattered across all C servers uniformly, but in a way that allows a compact description (this sounds somewhat confusing: one wants to decrease the ratio of throughput consumed by a rebuild to a total throughput of a system, while scattering rebuild IO over as many servers or spindles as possible). Together these ideas form the basis of *parity declustering* methods, described in papers in `Dropbox:techleads/storage-technical-resources/sns`.

- Front-end. T1 front-end is an IO client to data servers. Redundant striped IO is more involved than non-redundant (RAID0) IO that Lustre clients do, because parity has to be kept consistent with data. One of the following methods is traditionally used for writes:
    - read-modify-write: before updating some blocks in a parity group, a client reads old blocks contents. It then sends (as a part of the same distributed transaction) a new block data to the target servers, and a difference (XOR) of new and old data, XOR-ed together to the server storing corresponding parity block. This server applies the difference to the parity (XOR again);
    - reconstruct write: before updating some blocks, a client read contents of all other data blocks in the parity group. It then sends new data to the data servers, and XOR of all data blocks in a parity group to the parity server. The latter replaces parity with the new value;
    - full stripe write: this is special case of previous mode, where all data blocks of a parity group are overwritten. No additional reads are necessary. When applicable, this method is most efficient.

- Back-end. Data and parity block placement must be investigated. The problems of component file pre-creation (or, alternatively, CROW) are known since Lustre times.

- Benchmarking. The questions of optimal stripe size, layouts, data placement, *etc.* cannot be solved without systematic benchmarking and simulation.

- Multiple-failure handling. Usual RAID5 method protects against a single failure. C2 SNS and local RAID must be configurable to maintain reliability in the presence of multiple simultaneous failures.