Introduction

This specification describes a mechanism for splitting a large byte stream into blocks based on their content, and a mechanism for organizing those blocks into a balanced tree.

The general technique has been used by various systems such as:

- Perkeep
- Bup
- RSync
- Low-Bandwidth Network Filesystem (LBFS)
- Syncthing
- Kopia

…and many others. However, the exact functions used by these systems differ in details, and thus do not produce identical splits, making interoperability for some use cases more difficult than it should be.

The role of this specification is therefore to fully and formally describe a concrete function on which future systems may standardize, improving interoperability.

Notation

This section discusses notation used in this specification.

We define the following sets:

- U_{32}, The set of integers in the range $[0, 2^{32})$
- U_8, The set of integers in the range $[0, 2^8)$, aka bytes.
- V_8, The set of *sequences* of bytes, i.e. sequences of U_8.
- V_v, The set of *sequences* of *sequences* of bytes, i.e. sequences of elements of V_8.

All arithmetic operations in this document are implicitly performed modulo 2^{32}. We use standard mathematical notation for addition, subtraction, multiplication, and exponentiation. Division always denotes integer division, i.e. any remainder is dropped.

We use the notation $\langle X_0, X_1, \ldots, X_k \rangle$ to denote an ordered sequence of values.

$|X|$ denotes the length of the sequence X, i.e. the number of elements it contains.

We also use the following operators and functions:

- $x \wedge y$ denotes the bitwise AND of x and y
- $x \vee y$ denotes the bitwise OR of x and y
- $x \ll n$ denotes shifting x to the left n bits, i.e. $x \ll n = x2^n$

- $x \gg n$ denotes a *logical* right shift – it shifts x to the right by n bits, i.e. $x \gg n = x/2^n$
- $X \parallel Y$ denotes the concatenation of two sequences X and Y, i.e. if $X = \langle X_0, \ldots, X_N \rangle$ and $Y = \langle Y_0, \ldots, Y_M \rangle$ then $X \parallel Y = \langle X_0, \ldots, X_N, Y_0, \ldots, Y_M \rangle$
- $\max(x, y)$ denotes the maximum of x and y.

Splitting

The primary result of this specification is to define a family of functions:

$\mathrm{SPLIT}_C \in V_8 \to V_v$

. . . which is parameterized by a configuration C, consisting of:

- $S_{\min} \in U_{32}$, the minimum split size
- $S_{\max} \in U_{32}$, the maximum split size
- $H \in V_8 \to U_{32}$, the hash function
- $W \in U_{32}$, the window size
- $T \in U_{32}$, the threshold

The configuration must satisfy $S_{\max} \geq S_{\min} \geq W$.

Definitions

The "split index" $I(X)$, is either the smallest integer i satisfying each of:

- $i < |X|$
- $S_{\max} \geq i \geq S_{\min}$
- $H(\langle X_{i-W+1}, \ldots, X_i \rangle) \mod 2^T = 0$

. . . or $\max(|X| - 1, S_{\max})$, if no such i exists.

We define $\mathrm{SPLIT}_C(X)$ recursively, as follows:

- If $|X| = 0$, $\mathrm{SPLIT}_C(X) = \langle\rangle$
- Otherwise, $\mathrm{SPLIT}_C(X) = \langle Y \rangle \parallel \mathrm{SPLIT}_C(Z)$ where
- $i = I(X)$
- $N = |X| - 1$
- $Y = \langle X_0, \ldots, X_i \rangle$
- $Z = \langle X_{i+1}, \ldots, X_N \rangle$

Tree Construction

TODO

Rolling Hash Functions

The RRS Rolling Checksums

The `rrs` family of checksums is based on a rolling first used in rsync, and later adapted for use in bup and perkeep. `rrs` was originally inspired by the adler-32 checksum. The name `rrs` was chosen for this specification, and stands for `rsync rolling sum`.

Definition

A concrete `rrs` checksum is defined by the parameters:

- M, the modulus
- c, the character offset

Given a sequence of bytes $\langle X_0, X_1, \ldots, X_N \rangle$ and a choice of M and c, the `rrs` hash of the sub-sequence $\langle X_k, \ldots, X_l \rangle$ is $s(k, l)$, where:

$a(k, l) = (\sum_{i=k}^{l} (X_i + c)) \mod M$

$b(k, l) = (\sum_{i=k}^{l} (l - i + 1)(X_i + c)) \mod M$

$s(k, l) = b(k, l) + 2^{16} a(k, l)$

RRS1

The concrete hash called `rrs1` uses the values:

- $M = 2^{16}$
- $c = 31$

`rrs1` is used by both Bup and Perkeep, and implemented by the go package `go4.org/rollsum`.

Implementation

Rolling

`rrs` is a family of *rolling* hashes. We can compute hashes in a rolling fashion by taking advantage of the fact that:

$a(k + 1, l + 1) = (a(k, l) - (X_k + c) + (X_{l+1} + c)) \mod M$

$b(k + 1, l + 1) = (b(k, l) - (l - k + 1)(X_k + c) + a(k + 1, l + 1)) \mod M$

So, a typical implementation will work like:

- Keep $\langle X_k, \ldots, X_l \rangle$ in a ring buffer.

- Also store $a(k,l)$ and $b(k,l)$.
- When X_{l+1} is added to the hash:
- Dequeue X_k from the ring buffer, and enqueue X_{l+1}.
- Use X_k, X_{l+1}, and the stored $a(k,l)$ and $b(k,l)$ to compute $a(k+1,l+1)$ and $b(k+1,l+1)$. Then use those values to compute $s(k+1,l+1)$ and also store them for future use.

Choice of M

Choosing $M = 2^{16}$ has the advantages of simplicity and efficiency, as it allows $s(k,l)$ to be computed using only shifts and bitwise operators:

$s(k,l) = b(k,l) \vee (a(k,l) \ll 16)$