Design for Historical Journal Consumption

1. Introduction

This document explains design of an API for historical journal consumption (History API). It facilitates Glusterfs with the ability to detect file-operations happened in past by scanning the back-end(brick-level) glusterfs journal (changelog).

2. Design Comparison

Old Design

- Changelogs produced in one perfectly running session were used to maintain linkages, ie. Rolling-over changelog will have time-stamp of just before rolled-over changelog.
- Involved large number of xattr file-operations during linkage traversal, which was un-optimal.
- Required O(n), (where n is number of changelos in the list), time to identify the end changelog for the given start-end time interval.

New design

- List of changelogs produces in one perfectly running session are stored in htime file which also holds necessary information about the session start and end time.
- Involves fixed sized seeks to identify N'th changelog in the list.
- Requires O(log n), (where n is number of changelogs in the list), time to identify the end changelog for the given start-end time interval.

3. Architectural Design

3.1 APIs

- **Register**
  - int **gf_changelog_register** (char *brick, char *scratch_dir, char *log_file, int log_level, int max_reconnects);
  - This is the first call an application should invoke. A successful return from this call signifies that the application can now use other APIs (including history APIs) to access changelog data.
  - *@brick* - brick path (export directory) for which the changelog is needed.
  - *@scratch_dir* - working directory.
  - *@log_file* - log file path to log library messages.
  - *@log_level* - log level (defaults to INFO).
  - *@max_reconnects* - number of retries before giving up (0 for no retries).

    - The Register API returns zero on successful register, which means that the consumer can successfully make use of further historical journal consumption APIs.
    - Return -1 on failure(error).

- **History**
    - int **gf_changelog_history**(char* changelog_dir, unsigned long start, unsigned long end);
    - This API expects changelog path where the glusterfs journals are stored, start and end time-stamps (Unix epoch time), which stands for the duration between which the historical journals are expected to be consumed.
    - @changelog_dir – Directory where the brick changelogs are stored.
    - @start – Unix time-stamp, representing start time for history changelog consumption.
    - @end - Unix time-stamp, representing end time for history changelog consumption.
    - Returns zero on success, which means a consumable set of history changelogs are available for the given start-end time duration.
    - Returns -1 on failure(error), which means history changelogs are not available for the given start-end time duration. Reasons could be, changelog/brick crashed or changelog was off for some time in the given start-end time duration.
    - The API is non-blocking in nature.

- **history_scan**
    - ssize_t **gf_history_changelog_scan** ();
    - Scan and generate a list of new changelogs produced after parsing the total history_changelogs in groups of N (configurable based on performance improvements, default is 3). Invoking this multiple times one after the another results in refreshing the consumable history list.
    - The API returns zero, when further scan() calls are required to get the complete list of parsed history changelogs.
    - The API returns -1 on failure(error), which means either the current scan failed or couldn't scan the consumable changelogs though they were available.
    - The API returns 1, when no more consumable changelogs can be produced as the changelog_history_list has exhausted.
    - In any of the cases when the scan API returns non zero value, it should not be called for this history request.

- **next_history_change**
    - ssize_t **gf_history_changelog_next_change** (char *bufptr, size_t maxlen);

- Get the next changelog file from the set scanned by history_scan.
- @bufptr - buffer to store the changelog file path
- @maxlen - length of @bufptr
- This API on subsequent calls provides path of each changelo (from the list prepared by history_scan), which can be used by the consumer (say Geo-replication) to use it the way it desires.
- The API returns the string length of the changelog file path (stored in @bufptr).
- The API returns zero which signifies end of change list, and time call next history_scan.
- The API returns -1 on failure(error).

- **done_history**
  - int **gf_history_changelog_done** (char *file);
  - Invoking this API results in the history changelog (@file) to be noted as processed. Normally, this API is invoked after the history changelog is processed.
  - @file - the changelog file path to be marked as processed.
  - This API when called with the changelog file location (which was updated by “get_next_history”)and moves the consumed changelog to a consumed directory (.processed). This API must be called after the consumption of the changelog at location “file” is done completely by the involved consumer (say Geo-replication).
  - Returns 0 on success, which means the consumed changelog is moved successfully.
  - Returns -1 on failure(error).

3.2 Changelog design modification

- **htime**
  - Htime is a directory maintained at the same location where journals(changelogs) are stored.
  - It contains files with the format “HTIME.<time-stamp>”. Each htime file stores information about every uninterrupted glusterfs session where changelog was ON.

4. Detailed Design

4.1 Register to libgfchangelog to use history API (history_register)

- Get the changelog directory where changelogs are stored .
  - Either by dumping the changelog path in the socket connection between libgfapi and changelog translator. (OR)
  - Let gsyncd give the changelog path from which history changelogs should be consumed.

- Create and initialize history_directory location in “priv” with “.history/.current”, “.history/.processing” and “.history/.processed” for changelog parsing, processing and consuming purposes respectively.

4.2 Maintenance of HTIME of each changelog session (changelog)

- HTIME is a directory maintained by changelog translator which contains files in format “HTIME.<time-stamp>”.
- Each file (say HTIME.T1), will have a “time-stamp:number_of_changelogs_in_that_session” in its extended attribute (say T2:N). T1 stands for the start time of the changelog, T2 stands for the last successfully rolled-over changelog time-stamp and N stands for number of changelog file location entries in the file (HTIME.T1). N is used for searching the desired time-stamp in the HTIME file optimally.
- The contents of HTIME.T1 file would be paths of successfully rolled over changelogs in rolling sequence seperated by NULL.
- Example, lets assume that we have the following changelog set available in the current changelog directory [C.t0, C.t1, C.T0, C.T1, C.T2, C.T3, C.T4, C], where C.t0 stands for CHANGELOG.t0. The scenario is that changelogging was interrupted at sometime after t1 and before T0 but now running perfectly.
  - If the previous session interruption was a crash, then changelog C.T0 is partially written rolled-over changelog and the htime file for the currently running session would be HTIME.T0 with extended attribute as “T4:4” and contents as [C.T1, C.T2, C.T3, C.T4].
  - If the previous interruption was a stop, then changelog C.T0 is consumable and hence htime file for this session would be HTIME.<current_timeofday> with extended attribute as “T4:5” and contents as [C.T0, C.T1, C.T2, C.T3, C.T4].

4.3 Availability of changelogs for given time-stamp (history)

- Identifies the HTIME file that has the changelog information about the duration (between start and end) which was inquired to history.

4.4 Group and process the changelogs (history)

- Reads the HTIME file identified from section 3.3, searches optimally (O(log n), where n is number of changelog path entries in the HTIME file) the HTIME file for changelogs falling under the time interval the consumer is interested in.
- Spawns N (N is configurable depending on performance) threads to parse the changelogs from brick
- Each thread parses and moves the parsed consumable file to “.history/.processing”.
- Wait for threads to join and parse next N relevant changelog files.

4.5 Scan and provide the list for consumption (history_scan)

- Readdir the ".history/.processing" and update the tracker with files available for consumption.

4.6 Provide path of each conumable changelog in the list (get_next_history_change)

- Read tracker and provide the path of consumable changelog path (located at .history/.processing") to consumer.

4.7 Move and mark the consumed changelos as "done" (history_done)

- Move the consumed changelog file from ".history/.processing" to ".history/.processed"

5. API usage and control flow

- Consumer calls "register", which sets the needed environment for history API usage.
- On successful return of "register" consumer calls "history", which checks the availability of history changelogs for given start-end time interval and spawns a thread that starts parsing changelogs in parallel.
- On successful return of "history" consumer should keep calling "history_scan" till return value is -1. Each call lists set of consumable changelogs.
- On non-negative return of "history_scan" consumer should call "get_next_history_change" to iterate over the list created by "history_scan" to get the individual changelogs for consumption.
- For each "get_next_history_change", consumer should call "done_history" to clean backlog of changelogs.

6. Assumptions

- Fixed path length of changelogs.
- Htime directory is non configurable and resides at the same location as that of changelogs.