Sanedit Implementation Notes

This document contains implementation details and notes. It should provide answers why certain design decisions were made.

1 Buffer

Text buffer implementation. It is a piece tree a piece table variant which stores the pieces in a red-black tree instead of a vector. To understand what a piece tree is lets start with a piece table.

Piece table uses two text buffers, original buffer which contains the original text and an add buffer which contains added text. The original buffer is read only and the add buffer is append only. This means that you can index into these buffers as the text never changed. Piece table stores pieces which are an offset + length pair. The pieces then reference the buffers and form the buffer contents. This means the buffer content is not contiguous in memory. For more information on piece tables

Then a piece tree is a piece table that stores the pieces in a red-black tree. The nodes contain left subtree length (in bytes) so we can search for a bytes offset in $O(\log(n))$ where n is the number of pieces in the tree. Rust provides easy access to cow functionality using `Arc::make_mut` which is used in the tree. It allows us to take lightweight snapshots of the tree, which can then be restored. This is a built in solution for undoing changes. The snapshots are lightweight as the trees can share nodes, new nodes are only created if snapshots referencing the node still exist.

For more information on piece trees see Improving the AbiWord's Piece Table,
VS Code Text Buffer Reimplementation

This implementation operates on bytes and does not guarantee that the content is valid UTF-8. The bytes are decoded to UTF-8 when needed using DFA method by Bjoern Hoehrmann.

Line counts could also be cached to nodes (like left subtree byte length) to provide a fast line search, but this implementation opts of that so we do not need to read the whole file on load. This means opening large files should be as fast as opening small ones.

1.1 Piece tree is good at

- Random insertion and deletion
- Large files
- Easy undo implementation
- Easy text position tracking by indexing into backing buffers

1.2 Piece tree is bad at

- Sequential insert and delete are not $O(1)$ amortized time.
- Performance deteriorates over time as more pieces are in the tree
- Complex implementation because of red-black trees

1.3 Marks

Marks track a text position through edits. Piece tree makes marks easy to implement, because the add buffer or original buffer never change we can point to them using offsets ie. `(BufferType::Add, 500)`.

To create a mark we find the piece responsible for that position of the buffer and get the buffer type and appropriate offset into the buffer the piece references. Now the mark can point to its new position as long as the piece that holds that offset is still in the tree even through edits.

To find the marks position after edits we can iterate over all the pieces and find out if the mark is included in them. If it is we can return the pieces position in buffer + appropriate offset the mark is pointing to.

If the piece is deleted the mark can not find its position again.

1.4 Multicursors and piece tree

1.4.1 Problem

Suppose we are inserting a character at m positions (cursors) when a character is inserted at each position, it is appended to the add buffer m times where m is the number of cursors. This is relatively bad in itself. But the real problem comes with the next character, now we add the character m times to the add buffer again, but notice, in single cursor case the characters would be sequential in the add buffer but in this case they are not. This means that the pieces referencing this part of the buffer cannot be appended to at all, we must create a new piece. So if we insert n characters at m points, it would produce $m \cdot n$ new pieces. This would deteriorate performance way too quickly.

1.4.2 Solution

Problem is solved by exposing a new function for multi insert. It takes the text and multiple positions to insert to. Now we can simply append the character to the add buffer and reference it multiple times. This however creates an unexpected problem, previously all of the pieces were unique in a sense that only one piece referenced to a slice of add buffer. This assumption was used in the mark implementation. To identify the pieces from one another we need to add a `count` field. The `count` field is the index of the multiple positions array we got at the function call. Now the pieces can again be uniquely identified and the mark implementation is happy.

1.5 Reading buffer in a different thread while making changes in another

The buffer supports reading it while other thread is making changes. This is useful because now the buffer contents can be used with asynchronous jobs. For example we could send a job to save a large buffers contents on the background or syntax highlight the buffer.

The functionality is achieved using copy-on-write provided by `Arc::make_mut`. This ensures the copies on different threads share most of the tree. Original buffer is already thread safe as it is never modified. Add buffer is implemented by allocating buckets of size 2^x that never move. Preallocating the buckets has a problem when inserting text to the add buffer, there can be a gap, meaning the contents are not contiguous in memory. To ensure we do not slice non-contiguous data while reading the tree, the pieces are always separate when appending to the add buffer cannot be done contiguously.

1.6 Searching

Searching the buffer should be fast and with possibly gigabyte sized buffers searching should be supported forwards and backwards. The search algorithm used here is Boyer Moore. Boyer Moore is good at searching text that has a large alphabet and not much repeating. It runs in sub-linear time in the best case, and the performance increases as the length of the searched term increases.

2 Cli

Commandline interface to launch the editor.

3 Editor

Editor implementation itself.

The editor server accepts connections through TCP and unix domain sockets. The connections are considered to be a single window in the editor. Messages that are passed from the client to the server are described in Section 4.

3.1 Structure

The editor is structed into a few subdirectories.

Directory	**Usage**
common	functions that do not directly change the editor state
actions	functions that directly change the editor state
draw	used to draw all the things for the client
editor	holds the editor state model and functions to ease operating on it
server	handles async with tokio, handles all the client connections, jobs, etc.

3.2 Threading

The editor itself is run on a single thread and tokio runtime is used along with channels to provide async functionality. Async is used to accept and handle the client connections and to run jobs that the editor requests.

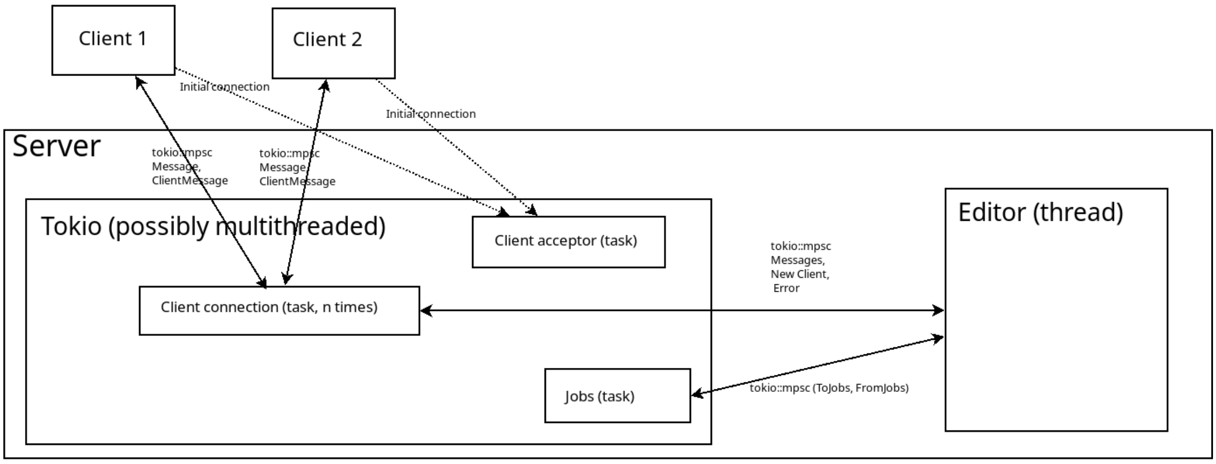

Figure 1: Editor threading.

3.3 Jobs

The editor can ask tokio to run jobs. Jobs are just functions that can produce gradual output. Jobs were created to run long running processes in the background whilst having a responsive editor. Best usages for jobs are shell commands, large buffer saving, listing files can be implemented as jobs.

4 Messages

Messages sent between editor and clients.

5 Terminal-client

Terminal client for the editor. Its job is to send keyboard and mouse events to the editor and draw the screen.

6 Ucd

Unicode character database, used to implement unicode properties, grapheme breaks etc. This is generated using the ucd-generate utility.

7 Parser

Requirements:

1. Easy to modify at runtime
2. Fast enough to parse on every keystroke
3. Easy to maintain grammars

7.1 Why not one of the existing solutions

Regex syntax highlighters are not easy to maintain or understand.

Tree sitter is good but we need support for larger than memory files too also don't like the hassle of compiling parsers for individual languages. I want to change the grammar at runtime and see changes immediately.

Luas LPEG library would be a good fit, but I don't want to include a language just for one library.

7.2 PEG

Decided to implement a parser using PEG rules.

7.3 Parsing

7.3.1 Pikaparsing

The first parsing strategy tried was pikaparsing pikaparsing. The memory usage was too high and performance was not that good.

7.3.2 LPEG & GPEG

LPEG library was a good fit and should be more performant than pikaparsing. Also found an improved version of LPEG called GPEG that also supports incremental parsing. This is a good reference point for our own implementation of something similar.

7.3.2.1 UTF8

Byte sequences can naturally represent also UTF8. Also UTF8 Ranges are added to ease usage for example to match any UTF8 characters `[\u0000..\u10ffff]`.

`\u` enables UTF8 support: If an UTF8 point or range is used in negated sets `[^\u0020]` the negated set is expanded to `[\u0000..\u0019] / [\u0021..\u10ffff]`, but the same negated set if written `[^\x20]` is expanded to `[\x00..\x19] / [\x21..\xff]`

Note that all the sets are inclusive.