FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

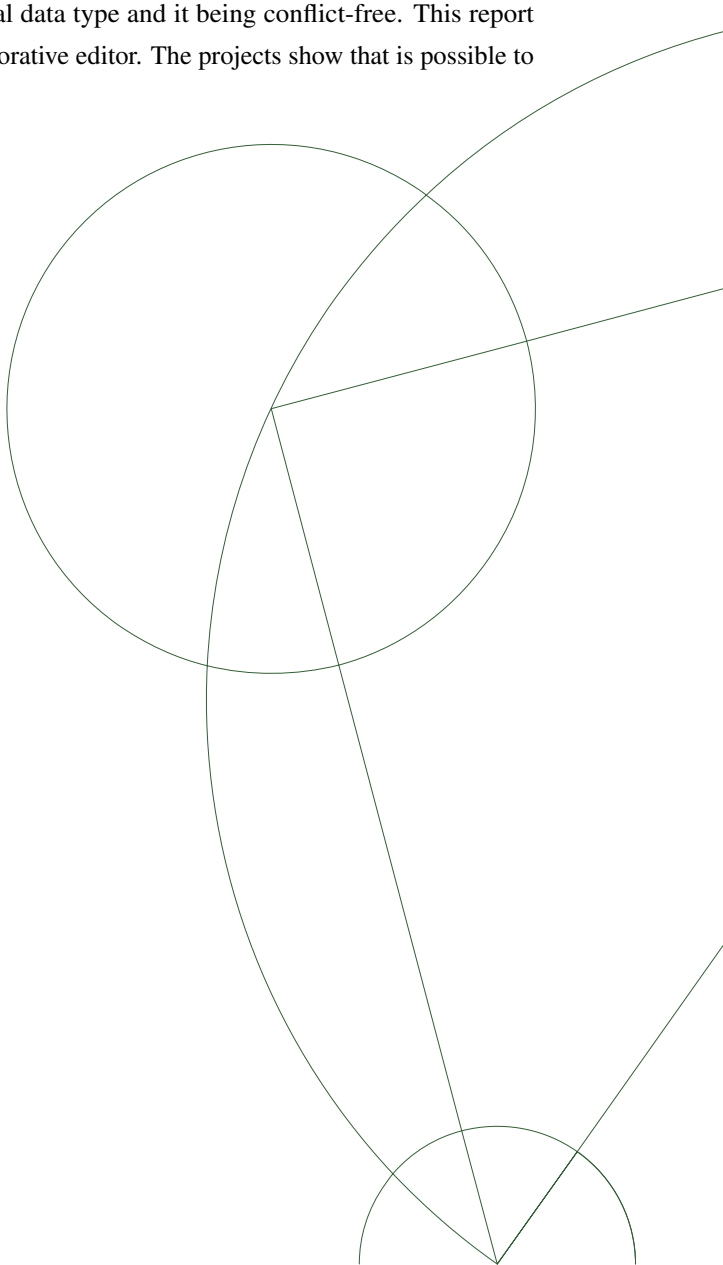
BSc Project

Collaborative editor with Conflict-free Replicated Data Type in Rust

Oliver Malling Laursen mdr335@alumni.ku.dk
Rune Taj Clemens Petersen ckq931@alumni.ku.dk

Abstract

Optimistic approaches to share data is gaining attention with database solutions like Riak and collaborative editors. And it has its properties to back it with immediately response of the local data type and it being conflict-free. This report implements a CRDT, a network stack and a terminal to make a collaborative editor. The projects show that is possible to integrate and have functional editor.



Dato: 1st July 2018

Contents

Chapter 1

Introduction

Collaborative editing allows users to work together on the same document simultaneously. A traditional way to handle this problem is a centralized client-server solution where the server owns the data type and manages updates from the users send to it. While this approach ensures no conflicts arise, bottlenecks in request can quickly become and issue, as well as data loss should the server fail.

A Conflict-free Replicated Data Type (CRDT) and P2P network poses an alternative to this model. This model replicates the data type across all nodes and encapsulates Strong Eventual Consistency, meaning that all replicas will reach the same state at some point.

CRDTs offers a decentralized solution that rely on optimistic approach and is free of conflict. This, in turn, can also eliminate the bottleneck problem residing in a centralized model and since each node has its own replica, data loss and connectivity is minimized.

This thesis showcase how the Ditto frameworks CRDTs work with a transport layer and a collaborative editor that we've build to illustrate the benefits and workings of a CRDT in collaborative editing.

Chapter 2

Background, analysis and theory

In this section we briefly touch the history of CRDTs and give a brief introduction to Rust. We will go through the different categories and groups of CRDTs, and look into the theory of the CRDTs we are using in this project, and some of the important features of Rust

2.1 Problem statement

Collaborative editing allows users to work together on the same document simultaneously. A traditional way to handle this problem is a centralized client-server solution where the server owns the document and manages updates from the users for the document. The side effects of the approach - albeit simple - is high latency which can be caused by bottleneck. Another drawback for this solution is that all data is stored in a single location, which means if the server faults, the whole system stops working.

We explore a decentralized alternative, using a Conflict-free Replicated Data Type, wherein each user (node) holds their own copy (replicated) of the data type and rely on optimistic approach. Through example we establish a collaborative editor to illustrate this.

2.2 Collaborative Editor

As stated part of our problem is to implement a collaborative editor.
We define a collaborative editor to be in similarity to already existing ones such as, Google Docs [2] and Share LaTeX [13]. This is defined in broad strokes. These editors have among them; history backtrack, continuous saving, rich-text editing, etc.

We will be focusing on a minimal collaborative text-editor in style with Nano[5]. That is we establish a simple text-only, terminal-based editor with key-navigation.

2.2.1 Scope

The argument for a minimal editor is we want the collaborative editor to be a demonstration of our replicated data type: A means to an end.

2.3 Architecture

Traditionally a distributed data type, has been handled by a single node, the server, that contained the data type, and was said to be proven to comply with the CAP theorem. This server-client structure ensures that only one instance of the data type is correct and all the other node gets a copy of that data type. Because the server has per definition the correct value of the data type, we call this *Strong Consistency*. Strong Consistency is a model, that if one node gets an update, other node immediately get the same update. So all node see the same order of updates to the data. This can be seen as a large sequential system, that is good for fault tolerance but is really bad for performance. The serialization process is called *consensus*. This consensus is a well studied concept that does not scale very well [11]. One of the ways the serialization process is not performing well, is when a piece of data, like a character, needs a round trip to the server and back, before we can safely show the character on the screen. So in the case a client has a high latency to the server, the changes could take a long time to be reflected in the local value. An other drawback is that the server-client architecture, breaks down when the server breaks down. This single point of failure is critical because no data can be distributed and data might be lost.

Another architecture of the distributed data type is to set the network up in a peer to peer configuration. All updates to the value of the data type is distributed to all the other peers, in some sort of topological order. An approach in a peer to peer network is the *Eventual Consistency* (EC), where all nodes will converge to the same value, but its will roll-back when there is a conflict. This approach tends to get complicated because the roll-back also has to be distributed.

What Mark Shapiro suggests *Strong Eventual Consistency* (SEC) where as soon as the last update arrives we will have consistency (no role-back). The CRDT delivers such a SEC system. You can do your updates locally and propagate the updates to other nodes. Deterministic rules for conflicts so that those conflicts can be resolved immediately and equally on all nodes. This concept is a solution to the CAP problem, that was coined as a theorem by Eric Brewer [1] and is an acronym for Consistency, Availability and Partition tolerance. This have the effect on the editor with local updates, have an acceptable responsiveness and the SEC ensures that there are no conflicts that will render the typed characters not valid.

2.4 CRDT

Conflict-free Replicated Data Type (CRDTs) are a work of Mark Shapiro et al. and is a data type that can be distributed across network. It has the properties of strong & eventual consistency. It has two ways of updating; either by operations performed sequential (operation-based) or by sending an entire state (state-based).
What CRDTs are trying to solve is the bottleneck that comes with strong consistency and the conflicts that occur when there's concurrent updates in the eventual consistency solution. There exists different basic types of CRDTs such as; sets, counters and integers which can be combined to compose more advanced CRDTs.

2.4.1 The mathematical model of the CRDT

In the article "Conflict-free Replicated Data Type" by Shapiro et al., they generalizes all these different types of CRDTs into one mathematical model [12]. An state based object is a tuple, and all the states $s_i \in S$ are called its payload. The initial state is s_0, and q, u and m are the methods `query`, `update` and `merge` of this object.

$$(S, s^0, q, u, m, (t, u), P)$$

Additionally the operation based can work together with the State based CRDT. This addition adds the tuples of methods (t, s) where t is a side-effect-free prepare-update method and u is an effect-update method. The set S is lattice that is equipped with partial order $\leq$, so that the S is actually in a total order. With this order, the lattice will always have an Least Upper Bound (LUB) that is also called *join*. So the S in the object is a *join semi-lattice*. The LUB means that we can take any two values of S and compare them to get the greatest of the two. The state CRDT gets the idempotent, communicative and associative properties from operating on the join semi-lattice. That means a state can be merge multiple times without any effect, it can be merged in any order, and operation can be grouped in the state for later merge.

2.4.2 The sequential CRDT

There are different types of CRDTs; Grow-only Counter, grow-only set, Positive Negative counter, Two-Phase Set, Last-Write-Wins-Element-Set, Observed-Removed Set, and the Sequence CRDTs. The type of CRDT that are used for text editors, are the sequence CRDT. The sequential CRDT is basically a distributed vector of elements, where the $s \in S, s = (id, element)$. The *identifiers* (id) can either be of a fixed or variable size. Implementations of the fixed size sequential CRDTs are WOOT [7], RGA [9]. The fixed size id typically needs a tombstone mark for deletions of characters. This The variable id class of CRDT includes Logoot [14] and LSEQ [6], and requires no tombstones to delete characters. Treedoc uses both a variable id and tombstones [8]. The LSEQ id forms a tree structure by having the id with an vector of indices.

$$\texttt{element} : (\texttt{id}, \texttt{label}) \tag{2.1}$$

$$\texttt{id} : ([\texttt{index}], \texttt{site\_id}) \tag{2.2}$$

An example of one site producing a tree of characters is shown in figure 2.1. The tree starts out by having two elements: (([0],1),begin)(([99],1),end). It can be produced if the same site creates string "aefg" and there after moves the cursor to between the a and e, to insert b, c and d. The illustration of this example is shown in figure 2.1 does not include the site_id, but illustrates how the string "abcdefg" is build.

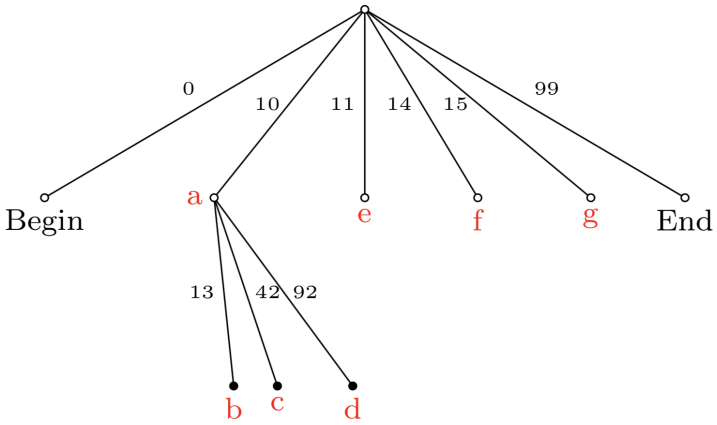


Figure 2.1: LSEQ tree structure example

The id is thereby unique and the set *S* has a total order, or the properties of a join semi-lattice.

2.5 Ditto's Text CRDT

The Rust framework Ditto's [10], have implemented several CRDTs. One of those CRDTs is `Text` type that is an implementation based on LSEQ. Ditto only implements the non-network part of the CRDT, so when considering this framework the network stack is not locked in. The two points that needs implementation to function is,to send ops/state from one site to another, and to assign a site id to each site.

2.6 The implementation language

Since performance and reliability is some of the properties for choosing a CRDT, the same properties should be found in the language we would choose for the implementation of the editor. The language Rust is a fine candidate for these criteria. Its is a system language like C but some of its most valued features is the focus on data integrity and security, which makes it fast and reliable. Rust compiler have an ownership of variables that enables the compiler to know when to free resources, so there is no loner a need for a garbage collector.

2.7 The network of the application

The network stack can be divided up into several layers, as in the OSI model [15]. Where for our consideration the network layer is kind of given, since other solutions that the IP is considered to have a very narrow market [3]. The transport layer is dependent on the network layer in terms of the the IP we have UDP, TCP, WebSockets. These are edges in a graph that can have several topologies. Some of these topologies are fully connected, circular, line, star graph.

2.7.1 Analysis of the transport layer

User Datagram Protocol (UDP) is a message protocol implemented on the the Internet Protocol (IP). It is very light weight but it also have no guaranties of delivery and order of delivery. The UDP would work well with the state based CRDT since, the merging of states are idempotent and would catch up if a package were to be lost. The operation based CRDTs requires that all messages are delivered and in the right order. These two features are not a part of the UDP, so if chosen, should be implemented on top.

An other protocol for the IP network is the Transmission Control Protocol (TCP). The TCP creates a bidirectional stream between two IP addresses, and it ensures that the packages are delivered and in the same order as they are send. This enables us to send CRDT operations right out of the box. The receiver of each stream are blocking and are therefore put in their own thread. An alternative to using the TCP or UDP directly is to use an event driven framework to multiplexer between the connections. There are two major frameworks that can offer this, `mio` and `tokio`. These are non-blocking so no threads are needed maintain a connection.

2.7.2 Analysis of the network topologies

A typical server-client based network has a star structure where the server is the center connecting to all the clients. This topology is sensitive to any errors happening on the server side, since that would affect all the other clients. A more resilient approach is the peer to peer, where the data is distributed by means of a CRDT. We can consider the fully connected graph where we either can share the CRDTs state or the operations of the CRDT.

We could also make a ring topology where each node sends the state or operation to the next node in an ordered list. This list could be a CRDT of the vector type, to ensure that the vector with the IP addresses would converge to the same content of the vector.

The drawback regarding the ring structure would be the worst case latency could be quite high, in the case of a larger state getting passed around the ring with a hundred nodes. Even if it was operation getting passed around, it could take a substantial amount of time, and the user would lack responsiveness from the other users.

Chapter 3

Design

In this section we define the scope for the editor and consider the design choices based on our analysis. We look at the choices for the transport layer, CRDT and the collaborative editor.

3.1 Scope

We scope our collaborative editor we set up some preliminary assumptions about the use of our editor. We assume a user base corresponding to the size of medium to large corporations (20 - 400 people) working simultaneously. We also assume a stable connection to the internet. We operate with the distance of communication to be countrywide.

3.2 Transport layer

The UDP is simple to set up in a fully connected graph, because we only have to spawn one thread for receiving packages. But because it lacks the two features we need for distributing operations, we are going to skip this protocol. Since we are making a terminal based application we are not going to use WebSockets because the TCP protocol is simpler by not being dependent on external crates and we are not using the connection to a browser. So to summaries the benefits of choosing the TCP protocol is, that we get the messages are delivered in order and we are sure the packages arrive.

In our case we want to have the lowest latency between each node, because it affects our user experience with regards to responsiveness from other users. The fully connected graph and an operation based distribution makes a simple system. But there is one issues with this solution. The system is not resilient to peers connecting to the graph of nodes from each side of the network. In that case the two new nodes would not have each other in their network address list, and would thereby ignore each other, resulting in a not fully connected graph. The solution to this problem contains a circular topology, and is covered in the reflection of this project.

The IP have two address implementations, the IPv4 and IPv6. We are only considering the IPv4 for this project, because we are focusing on the CRDT implementation and a possibly IPv6 expansion should be trivial.

3.3 CRDT

A sequence CRDT is well suited for implementing a collaborative editor and Ditto's `Text` data type built on top of LSEQ provides a solid foundation for just that. It can contain mutable strings and LSEQ is ensures a good allocation strategy.

So in our design we have chosen to opt for this solution and use Ditto's framework as a component to construct our collaborative editor with our transport layer.

3.4 Editor

The chosen domain for our editor is terminal-based, a simple text-only and key-navigation version.
We chose to keep the editor simple as to not take focus away from the underlying CRDT and transport layer.

This way we get a simplistic editor that can appropriately highlight the workings of the replicated data type.

Chapter 4

Implementation

4.1 Structure of the program

The main program have a blocking iterator to iterate over for all the events the keyboard and the network makes, instead of having a loop that pools over all the inputs. The keyboard reader and the stream receivers, seen in figure 4.1, are blocking, we therefor want to put those in there own threads. Instead of using variables with mutexes to communicate with the threads, we use channels. These two choices, brings us to a design where we use one channel for all inputs, that are wrapped in the enum `Source`, for the main programs to iterate over.

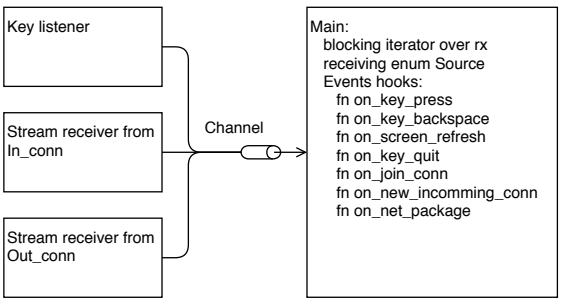


Figure 4.1: Transport layer illustration

The blocking receiver iteration block, unwraps the enum `Source` and go though all the event hooks, also called callbacks. These event hooks are is an implementation of the observer pattern in object oriented languages [4]. Each of these events are lists of functions, that are called when event is initiated. The trait `Events` are implemented on the structure where we have the methods we want to call, in this case the structure `crdt`. In this way we can have all the input from the keys and the network, directed to the CRDT for the appropriate method to update it. And since we have a serialized structure we now also have a hook that is activated each time any of the input updates the CRDT, so that we can update the screen with the value of the CRDT.

The editor we are implementing is divided into workspaces, as shown in figure 4.2. These workspaces consist of the main application, `editor`, and three libraries, the terminal functionality in `terminal`, the text string manipulation in `crdt` and the transport layer in `transport`.

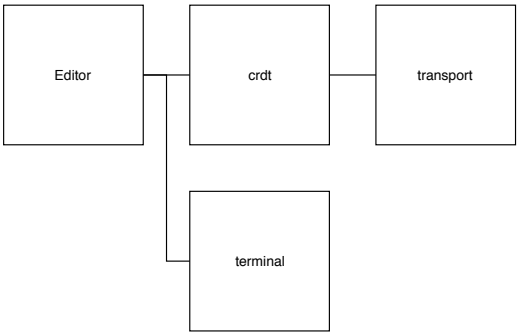


Figure 4.2: Workspaces of the editor

4.1.1 Data flow

Before diving into the details of the program, it would be beneficial to gain an overview of the data flow of when a key is pressed and how that information is handles at the local node, that we in figure 4.3 call node A, to an other node called node B.

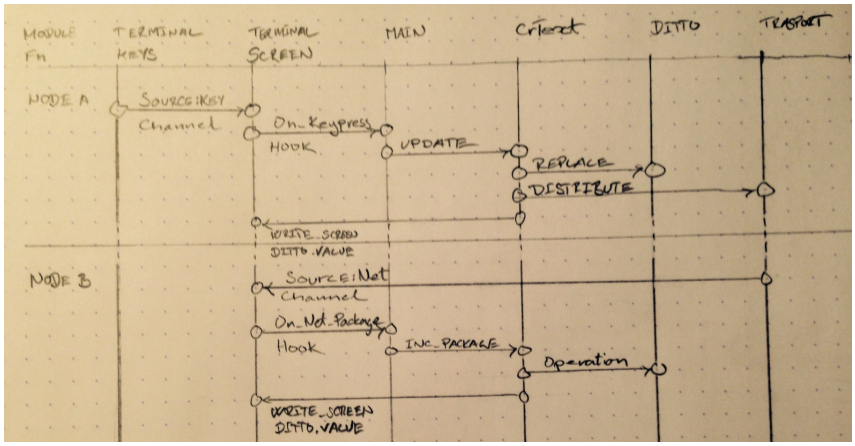


Figure 4.3: Data flow of a key pressed

The key stroke action is handled in the thread the `keys` is

4.2 The main function

The main function is where the terminal and the `crdt` module are stitched together. A new terminal `term` is generated by its method `new`. The struct has an `add_event_hook` method that takes an other struct, `Editor` figure 4.4 line 1, that implements the `Events` traits, seen on line 5. The `Editor` struct contains a `crdt` struct that takes the input argument, that is expected to be the file name of the configuration file and the main channel sender part, and a cursor.

```
struct Editor {
    crdt: crdt::CrText,
    cursor: Cursor
}
impl Events for Editor {
    fn on_key_press(&mut self, s:&str) {
        self.crdt.update(self.cursor.index,0,s);
        self.cursor.inc();
    }
...
fn main() {
    let args: Vec<String> = std::env::args().collect();
    let mut term = Terminal::new();
    let tx_ch = term.screen_tx_channel();
    term.add_event_hook(Editor {
        crdt: crdt::CrText::new( &args[1], tx_ch ),
        cursor: Cursor {index:0,col:0}
    });
    ...
    term.keys();
    term.screen();
    term.exit();
}
```

Figure 4.4: A1.3 - The `new` method and the `main_channel`

The last two things the main function does before the editor is functional is to call the two methods `keys` and `screen`. The `keys` method enables the keys and the `screen` is responsible for showing terminal screen and updating its content.

4.3 The terminal

The terminal module has two jobs, to read the keys that are typed and show the content of the editor on the screen. It is this module that creates the main channel, used for all events, and stores it in its struct. The transmitting part of the main channel is stored here to enable the network part of the editor also to use this channel so that the screen will update when a package is received.

To enable other modules to call their functions and methods, and the observer design pattern is implemented. This is done by defining a trait, that we here call `Events` also shown in figure 4.5 line 1. The `Terminal` struct then has a vector of pointers to the implemented functions.

```
pub trait Events {
        fn on_key_press(&mut self, s:&str);
        fn on_key_backspace(&mut self);
        fn on_screen_refresh(&self) -> (String,usize);
        fn on_key_quit(&self) {}
        fn on_join_conn(&mut self) -> Vec<std::net::SocketAddr>;
        fn on_new_incomming_conn(&mut self) -> Vec<std::net::SocketAddr>;
        fn on_net_package(&mut self, s:&str);
}
pub struct Terminal {
        screen: termion::screen::AlternateScreen<termion::raw::RawTerminal<std::io::Stdout>>,
        hooks: Vec<Box<Events>>,
        rx: std::sync::mpsc::Receiver<String>,
        tx: std::sync::mpsc::Sender<String>,
}
impl Terminal {
        pub fn new() -> Self {
                let (tx, rx) = mpsc::channel();
                Self {
                        screen: AlternateScreen::from(stdout().into_raw_mode().unwrap()),
                        hooks: Vec::new(),
                        rx: rx,
                        tx: tx,
                }
        }
        pub fn add_event_hook<E: Events + 'static>(&mut self, hook: E) {
                self.hooks.push(Box::new(hook));
        }
...
```

Figure 4.5: The `Terminal` struct and the `main_channel`

The `termion` package has an option to use an other screen buffer, called `AlternateScreen`, for the application, and is initialized as shown in line 5 of figure 4.5.

The main channel mentioned earlier in this section, is the channel that binds the application together. The application is thereby idle and only invoked when a message is received. The type of the message is defined by an algebraic data type or a enumerations (enum), with the enum values shown in figure 4.6. The Key contains a String with the character in it.

```
#[derive(Serialize, Deserialize)]
pub enum Source {
    Key (String),
    Quit,
    Join,
    Backspace,
    Stream,
    Net (String),
}
```

Figure 4.6: The `Source` enum

The other enum values are commands, except the Net value, that contains a serialized JSON string, with the network package. This means that the Termion key detection is done in the read key thread, for then to be serialized into a JSON string.

```
1     pub fn keys(&mut self) {
2         let stdin = stdin();
3         let ksx = self.tx.clone();
4         thread::spawn(move || {
5             for c in stdin.keys() {
6                 let package = match c.unwrap() {
7                     Key::Esc       => Source::Quit,
8                     Key::Char(c)   => Source::Key(format!("{}", c)),
9                     Key::Backspace => Source::Backspace,
10                    _              => Source::Key( "".to_string() ),
11                };
12                let st = serde_json::to_string( &package ).unwrap();
13                ksx.send( st ).unwrap();
14            }
15        });
16    }
17    pub fn screen(&mut self) {
18        let mut list:Vec<std::net::SocketAddr> = Vec::new();
19        for c in self.rx.iter() {
20            match serde_json::from_str(&c).unwrap() {
21                Source::Key(key_pressed) => {
22                    for hook in &mut self.hooks {
23                        hook.on_key_press(&key_pressed);
24                    }
25                }
26                ...
```

Figure 4.7: The `keys` and the `screen` methods

When there is send anything over the main channel, the screen needs to be updated. What is needed to be done is deserialized into our enum `Source` from figure 4.6. The enum value is found by the match, and the corresponding hook is activated. So in the case of a key press, the match deconstructs the string with the key pressed and passes it on to all the functions that are subscribed to that hook, as seen on line 21-24 of figure 4.7.

4.4 CRDT

The `crdt` module is dependent on the transport layer module and `ditto`, since the `ditto` module doesn't cover the transport layer. This enable a more modular approach to change or extent the network layer to other types of network.

```rust
pub struct CrText {
    obj: ditto::Text,
    node_id: u32,
    transport: transport::Transport,
}
impl CrText {
    pub fn new(filename:&str, tx_ch:std::sync::mpsc::Sender< String >) -> Self {
        let mut contents = String::new();
        File::open(filename)
            .expect("config file not found")
            .read_to_string(&mut contents)
            .expect("something went wrong reading the file");
        let config: Config = toml::from_str(&contents).unwrap();
        let tx_ch_clone = tx_ch.clone();
        let mut trans = Transport::new(config.ip, config.port, tx_ch_clone);
        match (config.other_ip, config.other_port) {
            (Some(ip),Some(port)) => {
                let listener = Some(trans.listener_addr);
                let listener_package = serde_json::to_string( &Package::Listener(listener))
                    .unwrap();
                let addr:std::net::SocketAddr = (ip + ":" + &port).parse().unwrap();
                trans.join(addr,tx_ch,listener_package);
            }
            (_,_) => (),
        }
        CrText {
            obj: Text::new(),
            node_id: config.node,
            transport: trans,
        }
    }
    ...
}
```

Figure 4.8: The `CrText` struct and its `new` method

The `new` method of the `CrText` struct returns a CRDT struct containing both a Ditto text CRDT and transport struct. It expects a file name for configuration the network and the main sending channel to return incoming data from each node. It is also in the configuration file we setup the unique node identification number for the `node_id` later used in Ditto's `site_id` value. The transportation layer is set up by first creating a listener address, for new nodes that want to connect to the network, and if it has an address to connect to it will join that network. When it joins a network it also send the address it is listening on, along with the join request.

```
    ...
    pub fn incomming_conn(&mut self) {
        let mut stream = self.transport.listener_receiver.iter().next().unwrap();
        let list = self.transport.other_listeners.clone();
        let p = &Package::Peers( list );
        let responce_of_ip_list = serde_json::to_string( p ).unwrap();
        stream.write( responce_of_ip_list.as_bytes() ).unwrap();
        let state_package = &Package::State( self.obj.clone_state() );
        let state_package_str = serde_json::to_string( state_package ).unwrap();
        stream.write( state_package_str.as_bytes() ).unwrap();
        self.transport.net_txs.push(stream);
    }
    ...
```

Figure 4.9: The `CrText` struct and its `incomming_conn` method

When a new node is connecting by joining, the host peer get the incoming connection in the transport layer. Since all our communication from our incoming thread is piped through our main channel in an enum, all the data in that enum should be serializable. The stream we are cloning, does not derive the necessary traits to fit in the enum serializable trait. So as an hack a new channel dedicated to only transport a Stream is created and is named `listener_sender` and `listener_receiver`. So the main channel actually function as a signal, to invoke the `incomming_conn`, and in that function we know that a package with a stream in it, is waiting in the `listener_receiver` to be moved to the `transport.net_txs` vector. The transport layer maintains a list of the listener addresses of all the connected nodes. This list of listener addresses is replied to the joining node. After that the CRDT text state is cloned and also replied to the joining node.

Before the listener addresses are pushed to its place, we reply to the stream with two packages. The first reply is the list of other listeners, so that the joining node have a list of peers to send its operation to. The second reply is the state itself, for the joining node to merge with its own empty state.

When the `CrText` structs `inc_package` gets a string from hook in the main function, it deserializes it because we know it is encoded in JSON. It deserializes to the enum `Package` that we then can match. So, from before we know that when a node joins the network it get the list of listeners addresses in the `Package::Peers`. The joining node unwraps the list of listeners, and joins each of them with an empty listener package. Right after the state was send the `Package::State` is then send and merged on the joining node.

```
    ...
    pub fn inc_package(&mut self, s:String) {
        let package = serde_json::from_str( &s ).unwrap();
        match package {
            Package::Op(op) => {
                self.obj.execute_op(op);
            }
            Package::State(state) => {
                self.obj = ditto::Text::from_state(state,Some(self.node_id)).unwrap();
            }
            Package::Listener(addr) => {
                match addr {
                    Some(address) => {
                        self.transport.other_listeners.push(address);
                    }
                    None => {}
                }
            }
            Package::Peers(ip_list) => {
                for addr in ip_list {
                    let ch = self.transport.tx_ch.clone();
                    let listener_package = serde_json::to_string( &Package::Listener(None)).unwrap();
                    self.transport.join(addr, ch, listener_package);
                }
            }
            Package::Leave(addr) => {
                let index = &self.transport.net_txs
                    .iter()
                    .position(|tx| tx.peer_addr().unwrap().to_string() != addr)
                    .unwrap();
                    self.transport.net_txs.remove(*index);
            },
        }
    }
}
```

Figure 4.10: The `CrText` struct and its `inc_package` method

4.5 Transport layer

The transport layers `Transport` struct implements a `new` method to initiate a thread for the TCP/IP incoming connections. It expects the IP address and port number that it needs to bind to, and the clone of the main-channel sending part, for the new connections. As mentioned earlier, when a new connection is created the stream is cloned and send through its own channel call `(listener_sender, listener_receiver)`.

```rust
impl Transport {
    pub fn new(ip:String, port:String, tx_ch:std::sync::mpsc::Sender< String >) -> Transport {
        let listener = TcpListener::bind( ip + ":" + &port )
            .expect("Transport::new() Could not bind to the given ip");
        let listener_addr = listener.local_addr().unwrap();
        let (listener_sender, listener_receiver) = channel();
        let tx_ch_clone = tx_ch.clone();
        thread::spawn(move || {
            for stream in listener.incoming() {
                match stream {
                    Err(e) => {
                        eprintln!("Transport::new() - Stream failed: {}", e)
                    }
                    Ok(stream) => {
                        let tx_ch_clone_2 = tx_ch_clone.clone();
                        listener_sender.send( stream.try_clone()
                            .expect("Transport::new() - clone failed") ).unwrap();
                        tx_ch_clone.send( serde_json::to_string(&Source::Stream)
                            .unwrap()).unwrap();
                        thread::spawn(move || {
                            incoming_conn(stream,tx_ch_clone_2);
                        });
                    }
                }
            }
        });
        Transport {
            listener_addr: listener_addr,
            other_listeners: vec![],
            tx_ch: tx_ch,
            net_txs: vec![],
            listener_receiver: listener_receiver
        }
    }
    ...
```

Figure 4.11

The function `incomig_conn` is spawned in its own thread when a new stream is created, since the `read` method of the stream is blocking. The stream is loaded in the `buf` and converted into a utf8 string, that is then wrapped in the Source::Net ENUM, and send through the main channel.

```rust
fn incoming_conn(mut stream: TcpStream, tx_ch:std::sync::mpsc::Sender<String>) {
    let mut buf = [0; 512];
    loop {
        let bytes_read = stream.read(&mut buf)
            .expect("Transport::new() Failed to read stream");
        if bytes_read != 0 {
            let s = str::from_utf8(&buf[..bytes_read]).unwrap().to_string();
            let st = serde_json::to_string( &(Source::Net(s)) ).unwrap();
            tx_ch.send(st).unwrap();
        }
    }
}
```

Figure 4.12

When a peer is connecting up to an other peer it uses the `join` method. When the peer connects to the network for the fist time, it needs to tell the network what its listener address is, so that other peers can connect to it.

```
impl Transport {
    ...
    pub fn join(&mut self,
            addr:std::net::SocketAddr,
            tx_ch:std::sync::mpsc::Sender< String >,
            listener_package: String) {
        let mut stream = TcpStream::connect(addr)
            .expect("Transport::join() - Could not connect to server");
        stream.write(listener_package.as_bytes())
                .expect("transport::distribute - Failed to write to server");
        self.net_txs.push( stream.try_clone()
            .expect("Transport::join - stream clone failed") );
        tx_ch.send( serde_json::to_string(&Source::Join).unwrap()).unwrap();
        thread::spawn(move || {
            let mut buf = [0; 512];
            loop {
                let bytes_read = stream.read(&mut buf)
                    .expect("Transport::join() Failed to read stream");
                if bytes_read != 0 {
                    let s = str::from_utf8(&buf[..bytes_read]).unwrap().to_string();
                    let st = serde_json::to_string( &(Source::Net(s)) ).unwrap();
                    tx_ch.send(st).unwrap();
                }
            }
        });
    }
    pub fn distribute(&mut self, op:String) {
        for mut stream in &self.net_txs {
            stream.write(op.as_bytes())
                .expect("transport::distribute - Failed to write to server");
        }
    }
    pub fn connected(&self) -> Vec<std::net::SocketAddr> {
        let mut ip_list = Vec::new();
        for t in &self.net_txs {
            ip_list.push( t.peer_addr().unwrap());
        }
        ip_list
    }
    ...
```

Figure 4.13: The `join, distribute` and `connected` methods of the transport layer

The `distribute` method, shown in figure 4.13 is used to broadcast strings to all peers. And the `connected` is used to get a vector of peer addresses, to be used when the editor needs to show who it is connected to.

Chapter 5

Evaluation and experiment

In this section we look at a collaborative editor which serves as an experiment to demonstrate Ditto's CRDT implementation.

5.1 Editor

We constructed a simple text-only terminal editor to showcase the the Ditto framework and Text CRDT. a full-scale web editor, is a commendable goal, but beyond the scope of this problem.
Though, having implemented the CRDT, and transport layer, the backbone, have been laid out to support an eventual expansion of the editor.

So our text-editor served us well in illustrating the Ditto framework and more specifically Text. We could get a graphical feeling for how Text worked and how we used it to make our text editor.

5.2 CRDT

Ditto's `Text` data type served well as a component in our 3-piece system: Editor, CRDT and transport layer. We've shown that it can works with an external transport layer and can be used to create a collaborative editor.

Other sequence CRDTs could have been used to carry out this experiment of creating a collaborative editor.

A CRDT library (crate) of our own would also have been a possible. Not only combining different components together, but making some design choices for the data type.

5.3 System tests

The system we are to test is the whole editor, that includes the terminal, CRDT and the transport layer. We set up a script that executes the n instances of the editor with a prepared configuration file. The configuration file is set up

with its own listening IP address and optionally an address that it should connect to. The script we have spawns the editors needed and takes over the `stdin` and `stdout` streams to and from the editors. In this way we can feed each instance of the editor a string and get the output string for comparison with the expected. To enable the test some of the race conditions we use the conditional compilation flags to introduce delays in the transport layer to simulate a real-world but consistent test scenario.

```
[features]
customfeature = []
```

Figure 5.1: Cargo.toml: The custom feature conditional compilation flag

And in the transport/lib.rs file with a 1 second delay when the testing flag is set.

```
    #[cfg(feature = "testing")]
    pub fn distribute(&mut self, op:String) {
        thread::sleep(time::Duration::from_millis(1000));
        for mut stream in &self.net_txs {
            stream.write(op.as_bytes())
                .expect("transport::distribute - Failed to write to server");
        }
    }
    #[cfg(not(feature = "testing"))]
    pub fn distribute(&mut self, op:String) {
        #[cfg(feature = "testing")]
        for mut stream in &self.net_txs {
            stream.write(op.as_bytes())
                .expect("transport::distribute - Failed to write to server");
        }
    }
```

Figure 5.2: transport/lib.rs: The custom feature conditional compilation flag for the code needed to be run when testing

5.3.1 Testing the CRDT and `site_id` configuration

To do this two node has to be set up to connect with each other with the –features testing flag. Node "a" writes a string "a" to it state and Node "b" writes the string "b", now both should we expect("ab") because of the CRDT and the site_id the two elements can be ordered.

```rust
fn main() {
    // Spawn a process. Do not wait for it to return.
    // Process should be mutable if we want to signal it later.
    let mut node_a = Command::new("cargo")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .arg("run")
        .arg("-p")
        .arg("--features testing")
        .arg("editor")
        .arg("node_a.toml")
        .spawn().ok()
        .expect("Failed to execute the cargo run -p editor command");

    let mut node_b = Command::new("cargo")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .arg("run")
        .arg("-p")
        .arg("--features testing")
        .arg("editor")
        .arg("node_b.toml")
        .spawn().ok()
        .expect("Failed to execute the cargo run -p editor command");

    let a = "a".to_string();
    let b = "b".to_string();
        write!(node_a.stdin.unwrap(), "{}", a).unwrap();
        write!(node_b.stdin.unwrap(), "{}", b).unwrap();

        let output_a = String::from_utf8_lossy(&(node_a.stdout));
        let output_b = String::from_utf8_lossy(&(node_b.stdout));
    assert_eq!(a, b);
}
```

Figure 5.3: The `main` function of the tests workspace

5.3.2 Testing multiple connections to a network

To test the network ability to connect two simultaneous joins from each side of the network, we need minimum four nodes. Node a and b, that are already connected and node c and d that are joining the network. If we put in a delay of 1 second when the the node joins, it will get an old listener list from the network. Resulting in a partition of the network if not handled correctly.

```rust
    #[cfg(feature = "testing_conn")]
    pub fn join(&mut self, addr:std::net::SocketAddr,
            tx_ch:std::sync::mpsc::Sender< String >,
            listener_package: String) {
        thread::sleep(time::Duration::from_millis(1000));
        let mut stream = TcpStream::connect(addr)
            .expect("Transport::join() - Could not connect to server");
        stream.write(listener_package.as_bytes())
            .expect("transport::distribute - Failed to write to server");
        ...
    #[cfg(not(feature = "testing_conn"))]
    pub fn join(&mut self, addr:std::net::SocketAddr,
            tx_ch:std::sync::mpsc::Sender< String >,
            listener_package: String) {
        let mut stream = TcpStream::connect(addr)
            .expect("Transport::join() - Could not connect to server");
        stream.write(listener_package.as_bytes())
            .expect("transport::distribute - Failed to write to server");
        ...
```

Figure 5.4: transport/lib.rs: The custom feature conditional compilation flag for the code needed to be run when testing

Testing the editor on a system level have shown to be more challenging than expected. It is suspected that the problem arises with the Termion ways of working with the key reading.

```
thread 'main' panicked at 'called Result::unwrap() on an Err value: Os {
    code: 25,
    kind: Other,
    message: "Inappropriate ioctl for device"
}', libcore/result.rs:945:5
```

Figure 5.5: The error message of the tests execution

Chapter 6

Meta/reflection

6.1 software structure

The editor program is centered around the event hooks, that make the software more modular. In the creation of this structure the terminal module was included in the event hook part of the program. This seem to have caused an unstructured form in the software separations. The terminal with the event hooks and the main communication channel, should have been separated from the terminal into separate modules.

6.2 Cursors

The implementation as it is, is only showing the editors local or own cursor, and has the flaw that if an other user adds or removes text before the local cursor, the cursor position stays at that index. What we should do is to stay at the position of that character the cursor is pointing at. Since the module where the cursor should be in (the terminal module), it would not have the information to figure out which index it should move to, because when an update on the CRDT comes from an other peer, the value method of the CRDT only outputs a string. A solution could be that the CRDT contained a marker for each cursor of each peer, that follows the identification of that element. The solution is implemented so that a query of the marker would give the list of indices of all the peers. In that way the terminal and the CRDT could still remain with the principle of separation of concern.

6.3 Node registration list conflicts

The nodes list of connected peers, could be added to a CRDT an ordered set to avoid any conflicts in the setup of the network topology. We could add a circular topologies to transport layer, to accommodate the registration conflict. The existing topology is a fully connected graph, where all the operations of the manipulated text is distributed. This ensures a high responsiveness of containing text in the editor. A key press gets distributed to all nodes directly. The fully connected graph does have a weakness. When two nodes connects to two different nodes at the same time, the network would be split in two sets of nodes with the original set intersecting. And the result would be that the two new nodes would not have each other in their network.

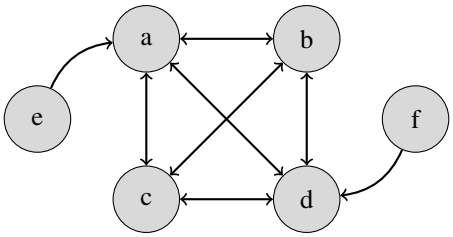


Figure 6.1: 1

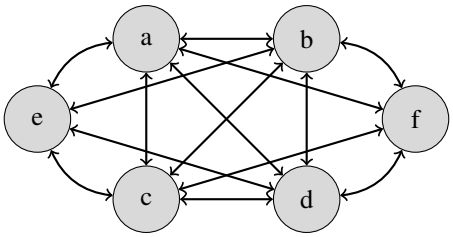


Figure 6.2

A possible solution is to make a ring connected graph, and distribute text state and a state of a set of nodes. In this way the two set earlier mentioned would merge and the text would converge to the same content. The node addresses are comparable so we can actually implement the next trait for the set ditto type. In this way we can send the two states to the next node and ensure we have visited all nodes and created a ring connected network.

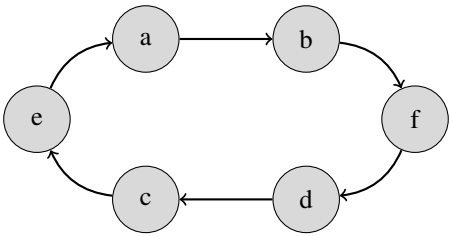


Figure 6.3: A network in a ring topology

6.4 Automatic node identification number assignment

All the nodes in the network has to be assigned a node identification number so that the CRDT can make a total order of the set values. In the current implementation node identification number is given by the configuration file. This identification number could be implemented with a CRDT counter, where new nodes are assigned the counter value plus one. To accommodate the problem with two nodes joining from each side of the network, the node id should be a tuple with the counter + 1 and the assigning node identification number. This would be an other variant of a CRDT where any two values can be compared, and we would have a total order of all the node identification numbers.

Chapter 7

Conclusion

The challenge of putting the three pieces, the terminal, Ditto and the transport layer, has shown to be a good case to integrate in Rust. To choose a channel for the main backbone of the program and that makes a durable internal architecture, because all actions are serialized. The observer pattern serves to separate the different module, although it might have been even better to dedicate it to its own workspace.

The CRDT brings both latency down and data resiliency up, and the Ditto implementation in Rust has shown to be easy to use. The implementation of the editor has shown that the transport layer opens up a whole new world of ways to arrange the topology in cooperation with the CRDT. Where our solution gave a simple fully connected graph, many problems can be solved with other topologies and uses of CRDTs.

Implementing a Text CRDT into an editor has shown a durable and fast way to connect a data type across several nodes. Although Ditto's `Text` CRDT (LSEQ) is highly text optimized the project has show that it is possible to make a an optimistic and data resilient distributed data type. These properties could be beneficial in other application, like peer to peer gaming or shared object in a peer to peer chat application. Although the different parts of the editor can be improved the overall solution is working.

Bibliography

[1] Cap theorem - wikipedia. https://en.wikipedia.org/wiki/CAP_theorem, (Accessed on 06/04/2018) (cited on page 3).

[2] Google Docs. http://docs.google.com (cited on page 2).

[3] Is there an alternative to the tcp/ip model that does not utilize the internet protocol (ipv4/6)? - quora. https://www.quora.com/Is-there-an-alternative-to-the-TCP-IP-model-that-does-not-utilize-the-Internet-Protocol-IPv4-6, (Accessed on 06/02/2018) (cited on page 5).

[4] G. Matt. Simple event hooks in Rust. URL: https://mattgathu.github.io/simple-events-hook-rust/ (visited on 05/21/2018) (cited on page 9).

[5] Nano. https://www.nano-editor.org/ (cited on page 2).

[6] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *13th ACM Symposium on Document Engineering (DocEng)*, pages 37–46, Florence, Italy, Sept. 2013. DOI: 10.1145/2494266.2494278. URL: https://hal.archives-ouvertes.fr/hal-00921633 (visited on 06/04/2018) (cited on page 4).

[7] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In page 259. ACM Press, 2006. ISBN: 978-1-59593-249-5. DOI: 10.1145/1180875.1180916. URL: http://portal.acm.org/citation.cfm?doid=1180875.1180916 (visited on 06/11/2018) (cited on page 4).

[8] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A Commutative Replicated Data Type for Cooperative Editing. In pages 395–403. IEEE, June 2009. DOI: 10.1109/ICDCS.2009.20. URL: http://ieeexplore.ieee.org/document/5158449/ (visited on 06/11/2018) (cited on page 4).

[9] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, Mar. 2011. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2010.12.006. URL: http://www.sciencedirect.com/science/article/pii/S0743731510002716 (visited on 04/19/2018) (cited on page 4).

[10] A. Shapiro. Ditto: CRDTs for common data structures like maps, vecs, sets, text, and JSON, May 10, 2018. URL: https://github.com/alex-shapiro/ditto (visited on 05/10/2018) (cited on page 5).

[11] M. Shapiro. Microsoft research talk: strong Eventual Consistency and Conflict-free Replicated Data Types. URL: https://www.youtube.com/watch?v=oyUHd894w18&feature=youtu.be (visited on 03/07/2018) (cited on page 3).

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg. Springer-Verlag, 2011. ISBN: 978-3-642-24549-7. URL: http://dl.acm.org/citation.cfm?id=2050613.2050642 (visited on 02/22/2018) (cited on page 4).

[13] Share LaTeX. http://sharelatex.com (cited on page 2).

[14] S. Weiss, P. Urso, and P. Molli. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In pages 404–412. IEEE, June 2009. DOI: 10.1109/ICDCS.2009.75. URL: http://ieeexplore.ieee.org/document/5158450/ (visited on 06/11/2018) (cited on page 4).

[15] Wikipedia contributors. Osi model — Wikipedia, the free encyclopedia, 2018. URL: https://en.wikipedia.org/w/index.php?title=OSI_model&oldid=843868486. [Online; accessed 2-June-2018] (cited on page 5).