1 Intro

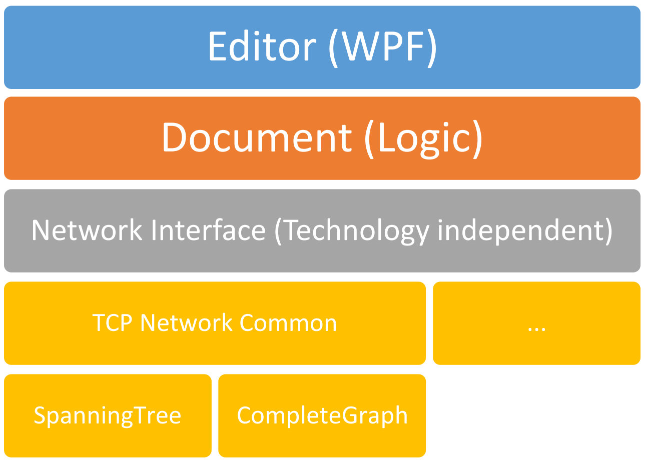
First, we decided that our application should be scalable upon a big number of clients. Therefore we first thought a lot about possible network topologies. A fully-connected graph would be easy to implement, but does not scale as well as a topology where each participant only knows a small subset of the infrastructure. We also decided against a pure UDP-broadcast solution because of the possible package loss, although UDP broadcasts and multicasts are predestinated for reaching a broad audience. Our decision fell on a spanning tree topology where packages are routed and flooded efficiently. This topology can be built simply, but is harder to maintain than fully connected graphs or connectionless topologies. Especially handling dead peers (disconnect, network failure) is tedious in spanning trees. Therefore, a fully connected infrastructure has also been implemented for comparison. The used network implementation is defined by the settings of the dependency injection framework in the class SyncEdModule in SyncEd.Editor.

Our second implementation decision regarded network technology and was based on the uncertainty whether an existing communication framework could realize our desired network topology or not. We considered using well-established technologies such as WCF, Remoting, App Spaces as presented in the lecture as well as the frameworks listed on the exercise sheet. However, we felt that all of these require some kind of distinguished super/server node. We therefore decided to not use any existing framework and rely on basic sockets for the time being. Although this decision might seem odd, as handling raw sockets is typically more fragile and harder than higher level communication frameworks, it turned out that the main difficulty was maintaining the network topology which is independent of network technology.

Concerning the data structure of the document, we decided not to partition the document into smaller pieces (e.g. per line or sentence) or represent it by a higher level structure. Therefore our text document is a simple string (-builder). Text changes, which are sent in the network, only contain the position in the text and the change itself. A change can include a string which will be inserted on a specified position, or a number of characters which should be deleted at a specified position. Furthermore, profiling sessions showed that operations on the document text data structure (string builder) are insignificant as the vast majority of runtime is consumed by the custom renderers of the UI as well as packet serialization inside the network stack.

2 Architecture overview

Since we were not sure which network-framework would fit our need best, we designed our architecture with a flexible network layer. Therefore we specified a network layer which defined only interfaces and common network logic. On top of the network logic the document-layer defines the business logic for managing the document, the curser positions of the clients, etc. Finally, the editor-layer, which uses the WPF, displays the document text, and handles user input.



The following sections contains a more detailed description of each layer.

2.1 Editor-Layer

The Editor only contains a single window (MainWindowView) and its corresponding ViewModel. Within the ViewModel everything we display in the GUI is stored. Changes made by the GUI are reflected to the view model, which are then forwarded to the document-layer. Some logic and some states of the ViewModel are introduced because not all changes are forwarded directly into the document. For instance, changes from the network which triggers updates in the ViewModel also causes the Gui to send change events. Events that are trigger by processing changes from the network are not forwarded into the network again to spare the network.

For preventing typing within the range of a “foreign cursor” we introduced a ranges where editing is not allowed. This ranges are displayed with different colors based on the clients IP and port. When the own cursor is moved into a forbidden region, the textbox is locked and editing is only possible when manually position the cursor into an allowed region. Additionally, editing on the first and after the last index of text is always possible.

The HighlightTextBox, derives from a normal WPF-Text box and overrides the Rendering procedure to visualize the color ranges. Two new Dependency Properties, the HighlightRanges and the CaretIndex, are added which are bound to the MainWindowViewModel.

3 Document-Layer

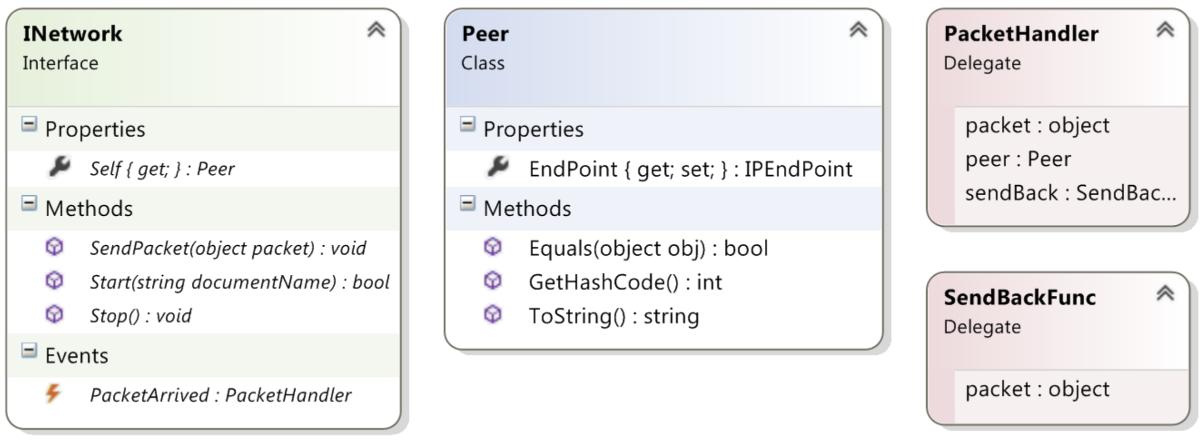
The document layer specifies a document-interface for connecting to a document, submit text changes, and events for text- and cursor changes. Changes to the document caused by the UI (Editor layer) are converted into network packets and given to the network layer. Incoming packets from the network layer are integrated into the document’s state and usually trigger events which update the UI. The document layer therefore serves the purpose of a business logic and the data model by interpreting incoming packets and UI events as well as perform operations on the document data itself. This layer is also further responsible for the initial setup and retrieval of existing information on the network such as the current number of editing peers and the document text.

The document layer is also responsible for “merging” concurrent changes into the document. We started with an implementation of a simple merging strategy which merges the changes based on their arrival sequence. Tests with this merging strategy showed that, even when hammering onto the keyboards with two clients on two machines, there were no merging conflicts. Within the same LAN the latency is small enough, that no complex merging strategy was required. Still, we thought about implementing a vector clock for detecting possible merging conflicts.

4 Network-Layer(s)

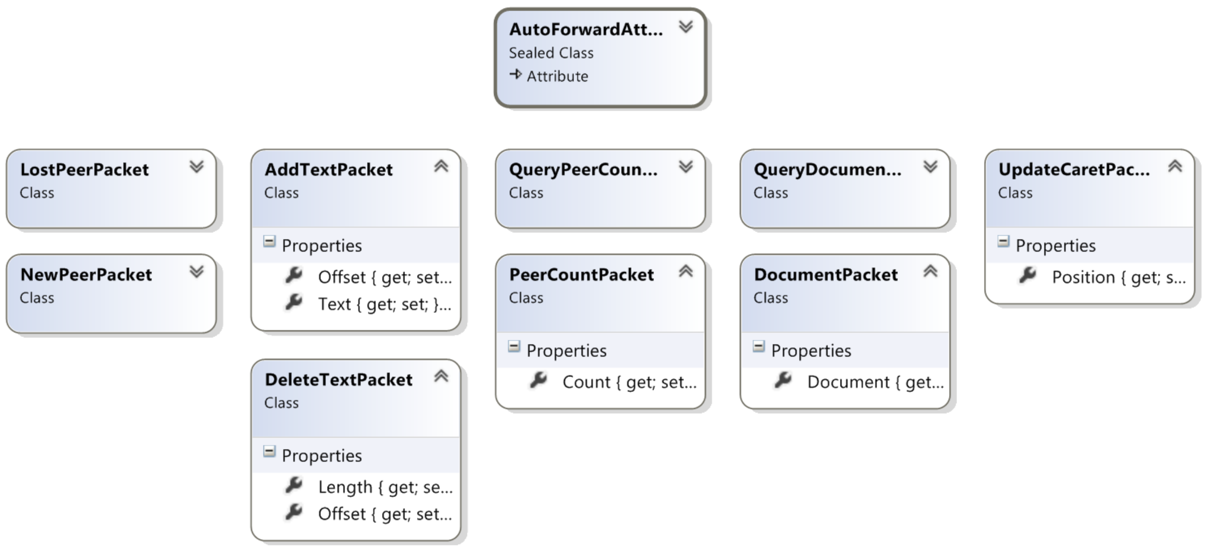
4.1 Network interface

The stack of network layers is responsible for managing connections between participants (called peers) of the same document. The network layer is itself structured into several further layers to achieve a clean software architecture. The very top layer contains the technology independent C# interface INetwork which does not specify the underlying connection technology (e.g. TCP, UDP, WCF, etc.). This interface which is injected into the document layer and offers the following four interactions:



- Start(): Initializes the network layer and triggers construction of the network topology. This method requires the document name which is used as an identifier on the network to find peers of the same document. The return value of Start() indicates whether a peer could be found (true) or the current application instance is the first peer for the given document name.
- Stop(): Shuts the network down. Disconnects from all peers of the network topology. After a call to this method no further packages can be sent or are received.
- SendPacket(): Sends a packet into the network. Although packets are themselves classes and all packet types used by the document layer are defined, this method accepts any object. The object passed to SendPacket() is serialized into a binary representation which is then transmitted to all (or one) peers of the network. Generally (apart from forwarding and custom packets) the network layer does not interpret the objects passed to this interface.
- PacketArrived: Event which is raised whenever a packet arrives from the network. This packet can again be any object. In addition to the packet data (object) an instance of Peer is delivered, which serves as a unique descriptor of a peer on the network. The provided sendBack delegate enables higher layers to directly respond to an individual peer as opposed to the SendPacket() method which does not allow the specification of a target.
- Self: Retrievable property holding an instance of Peer which describes the own node/peer on the network. This property is mostly used by the network layer itself to identify the current instance to other peers on the network.

On the highest network layer several kinds of (business logic) packets are available to describe changes to the document and the environment:



- NewPeerPacket/LostPeerPacket to inform other peers on the network that a peer has been added to the network or connection has been lost,
- The AddTextPacket/DeleteTextPacket for document text changes,
- *QueryPeerCount*/*PeerCountPacket* for querying the current number of participants on the network for the set document name,
- *QueryDocumentPacket*/*DocumentPacket* for querying and receiving the whole document text and
- UpdateCaretPacket for updating the caret position.

The optional attribute AutoForward can be declared on any packet and specifies that the packet should be forwarded to all peers of the network or only to a certain destination. Packets names in italic do not have the AutoForward attribute and are typically used to query information from a single peer.

4.2 TCP network commons

Our implementation provides two network layer implementations which are both built on top of TCP sockets with UDP broadcasts to discover peers. It is therefore reasonable to collect common classes and algorithms in a common layer.

4.2.1 UdpBroadcastNetwork

Encapsulates all required UDP mechanics. These include sending and receiving UDP broadcasts. The UDP listening port is constant as multiple UDP broadcast sockets can be bound to the same (protocol, address, port) tuple. Setting the ReuseAddress option on Windows is required though. To send and receive broadcasts, the Broadcast option has to bet set.

Sending is done by serializing the object to send and writing the data to the UDP socket. Receiving is done by a separate thread which waits in a receive call until an incoming broadcast is detected. The received data is deserialized into an object and passed to a registered callback function.

4.2.2 TcpBroadcastNetwork and TcpLink

Conceptually serves the same purpose as its UDP counterpart. The TcpBroadcastNetwork keeps a collection of established links (instances of TcpLink). Packets can be sent to all links of the TCP network (broadcast) or only sent to a subset of them (multicast/unicast). Sending is implemented as serializing the object to send into a byte array and then writing that data to all TcpLinks (containing TCP sockets). Receiving is implemented in each TcpLink itself, again by spawning a new thread per

TCP connection which blockingly waits until data is received on the socket. Incoming data is deserialized into an object and passed to a corresponding callback.

Apart from data transfer, the more interesting and complex procedure are how connections are established. Initially, a TCP listener has to be created (equivalent to a server socket) which waits for incoming connections. This listener has to be created and started when the network is booted up to be assigned a port from the operating system. The port and address of this listener uniquely define this peer on the network. On Windows, the ExclusiveAddressUse option has to be set on the socket as windows allows multiple TCP listener to be bound to the same (protocol, address, port) tuple. In case of an incoming connection the listener which will receive the connection is determined by how specific the listener specifies the address and port it runs on (e.g. listening on all NICs vs. listening on a specific NIC). Furthermore, as our application cannot determine on which network it will find peers, the IP address of the listener will be set to 0.0.0.0 and is determined by the first received UDP broadcast. Obtaining a valid IP on the right is essential to identify the peer on the network, as this information is also used to connect to peers. An additional limitation of our implementation is thus also the restriction to a single network. Although a peer on the crossing of two networks might be reachable via UDP from both networks, identifiers (instances of Peer with IP and port) would be valid in only one network (a peer would have a different identity in each network). Our implementation relies on this information to be unique and therefore does not support running across multiple networks.
A connection can be established between two peers when one of the peers actively waits for an incoming connection and the other peer connects to the waiting one. Active waiting was designed on purpose as there are cases where a peer would like to refuse a connection. However, as the TCP listeners have to stay opened throughout the application's lifetime to reserve their ports, each connect to a peer succeeds (even if accept is not called on the socket, as the operating system accepts the connection and enqueues the new peer for retrieval by the application). To allow refusing a client a separate handshake is required which is implemented as exchange of TCP listener ports between both peers. This information is at least required by the listening peer as the connection is usually initiated from a dynamic, operating-system-assigned peer and not from the listening socket. This handshake can be refused by not sending a port number at all which causes the connecting party's receive to timeout (timeout is set to several hundred milliseconds). If a connection is successfully established a new TcpLink is created, responsible for sending to and receiving from this link.

4.2.3 TcpObject and UdpObject

Although the network layer only operates on (mostly) not interpreted objects given by and given to the upper layers, all those objects are wrapped into a TcpObject or UdpObject, depending on the network subsystem used to transmit them. These wrappers contain additional data required on each packet. A TcpObject contains an instance of Peer identifying the original sender of the packet. This becomes important for network topologies where packets are routed over multiple hops. A UdpObject contains the document name specified when the network was started. As UDP broadcasts reach all application instances independent of their document name this information is vital to filter incoming broadcasts of different documents.

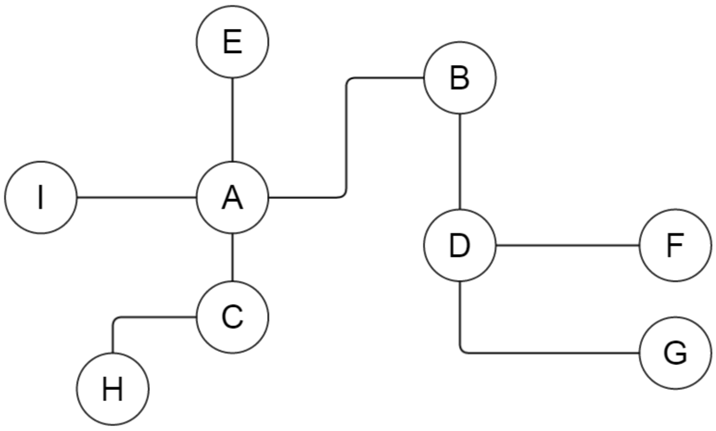
4.2.4 FindPacket

The TCP network layers than define a further packet used for UDP to discover new peers. This packet contains the listening port of the broadcasting peer.

4.2.5 BasicNetwork

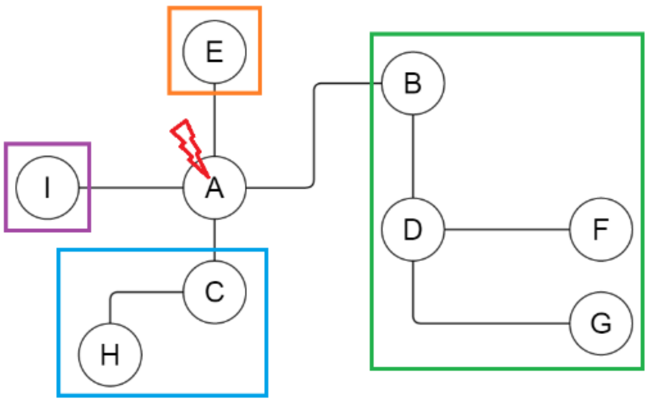
The base class for all TCP network implementations, implements INetwork. On network start, boots up an instance of UdpBroadcastNetwork and TcpBroadcastNetwork and tries to find a peer by UDP broadcasting a FindPacket and waiting for an incoming TCP connection. This procedure establishes the first link to a peer with an existing topology. Some basic functionality is also implemented here such has firing an event to the upper layers when TCP packets arrive, processing UDP FindPackets, removing dead TCP links (disconnect, network failure, timeout etc.) and detecting the own IP address.

4.3 Spanning tree layer



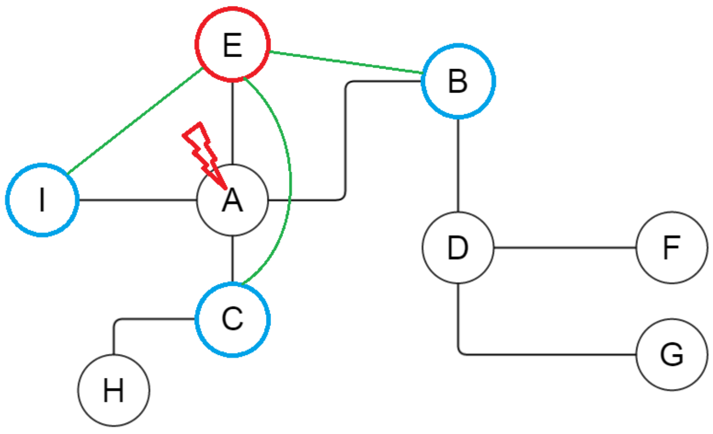
The first and initially developed network topology is a spanning tree. Its main benefit is that it requires far less connections per peer and will probably scale better if the logical topology closely resembles the physical interconnect (cf. sensor networks). Furthermore, a peer only has to know its immediate neighborhood and not the complete network (memory footprint if those networks would become huge). This design principle can also found in several network technologies used in sensor networks or the internet itself.

The topology is very easy to establish as it is completely sufficient for a peer to connect to only one other peer using the functionality provided by BasicNetwork. Incoming TCP packets are deserialized and the object's types are checked for the AutoForward attribute. This is the only time this network layer ever draws information from the packet data. If the attribute is present, the packet is forwarded (resent) to all TcpLinks excluding the link where it was received. No further logic is required to run this topology.



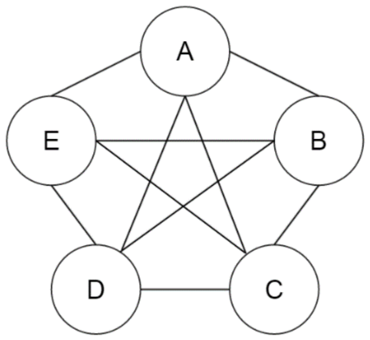
However, link failures (disconnects etc.) are where it becomes ugly. When a peer is lost, the network topology will be split into N parts where N is the number of connections the dead peer had. These network parts have to be joined together again, otherwise N independent documents with the same name will emerge. Trying to reuse the initial peer finding mechanic fails, as any node might answer a FindPacket which could lead to cycles in the topology (could be prevented by introducing a time to live field per packet). Furthermore, a peer does not know how many neighbors the dead peer had and how to find peers of the disjoint topology regions.

Peer loss is detected by the application (and generally by the OS) on the first send to a TcpLink after the failure with no answer (ACK) in a specified time. An exception is raised inside the TcpLink receiver thread which causes the link to be removed from all open links. The spanning tree network will now start a repair procedure (aka panic). Repair starts by broadcasting a PeerDiedPacket (spanning tree topology specific) with the unique identifier for the dead peer (instance of Peer) and the peer which found the dead peer (referred to as repair master later). Each receiving peer (might not be all but must be at least one) check their active TCP links if they contain a link to the dead peer. If such a link is found, it is closed (which again causes a further PeerDiedPacket to be broadcasted) and the peer also starts the repair mode. Therefore, all peers which where neighbors to the lost peer (addressed as panic set) have started the repair mode.

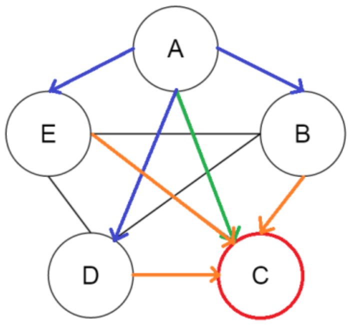


If a peer is in repair mode, all incoming TCP packets are processed but outgoing TCP packets are queued until the repair is completed. Furthermore, the packet which caused the initial repair is also queued. The queue will be emptied after the repair mode has rebuilt the topology to avoid package loss (the peer's documents would get out of sync). The peer which first noticed a dead peer (got an exception when trying to send) will become the master node of the repair operation (red in the picture above). Peers receiving a PeerDiedPacket will be part of the repair but no masters (blue in the picture above). As multiple peers can detect a dead pear at the same time, there may be multiple master nodes. To handle this case, each peer maintains a list of all master nodes it has received during the current repair session. After repair mode has been entered on all peers which have lost a connection a task is queued which will wait a small time interval before starting the actual repair. This time is required for all potential master nodes to send out and receive their PeerDiedPackets. After the time interval has passed, one of the master nodes is picked (by defining an order on instances of Peer using IP and port the picked peer will be the same on all peers). The master node then begins to accept as many incoming connections as there are for another period of time (the master does not know how many other peers where affected by the dead peer). All other peers connect to the master node. After this time window has passed, the network should have been rebuilt. All queued packets are flushed, a LostPeerPacket is created and broadcasted by the master (responsible for decrementing the editor counter on all peers) and the repair mode is exited.

4.4 Complete graph layer



The difficult repair strategy of the spanning tree topology (we had nightmares) ultimately lead to the consideration of replacing it by another topology which does not suffer from expensive repairs. Although the fully connected graph topology does not shine as bright as the spanning tree concerning lightness and number of connections, it has a clear advantage: Lost peers can simple be dropped and nothing else is required.



First things first, to reuse most of the existing infrastructure, connections are equally established as in the spanning tree topology. Let's take for example the peer C in the picture above which wants to connect to the existing infrastructure of A, B, D and E. C issues an UDP broadcast and waits for a TCP connect. Any (one or more) peers answers to C (A and green edge in the picture). A then TCP broadcasts to all nodes except a ConnectToPeerPacket with C as payload, furthermore A sends C a PeerCountPacket with the number of peers currently on the network. The peers B, E and D will try to connect to C when they receive the ConnectToPeerPacket. C will wait for 3 incoming connections. Afterwards, C should have a connection to each peer on the existing network.

But of course, it is not that easy. A might not be the only peer answering to C's FindPeer UDP broadcast. If for example B and E would have also answered (and in networks with almost no loss usually all peers try to connect) C would still have 2 queued connections. When C then receives the PeerCountPacket it will except 3 connections, but will receive 2 + 3 (2 queued and 3 new). These 2 + 3 connections of course contain duplicates (connections originating from the same peer). These could be filtered but lead to further strange effects in our tests. An easy solution would be to turn off the listener socket when it is not needed, but this is not possible as the operating system might than give this port to a new instance of our application running on the same machine. The best solution we have found was to wait a bit after the initial connect (A to C) and then accept and immediately close all pending connections on the TCP listener. This ensures that when the PeerCountPacket arrives, the receiver will truly get the number of incoming connection requests as stated in the packet. Furthermore, as A immediately requests all other peers (B, D, E) to connect to C by sending a ConnectToPeerPacket, C might accidentially discard a rightful connection. To solve this problem, A waits for C to finish discarding connections and sending a RequestInvitePacket to A, before A starts notifying its peers.

5 Concurrency mechanisms

The network and logic are not actively using any concurrency mechanisms, as we never experienced concurrency issues. This might come from the fact that most actions are forwarded and reflected to the view model in the GUI and the WPF-bindings schedules changes and events correctly into the GUI-thread. However, as each network link has its own receiver thread, multiple threads may run through the network layer as well as network events can be raised concurrently from different threads. As a consequence a few locks are required (e.g. when accessing the TCP listener, the collection of network links or the document string).

6 Difficulties and Known Bugs

The management of connections, especially the repair mechanism turned out to be very difficult to implement.

We started developing the client as a pure network-application and therefore never expected to run multiple client on the same machine. Because we initially used the IP-address and a well-known port for our application, the changeover to a different identifier and a flexible port was rather troublesome. The current implementation supports multiple peers on the same machine but on the cost of implications on the network layer. For example the TCP listener's port is found by try and error and also has to be communicated to other peers. Furthermore, an opened TCP listener has to be kept open for the application's lifetime. Otherwise the same port could be chosen by another instance running on the same machine and peers would no longer be uniquely identified by the IP and port of their TCP listener.

7 Performance and scalability

Our main performance issue emerges not from network or synchronization overhead. Instead it turned out that our custom Textbox, which highlights the locked regions near other editors' cursors, is very inefficient. The rendering procedure takes quite long and we weren't able to fix it. During typing at higher speeds, a CPU core is fully utilized by the UI, thus limiting the number of clients on a single machine to probably 1 or 2 per CPU core. On a distributed network, the number of peers is only limited by range of the broadcast, the port range of (1337 – 65535) and the network bandwidth. If the network becomes overstressed packets might be delayed, leading to inconsistency between the document's texts.

Network traffic and overhead can also not be neglected. Although packets contain only small changes of the document, a lot of small packages are sent into the network (more smaller were favored instead of less larger ones). Collecting changes over some time and sending one big package with all gathered information could reduce overhead.

8 Time and effort

First of all, thank you for a very interesting exercise. However, the nature of this application is probably far away from what was intended. With an effort of over 80 hours and more than a 100 commits, around 15% of the time where spent on the UI, 5% on writing the NetworkDocument (which I believe would be the intended subject of interest) with probably half an hour thinking about a suitable data structure before deciding that StringBuilder with some locks will do, **65% of the time to write spanning tree topology** and network infrastructure, 10% for creating the complete graph topology and finally 5% for documentation. This exercise would have been wonderful as a half-semester project for our network course, but clearly fails the target to teach concurrent/parallel programming.