Shared Text Editor

Matthias Leitner s1310454019

Robert Stoll s1310454032

Description

The Shared Text Editor provides a simple collaborative text-editing tool, which allows multiple users to edit the same piece of text simultaneously. Changes to the edited document are distributed to all associated editors instantly.

The editor is implemented using C# .Net and the Windows Communication Foundation (WCF) for Peer-to-Peer as well as Client/Server communication.

Architecture

Communication

As previously mentioned the communication of the editor is implemented using WCF technologies. In order to fulfill the requirement for automatic document discovery between multiple editing clients we use NetPeerTcpBinding, which has been supported since .NET Framework 3.0. It provides everything we need in order to discover clients within the same LAN and broadcast requests for document discovery to all possible hosts. Once a client has started editing a document it communicates with the owner of the document via HTTP using WCF BasicHttpBinding. Clients are sending their patches to the owner and the owner in turn sends the applied patches via multicast to the other known editors as well as an acknowledgement to the client who send the corresponding update. Channeling the document update through the owner should help to avoid patching conflicts and avoid unnecessary broadcast messages if possible. When patch is received via the UpdateRequest contract the sender will receive feedback via the AckRequest contract, letting the client know if the patch was applied successfully.

WCF Contracts

P2P

For discovering documents and clients the following contract is used:

```
        [OperationContract(IsOneWay = true)]
        void InitializeMesh();

        // Look for the given documentId within the p2p mesh
        [OperationContract(IsOneWay = true)]
        void FindDocument(string host, string documentId, string memberName);
```

Client/Server

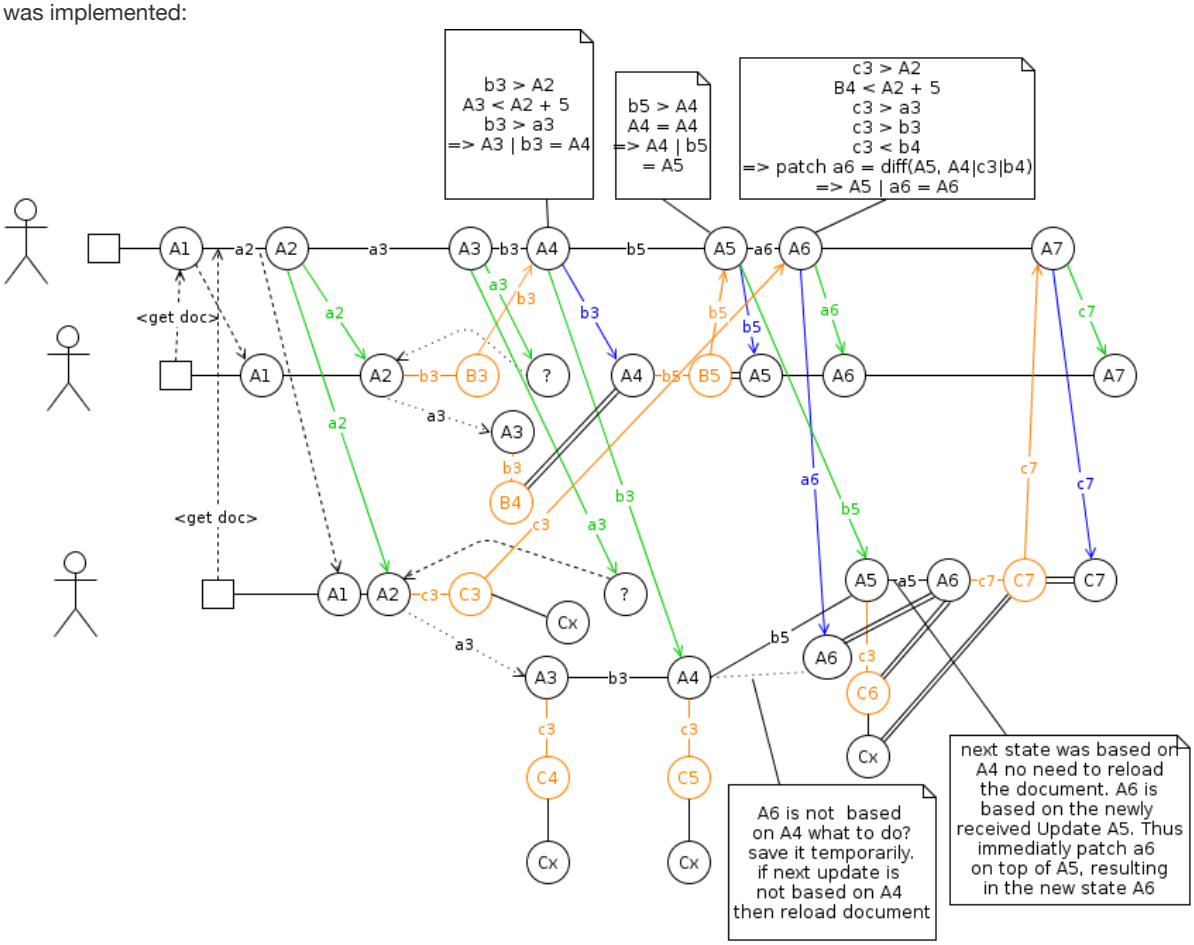
For the one-to-one communication between an editor and the owner of a document the following WCF contract is used:

```
    [ServiceContract(SessionMode = SessionMode.Allowed)]
    public interface ISharedTextEditorC2S
    {
        // Transmits a patch to other clients
        [OperationContract(IsOneWay = true)]
        void UpdateRequest(UpdateDto dto);

        // The callback to the UpdateRequest contract
        [OperationContract(IsOneWay = true)]
        void AckRequest(AcknowledgeDto dto);

        // The response/callback to the FindDocument contract, sending the requested document back to the client
        [OperationContract(IsOneWay = true)]
        void OpenDocument(DocumentDto dto);
    }
```

Update/Sync Logic

The editors use the open source Diff, Match and patch libraries from Google Inc. which provide robust algorithms to perform the operations required for synchronizing plain text. These algorithmns implement the principles of Operational Transformation which also represents a core concept behind colloparative software by the company such as Google Docs and Google Wave. Creating patches reduces the amount of data transferred and allows the changes to be applied without relying on static indexes within the text.

The following figure depicts a few use case scenarios and is used as basis to explain how operational transformation was implemented:



User A is the owner of the document in the above scenario. That means he has created it using the "Create" Button. User B and C both open the document (with the "Open" Button) - depicted by the in the above figure.

User A modifies the document and patch a2 is generated. The document has now revision A2 and the owner sends the patch a2 to the remaining editors (denoted by the two green lines going out of A2).

As a next step, User A, B and C simultaneously modify the document. User A creates the patch a3 and sends it to the editors. That is straight forward, an owner has always precedence over other simultaneous modifications.

Let us see what happens to the modification of user B. User B creates the patch b3, applies on top of A2, which results in B3, which in turn is saved as pending update (depicted by an orange circle). User B sends an UpdateRequest (denoted by an orange line) with the patch b3 to the owner (User A). Now, the owner checks whether the given patch is based on the current state or not. This is depicted by b3 > A2 (which shall be read as b3 is based on A2). A2 is not the current state and thus the owner checks whether it is not further behind than 5 commits of the current state (this number is configurable). This is depicted by A3 < A2 + 5 (which shall be read as is A3 not newer than A2 + 5 commits). In this case A2 is not older than 5 commits. As next step, the owner verifies if the patch put on A2 has precedence over the given patch b3. The patch put on A2 was a3 and a3 has precedence over b3 (depicted by b3 > a3 which shall be read as b3 has to follow after a3) - as a short parenthesis, precendence is determined by the memberId of a user. The user whith the lower memberId has precedence where the owner makes an exception, he/she has always precedence - In this case, the server can put the patch b3 on top of A3 which creates the new state A4. An update with the patch b3 is send to user C and an acknowledge (denoted by a blue line) is sent to user B.

As one surely noticed, there is a slight problem in the whole story. User B received the patch a3 before the acknowledge for his UpdateRequest b3. In this case the user proceeds as follows. First it checks whether the given update is based on the last state. In this case this is true, thus user B checks whether the given patch a3 has precedence over the pending update request b3. This is true as well and thus user B applies the patch a3 on top of A2 and enter the state A3 as well. The patch b3 is recalculated and is now based on A3 - the pending update B3 is updated accordingly which results in the new pending update B4. Once user B receives the acknowledge for b3 it checks whether the given state (based on NewHash in the given AcknowledgeDto) corresponds to the pending update (is B4 = A4). In this case this is true and thus the pending update is deleted and user B could send a new UpdateRequest to the owner, which is happening with the patch b5. The acknowledge for b5 is straight forward since no new update was made in the meantime.

Let us go back to state A2 and see what happened with the update request of user C. For what ever reason, the UpdateRequest c3 was transmitted very slowly and lot of interaction happened before the owner (user A) gets it. Once user A gets the UpdateRequest it proceeds as described above (UpdateRequest b3) which can be summarised as:

1. check if the UpdateRequest is based on the current state or not older than 5 commits: c3 > A2 (which reads as c3 is based on A2) - current state is B4 - is B4 not newer than A2 + 5 commits => not its not, ok
2. check if the applied patch on A2 has precedence over the given path c3: c3 > a3 => yes a3 has precedence
3. move to next state (A3) and check if applied patch has precedence: c3 > b3 => yes b3 has precedence
4. move to next state (A4) and check if applied patch has precedence: c3 < b4 => no, the applied patch should have been applied later on
5. calculate new patch a6 based on diff(A5, A4|c3|b4), which corrects the order
6. put patch a6 on top of A5, resulting in the new state A6.
7. Send an update to editors excluding the one which has initiated the update (user C) => send update to user B (blue line)
8. Send acknowledge to initiator (user C - green line)

There are two use cases left in conjunction with user C. Let us see what user C did while waiting for her acknowledgement. User C manipulated the document straight after the UpdateRequest c3 was send to the owner. Only one update at a time can be send to the owner and thus user C has created a temporary state Cx which represents the current text in her editor. In a next step, user C gets the update a3 from the server. User C takes the same procedure as user B when user B received the update a3. The procedure is very similar to the one taken as an owner and is outlined here again as a step by step description:

1. check if the UpdateRequest is based on the current state or not older than 5 commits: a3 is based on the current state A2 (notice, C3 is the pending update and not the current state).
2. check if the update has precedence over the pending update: c3 > a3 => yes a3 has precedence
3. apply the update on the current state (A2 | a3 = A3) which results in the new state A3
4. recalculate the pending update in order that it is based on the new state A3 and apply the pending update on A3 (A3 | c3 = C4) which results in a new pending update C4.
5. recaluclate the temporary state Cx in order that it is based on the new pending update C4 (notice, User C might have modified the document further in the meantime, Cx here does not need to correspond to the previous Cx. Cx merely stands for the text in the editor).

The same prodecure applies when user C receives the update b3. The last use case happens next. User C receives the acknowledgement a6 before the update a5 (for whatever reason). In this case, the pending update would not correspond to the given state (C5 != A6). Since A6 is based on a state which is not known so far, user C assumes that she missed an update. A6 will be saved temporarily in the hope that the next update is the one we missed, is based on the current state respectively. Luckily the next update (b5) is based on the current state A4. We then check if A6 is based on the new state A5. In this case it is true and we can apply a6 on top of A5 which results in A6. In the same time we apply our pending update on top of A5 (resulting in C6) and verify if C6 equals A6. In this case this is true and hence there is not need to reload the document. Since the pending update could have been acknowledged now, we can send the next update to the server which means we can now turn the temporary state Cx into an pending update (here C7). As a side notice, if A6 were not based on A5 then we would have waited for yet another update to see whether it is based on A5 or not and whether A6 would have been based on it or not. As long as we only get missed updates we do not reload the document. Otherwise we need to reload the document (we have effectively missed an update).

User Interface

The user interface is realized using the Windows Forms APIs. All the logic is decoupled from the UI and should therefore never block the user interaction. WindowsForms TabControl is used to allow opening and editing multiple documents at

the same time.

Dependencies

google-diff-match-patch

code.google.com/p/google-diff-match-patch

Known issues

- Open a document if there are multiple owners with the same document name. The owner responds to the document discovery response first is chosen.
- some error handling is missing. For instance, if a user sends an erroneous patch to the server (e.g. delete everyhting on line 100 but there are only 10 lines), then the server just swallows the patch without informing the client that it was not applied. The client would then wait forever for an acknowledge
- No multi caret support. The position where a user is currently editing is not visualized within the editor using multiple colored carets as in Google Docs.
- In order to support multiple clients on the same machine the port for the client to server communication is randomly choosen between 9000 and 9010. If the port is already in use the client will retry 10 times to find another free port. In order to allow access to ports for the BasicHTTPBinding you need to run the following command for range of ports you want to allow:

```
@for /L %%i in (9000,1,9010) do netsh http add urlacl url=http://+:%%i/ sddl="G:S-1-5-32-545"
```

Time spent

Planing/Research: 8h

Implementation: 38h

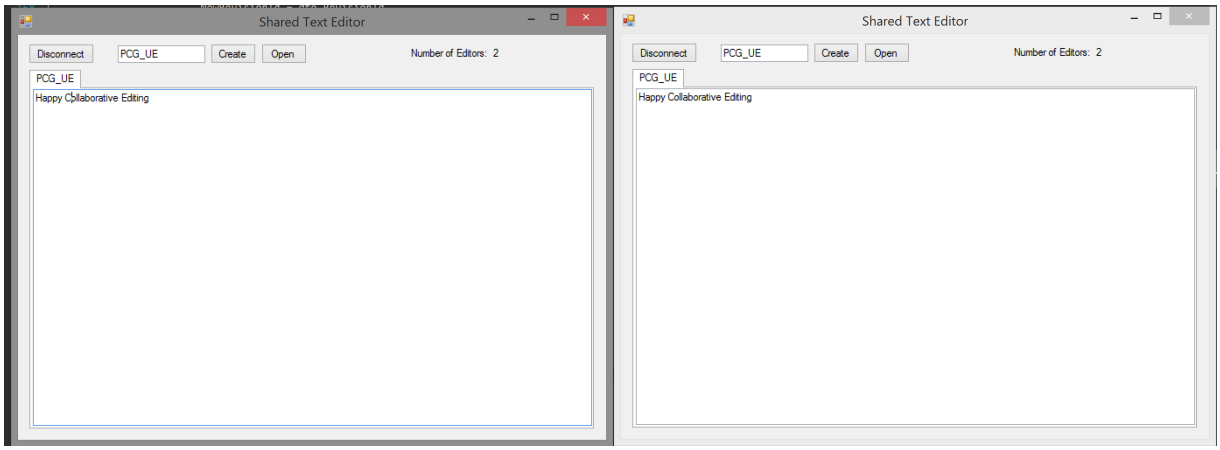
Testing/Bugfixes: 12h

Documentation: 5h

User Guide

- Start the client
- Click 'Connect' in order to join the P2P Mesh
- Wait until the connection to the mesh is established
- Enter the name of the document you want to edit
- If you want to join an existing document with the given name click 'Open'. If you want to create a new document click 'Create'
- Start editing the document in the text arear
- To close an open tab, type "CTRL+W"

Screenshot


Shared Text Editor
Disconnect
PCG_UE
Create
Open
Number of Editors: 2
PCG_UE
Happy Collaborative Editing
Shared Text Editor
Disconnect
PCG_UE
Create
Open
Number of Editors: 2
PCG_UE
Happy Collaborative Editing