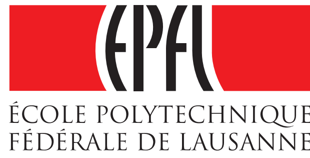

Implementation of an Algorithm for Peer-to-Peer Collaborative Editing

Damien Aymon

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Bachelor Semester Project

June 2017

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Kirill Nikitin
EPFL / DEDIS

List of Figures

1 Execution of an Operation on a String. Own Illustration . . . 3
2 Generation of two Concurrent Operations. Own Illustration. . 3
3 Effects Relation. Own Illustration. 5
4 General Organization of the Software. Own Illustration. . . . 8
5 ABTUInstance Struct. Own Code. 9
6 Receiving Buffer Manager Struct. Own Code. 10
7 Receiving Buffer Manager Control Structure. Own Code. . . 11
8 Frontend Controller. Own Code. 13
9 ABTU Controller. Own Code. 14
10 Remote Thread. Own Code. 15
11 The Big Picture. Own Illustration 16
12 Communication. Own Code. 17
13 Instantiation of an ABTU Instance. Own Code. 20
14 Instantiation of a Communication Service. Own Code. 21
15 Local Thread. [SLG10]. 21
16 IntegrateL. [SLG10]. 22
17 Remote Thread. [SLG10]. 22
18 IntegrateR. [SLG10]. 23

Contents

List of Figures 2

1 Introduction 1

2 Goals 2

3 Corresponding Background 2

3.1 Operational Transformation 2

3.2 ABTU Algorithm 4

3.2.1 Notions and Notations 4

3.2.2 Undo of Operations 5

3.2.3 Description of the Algorithm 6

4 Design and Implementation 8

4.1 ABTU Instance 8

4.2 Receiving Buffer Manager 9

4.3 Communication with Frontend 10

4.4 ABTU Control Structure 12

4.5 ABTU Local Thread 12

4.6 ABTU Remote Thread 13

4.7 The Big Picture 14

4.8 Communication with Peers 14

5 Results 16

6 Limitations of the Project and Future Work 18

7 Conclusion 18

References 19

A Installation 20

B Psoeudocode snaphots for local and remote thread 20

1 Introduction

Nowadays, real-time collaborative editing tools are well-known and almost daily used amongst students and employees who need an easy and ready-to-use solution to work on a common project. In contrast to version control systems like Git, users must be able to edit the same document simultaneously and at any time. The changes inferred by one user need to be integrated at every other site and the algorithm must converge towards a unique definitive document version. The most popular and used amongst current implementations are Etherpad and Google Docs, the latter having been launched in 2007.

Despite their accessibility and convenience of use, those existing systems rely on a central organ to process and redistribute operations generated at each site. This involves the loss of control over the data by users. In other words, current solutions infer a trust relation between users and the service provider. Possible system failure or data leakage towards other organizations might break this trust relation.

This problem can be addressed by using a distributed and independent algorithm, which communicates by secured peer-to-peer communication between sites. Collaborative groups can now have ultimate control over their data, communication and how different sites converge to a definitive version of their document.

The task is to implement an admissibility based operational transformation algorithm for peer-to-peer collaborative editing (ABTU) and evaluate its performance. The algorithm will be based on [LL10] and [SLG10]. This algorithm is the only one that has been proved to enable parties to converge their state in a peer-to-peer configuration. The possibility to undo any changes is also a key feature of such an algorithm and must therefore also be implemented. This part will be based on [reference.] The programming language of choice for the backend in this project will be Go!.

In order to complete this task, one will first have to study the ABTU algorithm. In parallel, the implementation of peer-to-peer communication will be done to learn the programming language. After having finished the implementation of the algorithm, testing will take place. The last part will be to define limitations and further improvement to be done.

Firstly, goals for the project will be further defined. The paper will then go through an introduction on operational transformation and the algorithm which will be used. The design and implementation will be precisely explained. Results and limitations of the project will be presented and lead to a discussion on the improvements and future work to be done. Details about the installation will be provided in the appendix.

2 Goals

In this section, we will further define the goals of this project.

The algorithm which will be implemented is part of a software which is organized as follows. The graphical interface and database management are implemented in JavaScript, this represents the frontend. The ABTU algorithm is implemented in Go. The two parts are linked by a management part, also implemented in Go, which handles different instances of the ABTU algorithm and the communication with other peers. The ABTU algorithm and the management represent the backend.

The goals for this project are:

1. Implement peer-to-peer communication between to sites.
2. Design and implement the ABTU algorithm and the interface with the management.

If the two first goals are achieved and time is left, the next goals are:

3. Test the implementation of the ABTU algorithm.
4. Evaluate the performance of the ABTU algorithm.
5. Link the implementation of the ABTU algorithm with the management and the frontend.

3 Corresponding Background

3.1 Operational Transformation

The ABTU algorithm is based on operational transformation, this subsection will introduce this concept.

A document is represented by a string of characters. The positions of the characters start from 0 and increase until the end of the document. A modification of the document is represented by two types of operations: insertions (INS) and deletions (DEL). As we will see, the identity of the site at which the operation was generated is also important. An operation is therefore defined by its site identity, type, position and character: $OP(siteId, type, position, character)$.

In figure 1, the initial state of the document is "abc". We apply the operation $OP(1, INS, 4, d)$, and the resulting string is "abcd".

In collaborative editing, multiple users simultaneously generate operations at different site, this means that they edit the document concurrently. When

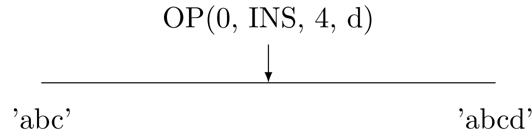

Figure 1: Execution of an Operation on a String. Own Illustration

an operation is generated at one site, it is first applied to the local copy of the document and then distributed to other peers. Therefore, at a given time, two sites do not necessarily have the same version of the document.

The following example from figure 2 will show why operations need to be transformed. Starting from the same document state "abc", site 1 and 2 respectively generate, apply and distribute operations $o_1 = OP(1, INS, 0, z)$ and $o_2 = OP(2, DEL, 3, c)$. After site 1 has executed operation o_1, its state is "zabc", one easily see that the operation o_2 no longer has the desired effect on the document: instead of deleting the character "c", it would remove "b", resulting in the state "zac". This example illustrates the fact that operation o_2 should be transformed to $o_2 = OP(2, DEL, 4, z)$. On the contrary, at site 2, the operation o_2 is already valid in the document state 'ab'. This is what is called *operational transformation*.

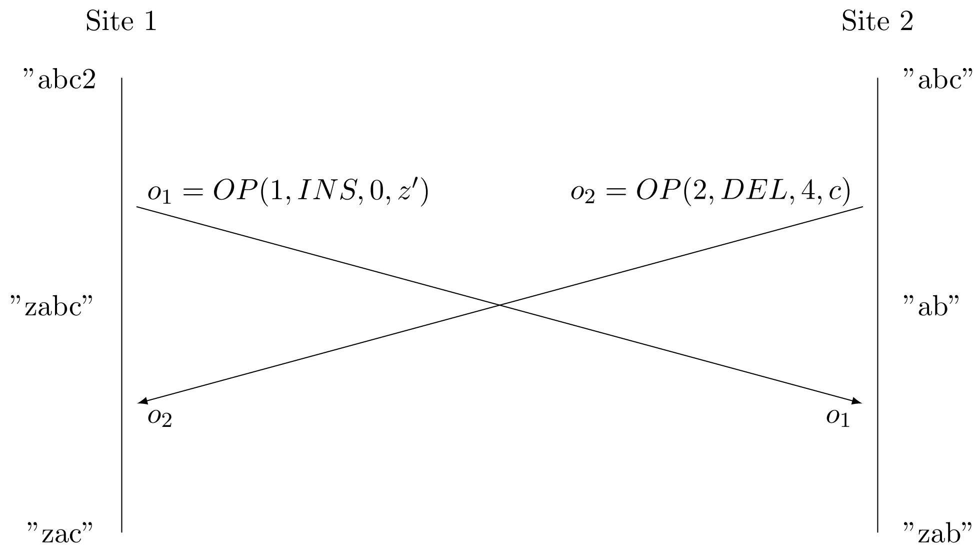

Figure 2: Generation of two Concurrent Operations. Own Illustration.

When multiple collaborators are editing a document concurrently, operations must be verified and transformed if necessary before being applied to the local copy of the document, this is what the ABTU algorithm handles.

3.2 ABTU Algorithm

In this subsection, notions about the framework of the algorithm and notations from [LL10] and [SLG10] will first be introduced, the algorithm will then be described.

3.2.1 Notions and Notations

Operation As seen in section 3.1, an operation is defined by its site id: $o.id$, type: $o.t$, character: $o.c$ and position: $o.p$.

Definition state: The definition state $dst(o)$ of an operation o, is the state of the document in which the operation has been generated. The result of the execution of o in $dst(o)$ is written $dst(o) \circ o$.

We note $o_1 \sqcup o_2$ if both operation o_1 and o_2 have the same definition state $dst(o_1) = dst(o_2)$ and $o_1 \mapsto o_2$ if $dst(o_1) \circ o_1 = dst(o_2)$.

Operation sequence: The execution of a sequence of operations $sq = [o_1, o_2, \cdots, o_n]$ is the result of the consecutive execution of its operations to the definition state of o_1: $dst(o_1) \circ o_1 \circ o_2 \circ \cdots \circ o_n = dst(o_1) \circ sq$. Two operation sequences sq_1 and sq_2 are equivalent if $dst(sq_1) = dst(sq_2)$ and $dst(sq_1) \circ sq_1 = dst(sq_2) \circ sq_2$.

Vector time and timestamp: In order to keep track of time in multiple places, the ABTU algorithm uses the concept of vector clock. For n sites, the time is represented by an array of integers of length n. Time increases as operations are applied at each site.

Each site has its own time vector SV. When an operation is generated at site i, the time vector SV_i is incremented by 1 at index i: $SV_i[i] = SV_i[i] + 1$. When a remote operation from site j is integrated at site i, SV_i is incremented at index j.

Time vectors can be partially ordered: $t_1 < t_2$ iff $\forall_i : t_1[i] \leq t_2[i]$ and $\exists_j : t_1[j] < t_2[j]$.

An operation also carries its own timestamp $o.v$. For an operation o generated at site i, $o.v$ is the time SV_i from site i at which o was generated. We write $o_1 \rightarrow o_2$ if $o_1.v < o_2.v$ and we say that o_1 and o_2 are concurrent ($o_1 \mid\mid o_2$) if neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$ hold.

Effects relation: A first definition can be given as follows: $a \prec b$ if the position of a is smaller than the position of b in the document.

However, this definition is not entirely satisfactory, this can be shown by an example from [SLG10] in figure 3. Three sites start with the same state

"ab", the effect relation is a $\prec$ b. Site 2 first generates $OP(2, INS, x, 2)$ and then receives $OP(1, DEL, 0, a)$ and $OP(1, DEL, 0, b)$ from site 1 and then $OP(3, INS, x, 1)$ from site 3.

As x and y were not generated at the same site, there is no direct effect relation between them, but it is still possible to order them by transitivity. As a $\prec$ b $\prec$ x and y $\prec$ a $\prec$ b, by transitivity y $\prec$ a $\prec$ b $\prec$ x and y $\prec$ x.

If two sites concurrently insert a character at the same position, the ordering is made accordingly to the id of both sites.

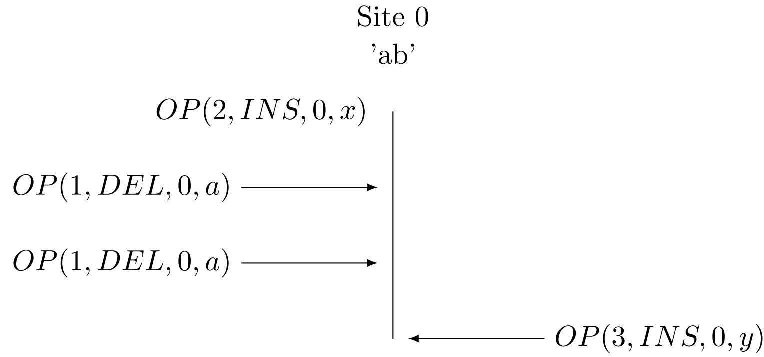

Figure 3: Effects Relation. Own Illustration.

Effects relation order: A sequence of operation is said to be in effects relation order if its operations are ordered according to the effect relation of their characters. This order is denoted by $<_e$, $=_e$ and $>_e$.

For two operations o_1 and o_2 where $o_1 \sqcup o_2$ holds, $o_1 <_e o_2$ if (1) $o_1.p < o2.p$; (2) $o_1.p = o_2.p, o_1.t = INS, o_2.t = DEL$; or (3) $o_1.p = o_2.p, o1.t = o_2.t = INS, o_1.id < o_2.id$. When $o_1 \mapsto o_2$ holds, $o_1 <_e o_2$ if (1) $o_1.p < o2.p$; (2) $o_1.p = o_2.p, o1.t = o_2.t = DEL$.

The $>_e$ operator is completely symmetric to $<_e$, and $o_1 =_e o_2$ is true if neither $<_e$ nor $>_e$ hold.

3.2.2 Undo of Operations

The undoing of an operation can simply be achieved by generating and executing its inverse operation. However, there are some subtleties to this process.

If o is an undo operation, $o.ov$ is the timestamp of the operation it undoes.

As an operation cannot be undone twice, we use $o.uv$ as the set of timestamps of operations that undo (or try to re-undo) o. When an operation is

undone twice, the two undo operation get merged into one. Their respective timestamps also get merged to form sets of timestamps.

Some operations cannot be undone. If we have $o_1 = OP(1, INS, 0, a)$ and $o_2 = OP(1, DEL, 0, a)$ then o_1 can no longer be undone because o_2 is dependent on o_1. To keep track of those dependencies, $o.dv$, the set of timestamps of operations that depend on o is used.

The set of timestamps $o.tv$ contains the timestamps of all operations whose effect objects are identical to $o.c$.

3.2.3 Description of the Algorithm

The ABTU algorithm is based on two important principles which are *Causality Preservation* and *Admissibility Preservation*.

Causality Preservation: Causality is preserved if, for any two operations o_1 and o_2, if $o_1 \rightarrow o_2$, then o_1 must be invoked before o_2 at any site. Therefore, a remote operation from site j is *causally ready* at site i iff $\forall_{k \neq j} SV_i[k] \leq o.v[k]$ and $o.v[j] = SV_i[j] + 1$. If remote operations are only executed when they are causally ready, causality preservation is respected.

Admissibility Preservation: Admissibility is preserved if the execution of any operation respects the effects relation $\prec$ of the system. At any site, remote operation must be transformed so that they become admissible and can be executed.

As seen in 3.1, before being executed, remote operations need to be transformed. In order to transform operations against each other, each site must maintain a history buffer H of every operation already executed. H is an operation sequence maintained in effects relation order. When remote operations are received, each site will transform, execute and integrate them into H.

When an undo operation is generated, the original operation must be recovered from the history buffer and its inverse generated. The undo operation is then executed, distributed and stored just after the original operation in H.

The ABTU algorithm has been proved to converge at some point in the future, i.e. all history buffers in all sites of the system will, at some point, end up to be equivalent sequences of operations.

Both local and remote operations need to be handled by the ABTU algorithm. The psoeudocode for the described functions is available in Appendix A.

Local Operations

Local operation must be timestamped, integrated into H and distributed to other peers, this is done by *localThread* and *integrateL* functions.

Function localThread: When an operation o is generated at site i, function localThread simply increments $SV_i[i]$, adds this timestamp to o, calls integrateL and propagates the result to other sites.

If the operation is an undo, localThread searches the original operation o in H, and checks if an undo is possible with $o.uv$ and $o.dv$. It then generates its inverse and sets all the necessary timestamps. The time $SV_i[i]$ is then incremented, integrateL called and the result distributed.

Function integrateL: Function integrateL scans H from right to left and modifies the position of all operations $H[k]$ as long as $H[k] <_e o$. If $H[k] =_e o$, $o.tv$ and $H[k].dv$ are set accordingly. It is important to notice that $H[j] \mapsto o$ to make the correct checks for the $<_e$ relation.

For an undo operation u, the index j in H of the original operation is recovered with the help of $u.uv$. All operations $H[k]$ $\forall_{k>j}$ are transformed against u and u is inserted at position $k+1$.

Remote Operations

Remote operations are first stored in a receiving buffer RB. When no local operation has to be integrated, the *remoteThread* and *integrateR* function can transform, integrate and execute the first causally ready operation from RB.

Function remoteThread: When a causally ready operation form site j is integrated at site i, the remoteThread function calls integrateR and increments $SV_i[j]$.

If the operation is an undo u, the original operation o is also recovered from H and $o.uv$ is set to $u.v$. If $o.uv \neq$, then o has already been undone by some operation u'. The timestamps of u' are updated accordingly and u discarded. The time $SV_i[j]$ is then incremented, integrateR called and the result distributed.

Function integrateR: The function integrateR serves two purposes:

1. Transform o against all operations $H[k]$ where $o \mid\mid H[k]$ holds and integrate o in H. Indeed, at the generation of o, $H[k]$ was unknown to the site j and therefore o has not taken into account $H[k]$. As H is sorted in effects relation order, the transformation must only be done as long as $H[k] <_e o$. It is important to notice that $H[k] \sqcup o$ holds when the $<_e$ check is made.
2. Transform all operations $H[k]$ for which $H[k] >_e o$ is true against o.

If the operation is an undo u, the original operation o is recovered from H with the help of $o.tv$ and $u.p$ is set to $o.p+1$. H is then scanned from left to right, starting from o, to integrate the effect of concurrent operation whose effect object are identical to $u.c$. If $\exists H[k]$ s.t. $H[k] =_e u$, then o has already been undone by $H[k]$ and the two operations are merged. The operation u is finally integrated into H just after o and executed.

4 Design and Implementation

As explained in section 2, the general structure of the software is organized in three parts. The frontend manages the interaction with the user and the database. It is coded in JavaScript. The ABTU algorithm is implemented in Go and represented by an *ABTUInstance* structure. Communication with the ABTU instance is made through four different channels. The management, also implemented in Go, manages different instances of the ABTU algorithm, handles the communication between frontend and the ABTU instance, and the communication with other peers. The communication between frontend and backend is done through a web socket.

The design of the model has been modularized to allow for greater flexibility: the ABTU instance can be simply be plugged in any management and frontend which respect the interface and the communication protocol.

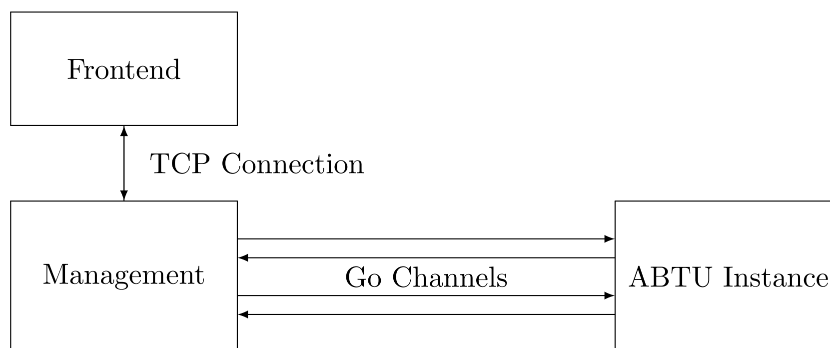

Figure 4: General Organization of the Software. Own Illustration.

4.1 ABTU Instance

The ABTUInstance struct is shown in figure 5. It stores the siteId, site timestamp and history buffer, and uses four channels for the communication with the management.

```
type ABTUInstance struct {
	id SiteId
	sv Timestamp // Site timestamp
	h []Operation // History buffer

	// ...

	// channel for receiving from local frontend
	localToABTU chan []byte
	// channel for sending to local frontend
	aBTUToLocal chan []byte

	// channel for receiving remote operations
	remoteToABTU chan []byte
	// channel for dispatching local operations to remote sites.
	aBTUToRemote chan []byte

}
```

Figure 5: ABTUInstance Struct. Own Code.

4.2 Receiving Buffer Manager

When remote operations are received through the remoteToABTU channel, they must be stored in a receiving buffer. Concurrently, the remote thread reads in the remote buffer to find the first causally ready operation to integrate. There is a clear need for a concurrent data structure in this situation, the *receivingBufferManager* (rbm) serves this purpose. This data structure is inspired from [McG].

As shown in figure 6, requests to the datastructure are represented by structs, containing a return channel, and sent through channels to the rbm. The possible requests are adding a remote operation to the buffer, getting the first causally ready operation given a SV or delete the last causally ready operation returned from the buffer.

Once started, the rbm continuously checks for incoming requests by using a select control structure on different channels. When the first causally ready operation is requested, it might still not be present in the buffer, in that case, the rbm checks the *aBTUIsWaitingCausallyReadyOp* flag, handles other requests and sends the operation when it is available. Figure 7 shows a simplified version of the control structure.

```
// A request operation
type GetCausallyReadyOp struct {
	CurrentTime Timestamp
	Return      chan Operation
}

type RemoteBufferManager struct {
	// Channels for requests
	Add chan AddOp
	Get chan GetCausallyReadyOp
	RemoveRearrange chan RemoveRearrangeOp

	// The actual receiving buffer.
	rb []Operation

	// ...
}
```

Figure 6: Receiving Buffer Manager Struct. Own Code.

4.3 Communication with Frontend

When the user generates a new operation, it is directly executed on the document in the frontend, and then sent to the ABTU instance for integration and distribution. If the operation is an undo, the frontend sends an undo request to the ABTU instance and waits for the operation. When a remote operation is handled in the ABTU instance, the resulting operation must be sent to the frontend.

Priority is given to local operations on remote ones. When a local operation o is created, o must be integrated into H before any other remote operation. If this is not the case, then the processing of the remote operation will not take into account o and the result will not be valid in the already modified state of the document. This problem can be illustrated by the following set of events. The starting state of the document is 'abc'.

1. User generates operation $o_l = OP(1, INS, 0, x)$ and sends it to ABTU.
2. ABTU integrates remote operation $o_r = OP(2, DEL, 3, c)$. The definition state of o_r is "abc". The goal of o_r is to remove "c" from the document.
3. ABTU sends the resulting operation o_r, which remains unchanged, to the frontend.
4. The frontend applies o_r to the document. The resulting state is "xac" whereas if o_l had been integrated into H before o_r, the result would be "xab". Contrary to the first, the second result is the correct one.

This shows the importance of giving the priority to local operations. As

```
 1 func (rbm *RemoteBufferManager) Start(rb []Operation){
 2     // ...
 3 
 4     // Used to manage pending GetCausallyReadyOp requests
 5     rbm.aBTUIsWaitingCausallyReadyOp = false
 6 
 7     go func () {
 8         for {
 9         select {
10         case addOp := <- rbm.Add:
11             rbm.rb = append(rbm.rb, addOp.Operation)
12             addOp.Ack <- true
13             // If ABTU is waiting for a causally ready operation
14             if rbm.aBTUIsWaitingCausallyReadyOp {
15                 // Get causally ready op
16                 // If present:
17                 // sent it through rbm.causallyReadyOpRetChan
18                 // set rbm.aBTUIsWaitingCausallyReadyOp = false
19             }
20         // Return the first causally ready operation if available
21         case getCausallyReadyOp := <- rbm.Get:
22             // Get causally ready op
23             // If present:
24             // sent it through rbm.causallyReadyOpRetChan
25             // and set rbm.aBTUIsWaitingCausallyReadyOp = false
26             //  If not present:
27             // set rbm.aBTUIsWaitingCausallyReadyOp = true
28         case removeRearrangeOp := <- rbm.RemoveRearrange:
29             // remove the last causally ready op from the buffer
30         }
31     }
32 }()
33 }
```

Figure 7: Receiving Buffer Manager Control Structure. Own Code.

local operations can be generated by the user at any time, when a local operation is received by the ABTU instance, the processing of the remote operation must be cancelled. In the frontend, if a remote operation o_r is received from the ABTU instance before the acknowledgement for the last local operation, o_r must be discarded.

The operations shared between the frontend and the ABTU instance are encoded into Json objects: *SimpleOperation*: {"OpType": OpType, "Character": []byte, "Position": Number}. *OpType* is 0 for insertion and 1 for deletion. The "Character" field is simply the byte representation of the character in utf-8 encoding.

As the use of control messages is needed, the Json representation of simple operations can be wrapped in. The control messages have the following

format: {"Type": TypeToken, "Content": Content}.

TypeToken is one of the following.

- "localOperation"
- "undo"
- "ackRemoteOperation"
- "ackLocalOperation"
- "nackLocalUndo"
- "acklocalUndo"
- "remoteOperation"

The content is either a SimpleOperation, a number for the undo, or nil.

The frontend manages the execution of operations and the communication by following the schema in figure 8. When an undo is requested (16-18), the frontend waits for the the undo operation or a "nack" (33-39), nothing can happen during that time (7-11 and 26-28). When local operations are generated, they are executed and sent to the frontend (13-15). No remote operation can be executed as long as some operations have not been integrated into H (no "ackLocalOperation" received) (26-28), a "nackRemoteOperation" is sent in that case. In the other, the remote operation is executed.

4.4 ABTU Control Structure

The control structure of the ABTU instance (figure 9) gives the priority to local operations by selecting the frontendToABTU channel first (4). If no message is coming from there, it requests the first causally ready operation from the receiving buffer manager (8-10). Again, priority is given to local operations over the waiting for the answer from the rbm (15-18).

4.5 ABTU Local Thread

The function *handleFrontendMessage* distributes the work between *handleLocalOperation* and *handleLocalUndo*. Operations get integrated, distributed and acknowledgments are sent to the frontend.

In the undo case, *localThreadUndo* is called. For a "localUndo" message where the content is n, the nth last locally generated operation must be undone. Therefore, a buffer (*localTimestampHistory*) with the timestamps of locally generated operations is kept in memory. LocalThreadUndo recovers the original operation from H using the localTimestampHistory. If the

```
1  var numberOfPendingLocalOperations int = 0
2  var pendingUndo = false
3
4  // Sending messages when local operation is generated:
5  switch {
6      case localOp generated:
7          if pendingUndo {
8              // Wait for "ackLocalUndo" or "nackLocalUndo"
9              // Tell the user he cannot write.
10             // Handle the incoming message
11         }
12
13         numberOfPendingLocalOperations ++
14         execute(localOp)
15         send({Type:"localOperation", Content: localOp})
16     case undo u generated:
17         pendingUndo = true
18         send({Type: "undo", Content: u})
19 }
20
21 // Receiving messages
22 switch incoming message msg {
23     case msg.Type == "ackLocalOperation":
24         numberOfPendingLocalOperations --
25     case m.Type == "remoteOperation":
26         if numberOfPendingLocalOperations>0 || pendingUndo {
27             send({Type:"nackRemoteOperation", Content:nil})
28         } else {
29             execute(msg.Content)
30             send({Type:"ackRemoteOperation", Content:nil})
31         }
32     case msg.Type == "ackLocalUndo":
33         pendingUndo = false
34         execute(msg.Content)
35     case msg.Type == "nackLocalUndo":
36         pendingUndo = false
37         // Tell the user this operation cannot be undone.
38 }
```

Figure 8: Frontend Controller. Own Code.

original operation cannot be undone, a "nackLocalUndo" message is sent to the frontend.

The function *integrateL* directly operates on abtu.sv and abtu.h.

4.6 ABTU Remote Thread

The function *handleCausallyRemoteOperation* in figure 10 calls the *remoteThread* function. The remoteThread and *integrateR* functions operate on copies of

```
for {
    select {
    // Prioritize local operations
    case bytes := <- abtu.localToABTU:
        abtu.handleFrontendMessage(bytes)
    default:
        // Request causally ready operation
        causallyReadyOperationChannel := make(chan Operation, 1)
        abtu.rbm.Get <- GetCausallyReadyOp{
            abtu.sv, causallyReadyOperationChannel}

        select {
        // Prioritize local operation
        case causallyReadyOp := <- causallyReadyOperationChannel:
            abtu.handleCausallyReadyOperation(causallyReadyOp)
        case bytes := <- abtu.localToABTU:
            abtu.handleFrontendMessage(bytes)
        }
    }
}
```

Figure 9: ABTU Controller. Own Code.

abtu.h and abtu.sv and return the resulting operation, time and history buffer to the caller (3). If the resulting operation is of type unit, it is discarded and the changes applied (25).

At the end of the handleCausallyRemoteOperation function, an answer from the frontend is awaited: if the message is a "ackLocalUndo", the changes are applies to abtu.h and abtu.sv (13-22). However, if the message is a "localOperation", it means that a local operation has been generated by the user in the meantime, and the changes to the history buffer and time should not be executed (9-12).

4.7 The Big Picture

Figure 11 shows the general organization of the implementation of the algorithm. The labeled arrows represent Go channels.

4.8 Communication with Peers

The communication with other peers is done by using the go-libp2p library from [lib]. This library implements advanced peer-to-peer communication features. The *Run* function in figure 12 is inspired from the echo example in the go-libp2p repository from [lib].

```
1  func (abtu *ABTUInstance)
2      handleCausallyReadyOperation(causallyReadyOp Operation){
3      toExecuteOp, h, sv := abtu.remoteThread(causallyReadyOp)
4
5      if toExecuteOp.OpType() != UNIT {
6          // Send result to frontend
7          // Wait for answer
8          switch frontendMsg.Type {
9          case LocalOp:
10             abtu.handleLocalOperation(frontendMsg.Content)
11         case Undo:
12             abtu.handleLocalUndo(frontendMsg.Content)
13         case AckRemoteOp:
14             // Apply changes
15             abtu.sv = sv
16             abtu.localTimestampHistory =
17                 append(abtu.localTimestampHistory, sv)
18             abtu.h = h
19             // Remove causally ready op from rbm
20             ack := make(chan bool)
21             abtu.rbm.RemoveRearrange <- RemoveRearrangeOp{ack}
22             <- ack
23         }
24     } else {
25         //Apply changes
26     }
27 }
```

Figure 10: Remote Thread. Own Code.

The *CommunicationService* struct contains a Host and two channels for sending and receiving operations. A peer is represented by the *ABTUPeer* struct. As secured communication is not yet implemented, a random peer identity can be generated using the testutils.RandPeerID() function from the lib-p2p library.

In the *Init* function, the peers are added to the host.Peerstore (4-8). In the Run function, messages from the mgmtToPeers channel are sent to all peers from the peerstore on the "epflDedisAbtu/Dispatch/0.0.1" protocol (20-28). On the other hand, incoming messages are put in the peersTomgmt channel in the streamHandler function (14-17).

With this simple design, the management can simply feed a channel to distribute messages and listen to a channel to receive messages.

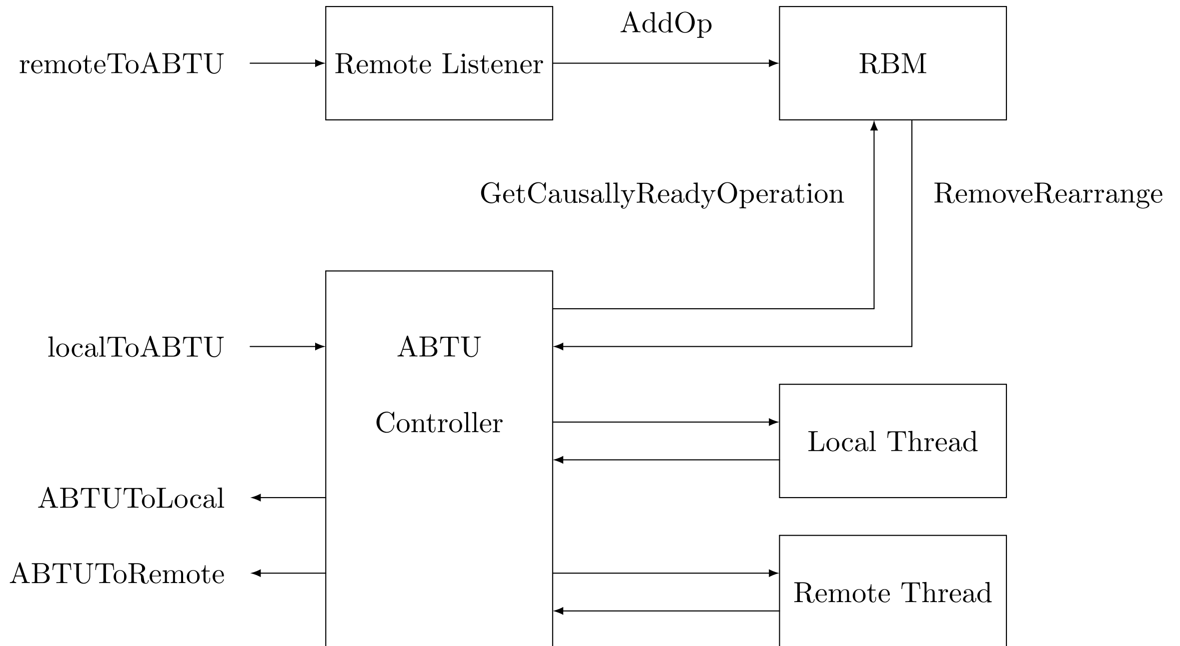

Figure 11: The Big Picture. Own Illustration

5 Results

The following tests on the code base have been completed:

- The receiving buffer manager is tested in the TestSimpleRequests function.
- Some unit tests have been written for the Timestamp struct, mostly for debugging.

The ABTU instance has been tested in three different ways:

1. The TestOneABTUInstance from abtu1_test.go simply feeds one ABTU instance with one local operation and one remote operation. The outgoing messages are simply printed out.
2. The TestABTU2Instances connects two ABTU instances only with their respective channels. Two local operation and one undo are fed into the first instance which then communicates the resulting remote operations to the second instance. Messages are again printed in the console. The test function must answer to "remoteOperation" messages sent to the frontend to avoid a deadlock situation.
3. The TestABTUWithCommunication1 from abtu1_test.go and TestABTUWithCommunication2 from abtu2_test.go work the same way as TestABTU2Instances but use the peer to peer communication. The

```
1  func Init(myId SiteId, ABTUPeers map[SiteId]ABTUPeer)
2      *CommunicationService {
3      // ...
4      for sId, ABTUPeer := range ABTUPeers {
5          // ...
6          host.Peerstore().
7              AddAddr(peerId, multiAddress, pstore.PermanentAddrTTL)
8      }
9  // ...
10 }
11 //...
12 func (comService *CommunicationService) Run()
13     (chan<- []byte, <-chan []byte) {
14     comService.host.SetStreamHandler(COMMUNICATION_PROTOCOL,
15                             func(s net.Stream) {
16         // ...
17         comService.peersToMgmt <- incomingMsg
18     })
19     // ...
20     go func() {
21     for {
22         outGoingMsg := <- comService.mgmtToPeers
23         for _, peer := range comService.host.Peerstore().Peers() {
24             // ...
25             outGoingStream.Write(outGoingMsg)
26         }
27     }
28     }()
29 }
30 // ...
```

Figure 12: Communication. Own Code.

ip addresses must be correctly set in the setupCommunicationService from abtu_test.go.
This test can be done between two computers. As TestABTUWithCommunication2 is the receiver, it must be launched first, the thread then sleeps for 10 seconds to allow TestABTUWithCommunication1 to be launched. Messages are again printed out in the console.

These tests show that the global design of the implementation is correct. However, they do not guarantee that the implementation of the algorithm is faultless. Further testing will have to be done in the future.

6 Limitations of the Project and Future Work

A lot of effort has been put into the design of the implementation to allow for a good understanding and easy maintainability of the project. However, there is room for improvement in some areas.

For the moment, an ABTU instance can be started but not stopped correctly. The mechanism which stops the instance should recover the history buffer, time and receiving buffer. Furthermore, when a peer leaves the collaborative editing, other peers must be informed.

This leads to the second point: messages which are shared between peers only contain operations. There needs to be a more complex communication protocol in order to add features such as a peer leaving or joining a collaborative editing session.

The problem of peers joining is not easy to solve. Indeed, a joining site must first recover the state of the document, the corresponding time and history buffer from another site. Timestamps - now basically implemented as slices of integers in a basic structure - also need to be extended for the joining of peers.

Another area where the implementation can be improved is error handling. For now, if an error occurs in the ABTU instance, the whole program simply fails with log.Fatal(). This has the repercussion of also crashing the management, thus making the frontend and the user completely clueless. An error should be recovered and error messages between the three parts of the software must be integrated in the communication protocol so that the management, frontend or user can be informed and react properly.

One of the main goals for the software this project is part of, is to make it as secure as possible. Adding security to the peer to peer communication would be a cornerstone for achieving this goal.

7 Conclusion

The first goal of the project is completed as the peer-to-peer communication between two sites has been implemented and is fully operational. The second goal is also completed as the implementation of an algorithm for peer-to-peer collaborative editing has been done. The third goal is partly achieved, as some basic tests have been done between two ABTU instances. The two last goals have not entirely been completed. The frontend, management and ABTU instance are almost ready for a linking. Nevertheless, there was not

enough time left to properly join the three parts. For the the performance evaluation, a Go benchmarking file must be set up.

The next phase of the project would be to further test the implementation, evaluate its performance and terminate the linking with the management and frontend. More features could then be added to the software, such as stopping, managing errors, secure peer-to-peer communication and allow peers to join and leave the collaborative editing.

Algorithms for peer-to-peer collaborative editing is a complex but nonetheless interesting subject. The implementation of such an algorithm is challenging and requires a good understanding of the ABTU framework described in [SLG10] and [LL10]. Finally, the project of creating a complete software with an intuitive graphical interface is exciting. The work accomplished during this project will hopefully provide a solid base towards that aim!

References

[lib] libp2p. Github repository of the libp2p implementation in go. https://github.com/libp2p/go-libp2p.

[LL10] Du Li and Rui Li. An admissibility-based operational transformation framework for collaborative editing systems. 2010.

[McG] Mark McGranaghan. Stateful goroutines. https://gobyexample.com/stateful-goroutines.

[SLG10] Bin Shao, Du Li, and Ning Gu. An algorithm for selective undo of any operation in collaborative applications. 2010.

A Installation

The code of the project is available at `https://github.com/DamienAy/epflDedisABTU`. Go must be installed according to these instructions: `https://golang.org/doc/install`. For the peer-to-peer communication, the go-libp2p library must be installed from `https://github.com/libp2p/go-libp2p`.

An ABTU instance can finally be instantiated and run with the code in figure 13:

```
1  var siteId SiteId = 1
2  var numberOfSites int = 3
3  var initialSiteTimestamp Timestamp = NewTimestamp(numberOfSites)
4  var initialHistoryBuffer []Operation = make([]Operation, 0)
5  var initialRemoteBuffer []Operation = make([]Operation, 0)
6
7  // Instantiate
8  var abtu ABTUInstance =
9      Init(siteId,
10           initialSiteTimestamp,
11           initialHistoryBuffer,
12          initialRemoteBuffer)
13
14 // Run
15 LocalToABTU, ABTUToLocal, RemoteToABTU , ABTUToRemote :=
16     abtu.Run()
```

Figure 13: Instantiation of an ABTU Instance. Own Code.

The communication service can be instantiated and run as shown in figure 14.

B Psoeudocode snaphots for local and remote thread

The snapshots from figure 15, 16, 17 and 18 provide from the [SLG10] paper.

```go
var siteId SiteId = 1

peer1 := ABTUPeer{1,
      "QmVvtzcZgCkMnSFf2dnrBPXrWuNFWNM9J3MpZQCvWPuVZf",
      "127.0.0.1",
      "1234" }
peer2 := ABTUPeer{2,
      "QmT1VesmGjDy4LnGzqSAbkr7ntqh67cgedU2dhsMk7dVGL",
      "127.0.0.1",
      "1235" }

ABTUPeers := map[SiteId]ABTUPeer{1:peer1, 2:peer2}

// Instantiate
var comService CommunicationService = Init(siteId, ABTUPeers)

// Run
mgmtToPeers, peersToMgmt := comService.Run()
```

Figure 14: Instantiation of a Communication Service. Own Code.

Algorithm 1 Thread $\mathcal{L}$: process local operation o at site k

1: **if** o is a normal operation **then**
2: $\quad o' \leftarrow o$
3: $\quad sv[k] \leftarrow sv[k] + 1$
4: $\quad o'.v \leftarrow sv$
5: **else** $//o = undo(i)$
6: $\quad$ **if** $H[i].uv \neq \emptyset$ or $H[i].dv \neq \emptyset$ **then**
7: $\quad\quad$ **exit** //undo request invalid
8: $\quad$ **else**
9: $\quad\quad o' \leftarrow \overline{H[i]}$
10: $\quad\quad sv[k] \leftarrow sv[k] + 1$
11: $\quad\quad o'.v \leftarrow sv$
12: $\quad\quad o'.ov \leftarrow H[i].v$
13: $\quad\quad H[i].uv \leftarrow o'.v$
14: $\quad$ **end if**
15: **end if**
16: execute o'
17: $o'' \leftarrow integrateL(o')$
18: propagate o''

Figure 15: Local Thread. [SLG10].

Algorithm 3 *integrateL*(o): o'

1: $o' \leftarrow o$; $k \leftarrow |H|$
2: offset $\leftarrow (o'.t = ins)?1 : -1$
3: **if** $o'.ov = \emptyset$ **then** // o is a normal do operation
4: **for** $(i \leftarrow |H|-1; i \geq 0; i--)$ **do**
5: **if** $H[i] >_e o'$ **then**
6: $k \leftarrow i$
7: $H[i].p \leftarrow H[i].p$ + offset
8: **else if** $H[i] <_e o'$ **then**
9: **break**
10: **else** // $H[i] =_e o'$
11: $o'.tv \leftarrow H[i].v$
12: **if** $o'.t = del$ **then** $H[i].dv \leftarrow o'.v$
13: **break**
14: **end if**
15: **end for**
16: **else** // o is undo
17: find $H[i]$ such that $H[i].v \cap o'.ov \neq \emptyset$
18: $o'.tv \leftarrow H[i].v$
19: $k \leftarrow i+1$
20: **for** $(j \leftarrow k; j < |H|; j++)$ **do**
21: $H[j].p \leftarrow H[j].p$ + offset
22: **end for**
23: **end if**
24: insert o' into H at position k
25: **return** o'

Figure 16: IntegrateL. [SLG10].

Algorithm 2 Thread $\mathcal{R}$: remote operation o from site r

1: **if** $o.ov \neq \emptyset$ **then** // o is undo
2: find $H[i]$ such that $H[i].v \cap o.ov \neq \emptyset$
3: **if** $(H[i].uv \neq \emptyset)$ **then**
4: find $H[j]$ such that $H[j].v \cap H[i].uv \neq \emptyset$
5: $H[j].v \leftarrow (H[j].v) \cup (o.v)$
6: $sv[r] \leftarrow sv[r] + 1$
7: **return**
8: **else**
9: $H[i].uv \leftarrow o.v$
10: $o.ov \leftarrow o.ov \cup H[i].v$
11: **end if**
12: **end if**
13: $o' \leftarrow integrateR(o)$
14: $sv[r] \leftarrow sv[r] + 1$
15: **if** $o' \neq \phi$ **then** execute o'

Figure 17: Remote Thread. [SLG10].

Algorithm 4 $integrateR(o) : o'$

```
1: o' ← o;   k ← |H|
2: if o'.tv = ∅ then
3:     for (i ← 0; i < |H|; i++) do
4:         if H[i] ∥ o' then
5:             offset ← (H[i].t = ins)?1 : −1
6:             if H[i].tv ≠ ∅ or H[i] <_e o' then
7:                 o'.p ← o'.p + offset
8:             else if H[i] >_e o' then
9:                 k ← i; break
10:            else // H[i] =_e o'
11:                o' ← φ
12:                H[i].v ← H[i].v ∪ o'.v
13:                break
14:            end if
15:        else if H[i] >_e o' then      // H[i] → o'
16:            k ← i
17:            break
18:        end if
19:    end for
20: else // o'.tv ≠ ∅, covering the undo case
21:    find H[i] such that H[i].v ∩ o'.tv ≠ ∅
22:    o'.p = H[i].p;   k ← i + 1
23:    while H[k].tv ∩ H[i].v ≠ ∅ do   //H[k] =_e H[i]
24:        if H[k] <_e o' then
25:            o'.p ← o'.p + offset
26:        else if H[k] >_e o' then
27:            break
28:        else // H[k] =_e o'
29:            o' ← φ
30:            H[k].v ← H[k].v ∪ o'.v
31:            break
32:        end if
33:        k ← k + 1
34:    end while
35: end if
36: if o' ≠ φ then
37:    offset ← (o.t = ins)?1 : −1
38:    for (j ← k; j < |H|; j++) do
39:        H[j].p ← H[j].p + offset
40:    end for
41:    insert o' into H at position k
42: end if
43: return o'
```

Figure 18: IntegrateR. [SLG10].