User Interface for Video and Image Stitching

1st Nguyen, Trung Han
Open Distributed Systems
Technische Universitt Berlin
Berlin, Germnay
trung.h.nguyen@campus.tu-berlin.de

2nd Tran, Minh Duc
Open Distributed Systems
Technische Universitt Berlin
Berlin, Germnay
minh.d.tran@campus.tu-berlin.de

3rd Tran, Nhat Duc
Open Distributed Systems
Technische Universitt Berlin
Berlin, Germnay
nhat.d.tran@campus.tu-berlin.de

***Abstract*—Due to progresses and benefits of cloud computing in the recent years, it has become easier than ever for developers to deploy any kind of services over the internet. Using and flexibly provisioning of resources from a pool shared by third-party organizations for reasonable prices, developers are no longer troubled with IT-infrastructural maintenance issues. Fraunhofer FOKUS developed a backend for video and image stitching relying on such a cloud computing solution in forms of Amazon's Simple Queue Service (SQS). Utilizing that tool however has been rather difficult because a proper frontend is still missing. For that purpose, we implemented a corresponding web user interface. This work presents all relevant steps of its realization and explains the most important conceptual decisions that were made. While in itself, the implemented user interface works flawlessly, the communication with the stitching backend via SQS ran into problems. Because of an unforeseeable error and the developers of the FOKUS backend not always being contactable, the final stitching result is not available at the moment. Furthermore, whenever two or more clientS concurrently poll on the result queue, the overall process is decelerated greatly. The reason for that might either lie in the long polling algorithm, a possibly budget server deploying the queues or even AWS' own receiving function. An intermediary backend between the UI and the queues is proposed as one possible solution so that the queues only need to handle one requesting unit at a time. In spite of mentioned deficiencies, the implemented prototype still qualifies as a fully functional user interface and forms the basis for a frontend of the existing stitching backend by Fraunhofer FOKUS.**

***Index Terms*—web UI, video and image stitching, MPD, cloud computing, Amazon Simple Queue Service (SQS)**

I. Introduction

With the recent advancements in cloud computing in the areas of cost-efficiency, flexibility, scalability etc., it is easier than ever for developers to provide on-demand software applications over the internet. They no longer require large sums to deploy their services as well as don't need to be troubled with infrastructural maintenance issues, such as under- or overprovisioning of resources, if an offered service exceeds or falls below expectations [1].

The Fraunhofer FOKUS[1] *(Fraunhofer Institute for Open Communication Systems)* is a research unit of the Fraunhofer Society[2] and is concerned with applied research and development of applications in the field of Information and Communications Technology (ICT). Among the projects of FOKUS is the development of a service for video and image stitching, similar to well-known traditional software like Windows Movie Maker[3], but in this case over the internet relying on cloud computing solutions.

Using media URLs from any host or cloud provider, the backend creates an MPD file (manifest file for adaptive media streaming [2]) according to a given stitching configuration. Amazon's Simple Queue Service[4] (SQS) serves as communication channel between frontend and service backend. SQS - being a fast, reliable and scalable message queuing service - makes it simple and cost-effective to decouple the components of a cloud application since the user can offload the administrative burden of operating and scaling, while paying only for what is really needed [3].

As of now FOKUS only provides a stitching backend. A proper user interface for seamless interaction with the backend is still missing. In this work we implemented corresponding a web UI[5] for that very purpose. In particular this inlcudes: **(II)** A glimpse into related work and what elevates this work. **(III)** and **(IV)** The Architecture and implementation, in which non-trivial steps are described and important decisions made while developing the UI are explained. **(V)** An evaluation of the approach that was taken for implementing the user interface as well as a discussion of its strong points but also its flaws. **(VI)** A conclusion to this work and suggestions for future work.

II. Related Work

There are various programs, on- and offline, available to edit videos, images and audios into one file. The majority of them are video editing tools.

For example, *Microsoft Fotos*[6], the successor of Windows Movie Maker and one of the simplest video editing tools, is able to stitch videos, images and audios. Though, this software can not overlap the video and picture track with soundtrack.

Adobe Premiere Pro[7] is the industry-leading video editing software. It also supports functions to adjust the length of different media files as well as to stitch them together. It is however not possible to export an Adobe Premiere Pro project as an MPD file for adaptive streaming.

[1]https://www.fokus.fraunhofer.de/
[2]https://www.fraunhofer.de/
[3]http://windows.microsoft.com/en-us/windows/get-movie-maker-download
[4]http://aws.amazon.com/sqs/
[5]Code available at https://github.com/SmokyDesperado/moveditor
[6]https://www.microsoft.com/de-de/p/microsoft-fotos/9wzdncrfjbh4
[7]https://www.adobe.com/products/premiere.html

An online solution for video editing is *Movie Maker Online*[8]. This Editor has every editing functionalities that we need but it also does not support exporting the finished product as an MPD file.

III. Architecture & Used Technologies

Frauenhofer FOKUS provides an already existing stitching backend. Not much is known except that requests can only be forwarded to that backend via Amazon's Simple Queue Service. The only left component is the frontend we needed to implement. So the architecture we use is similar to a classic frontend-backend architecture, but modified due to the constraint of using SQS.

The frontend utilizes the AngularJS framework and communicates with the back-end via SQS by using the AWS SDK[9] *(Amazon Web Services Software Development Kit)*.

The workflow should be as follows: 1.) Users upload media files (images, videos, audios) to any cloud service or host (e.g. Google Drive, Dropbox etc.) so that these files can be accessed through a publicly accessible URL. 2.) This URL can then be added to the implemented user interface. 3.) On the UI, users edit video, image and audio files and finish their work by sending requests to the back-end via the queues. 4.) At first, all used media files will be segmented. 5.) The requests are pushed to the request queue and then fetched by the FOKUS backend. 6.) The backend creates corresponding MPDs and the results in form of URLs are put on the result queue by the backend for the user to obtain. 7.) These MPD URLs are used to create a stitching configuration, which will again be forwarded to the backend via the queues. 8.) After that process is finished, the user gets the final stitching result.

Figure 1 shows the architecture and the used technologies. It also points out the workflow.

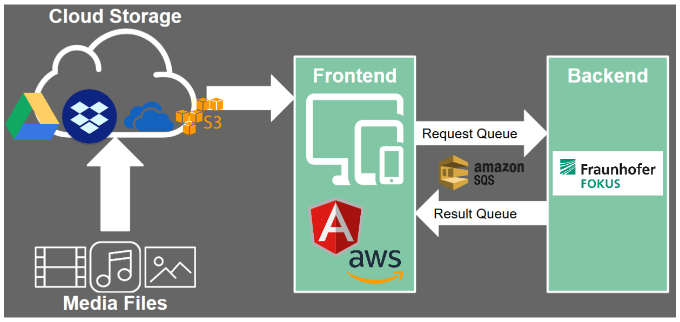

Fig. 1. Architecture and where which technologies are used.

IV. Design & Implementation

This section mainly focusses on the implementation of the user interface for image and video stitching. The realization of major components is presented and important decisions will be explained to give a better understanding of this application.

[8]http://moviemakeronline.com/
[9]https://aws.amazon.com/de/tools/

A. General

For the implementation of the user interface, various libraries were used. Its main structure is generated via AngularJS project scaffolding by Yeoman[10]. In doing so, a ready-to-use AngularJS project is created with provided directory structure, default templates and pre-installed components required for running the application as well as packages to boost the development process.

Node.js[11] serves as the main run-time enviroment for the development process and is used to create the server while developing. New modules to help the development can be installed via the package manager npm[12]. The used packages are saved in the file *package.json* and can be managed from there, e.g. changing the used packages or package versions.

Grunt[13], a Node task runner module, is used for the actual development server configuration. It enables defining tasks and using them to create custom development conditions and helpers. Tasks can be defined in the required *Gruntfile*. While the predefined *serve* task creates a local server on port 9000 with live reload, the *build* task does optimizations to the code (concat, minify, uglify etc.) and then outputs these in a specified directory. The latter is also used for production deployment.

For client sided libraries, that are utilized only in the application but not for the deployment, the Node module *Bower*[14] is used. Like the Node file package.json, bower has it own package management file, called *bower.json*. There the used libraries for the app are listed and can be managed easily.

AngularJs is used as JavaScript framework and allows the implementation of single-page web applications to create a user experience similar to desktop apps. Furthermore, its specific HTML attributes, like *ng-if*, *ng-repeat* or *ng-class*, and its *two-way-binding* principle eliminate the need for manual binding of variables and synchronization between HTML and JavaScript files. The ability of AngularJs to create reuseable modules, called directives, has a big impact on removing duplicate code, thus improving the overall code structure.

B. Content Area

The content area contains all the video, image and audio resources that are used in the app for stitching and creating new videos. It is further divided into two components: A small menu and the actual pool of imported media. New media sources can be added to the pool using the input field of the content area menu with a publicly accessible URL from any cloud provider or host[15]. Via HTML Audio/Video events, meta information (thumbnails, type or file length) will be loaded and added asynchronously to the object representating the resource that we call content (material) from now on. For a better distinction, the middle frame of each video is used as

[10]http://yeoman.io/learning/
[11]https://nodejs.org/en/
[12]https://www.npmjs.com/
[13]https://gruntjs.com/
[14]https://bower.io/
[15]see README of GitHub repository

thumbnail. Moreover, an icon on the lower right corner of the content also indicates the type of a content material. Using the mouse, materials can be dragged around to add them to the timeline area or to delete them. By double-clicking, individual content can be previewed in a dedicated pop-up player. Also, buttons are provided on the menu to save the state of whole working sessions and to recall them at a later time.

For displaying material in the content area the *contentList* object is used and bonded to the view. Each time that object is updated, the content area view will be refreshed, too. ContentList is a simple JavaScript dictionary object which consist of a random generated *contentId* as key and an instance of the JavaScript class *Content* as a value. The dictionary data type was chosen to enable fast and direct access to the objects of the contentList requiring only the contentId, which can then be passed throughout the application if needed elsewhere.

A Content object consist of a name, a type, a length, an url, an mpd and an active property. The name property is empty at the beginning and the user can change the name after the content material is loaded. The url property is the publicly accessible URL of the video, audio or image that is used to add a material and can not be changed afterwards. Each material URL serves as unique identifier in the contentList so that duplicated material URLs can not be added. As a result, a duplicated content material can still be added to the list by using by using different URLs of the same content. The active property of a material is used to keep track of the amounts of that material referenced on the timeline. Each material added to the timeline increases the active counter and each one removed from the timeline decreases the counter. Unless the active property equals 0 it is not possible to delete the content material from the pool. Last but not least, the mpd property is not editable by the user. It is used to temporarily store the MPD URL after the segmentation process of the FOKUS backend and then used for the stitching request.

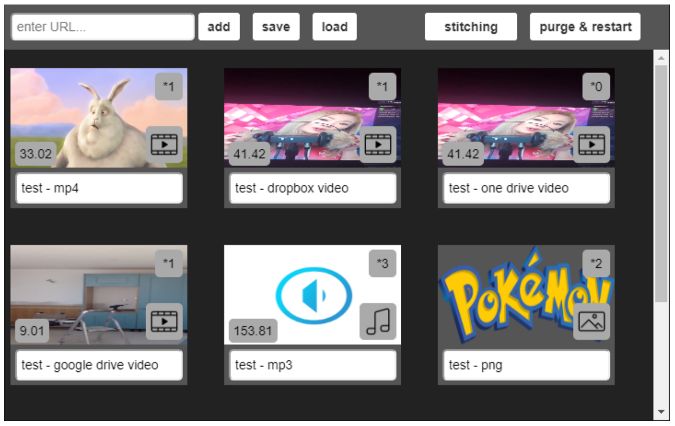

Fig. 2. Content area and menu bar with example resource materials.

The drag and drop functionality to add content materials to the timeline or delete content is enabled through the AngularJS module AngularHammer[16]. The JavaScript library to recognize touch, mouse and pointer events HAMMER.JS[17] was made into a AngularJS module for a simple usage. The three most important callbacks of this module were the *panStart*, *panMove* and *panEnd* events. PanMove provides functionalities to recognise the mouse events and manipulates the DOM element while panStart and panEnd provide manupulation right before and after panMove is executed. This way, it was possible to initialize needed helper variables (i.e. activate attributes to react while moving) and saving the current app state after moving elements. Dragging materials will create a drag indicator, showing and representing the active drag action. The indicator will disappear once the material is dropped and will execute specific actions depending on the dropping area.

A trash can icon appears in the content area whenever a material element is dragged around. When the user drops a content into the bin, that material object is removed from the content area (provided its active property is 0). Showing and hiding the trash is activated using panStart and panEnd. Dropping the material into the timeline will create a new chunk on that area.

Another aspect of the content area menu is saving and loading working sessions. Saving serializes and exports the current state of the content and timeline area while loading will recreate a working session from a saved file. Based on the used data structure of the internal objects to be saved, the created simple text file contains one composed JSON object. Because of the simplicity of the saved file and the absent of validation for loading sessions, a user might be able to input invalid data for the content and timeline area.

To preview individual contents, the user can double click on a specific material to open up a pop-up viewer/player showing images or playing videos and audios. The double click is handled by AngularHammer and will create an overlay on the app. Usually that overlay is hidden in the background but is brought to the front when needed. Dinamically the necessary HTML element is created depending on the content material type to be previewed. By clicking outside the overlay content, the content will be paused (only for audio and video player), destroyed and the overlay is hidden in the background again.

C. Timeline Area

The user will be mainly working on the timeline area to create and edit the stitching video. Unlike the content area, the timeline area is divided into three segments: a timeline bar where materials are added as chunks, a menu to execute serveral functionalities and an information area to view the chunk data. Once added, chunks can be focused and edited by dragging or by interaction with the menu buttons. The information area will show meta data and more detailed information of the focused chunks.

Chunks are JavaScript objects with data about how the material is being used in the timeline. The properties of chunks are *objectListId*, start, end, offset, mute and name. Naming chunks should help the user to easily distinguish two different

[16]https://github.com/RyanMullins/angular-hammer

[17]http://hammerjs.github.io/getting-started

chunks of the same material. Initially, the name of chunks are the same as the created material and can be changed afterwards by the user. Mute allows users to turn off the sounds of a video or audio while playback. The properties start and end are the time values when it starts and when it ends on the timeline. Offset indicates after how much time of the referenced material a chunk starts to play, e.g. if the user wants to start the original material after 9.8 seconds. To connect between the chunks of the timeline and the material of the content, the objectListId is used. It has the contentId of the referenced material and is used to get its properties (length, url, type etc.).

Added material chunks to the timeline are stored in a usual array object, called *timelineList*, unlike contentList which is a dictionary object. Reason for that is to have a simple way to store the chunks and to have an ordered data structure. The amount of chunks on the timeline referencing the same content is unlimited. Like the contentList object the elements of the timelineList is used to generate a visual representation of the chunks. Due to the characteristic of overlapping between video and audio content, the timeline is divided into two parts. Video and image chunks (no overlapping allowed) are positioned in one track while audio chunks are placed on a seperate track so that both tracks can be played at the same time.

The scaling on the timeline makes it easier to estimate the start and end position of chunks. This scaling is dynamically created depending on the final length of the stitched video to be generated. An important variable for this aspect is the *pixelPerSeconds* value, on which the dynamic calculations are based on. It is primarily used to convert the pixel position of the mouse or chunks on the timeline to a time value and vice versa. The visible timeline scales are basically dynamically created DOM block elements, which position is calculated then by the pixelPerSeconds and the time amount of the scale step size. The menu buttons for zooming in and out change pixelPerSeconds and the scale step value and the two-way-binding mechanism of AngularJS updates the view of the timeline according to these values.

Most of the chunk editing functionalities are chunk specific, so the chunks in the timeline can be focused by click events (using AngularHammer). A second click on the same or a click on another chunk will remove its focus. A focused chunk is required for using the delete button. Clicking this button only deletes the focused chunks by removing the element from the array but will do nothing if no chunk is focused.

To rearrange the time position of the chunks on the timeline, the user can drag the chunks to move them. Like in the content area, this dragging feature is provided by AngularHammer. To allow chunks to move only in a limited range, i.e. between the start and end of the timeline or between the end of the previous chunk and the start of the next chunk, the panStart event calculates the limits before the panMove event is executed. In a first version, the limits were used to check if panMove is allowed or not. Due to the delayed detection of fast mouse movement postions, the dragging was often stopped right before the limited position was reached and a undesired space was present. The optimization in the current version uses the limits to detect whether the chunk positon is within the allowd space and sets it to the limit value, if the boundaries are exceeded. The panEnd event then converts the pixel positon of the chunks to a time value and sets them. To eliminate the needs of a very precise mouse position by the user, the time value calculations will be quantized to 100 milliseconds. This may reduce a contents original length but the difference is miniscule and not noticeable.

Similar to dragging the chunks within a given free space on the timeline, the shortening of the start and the end of chunks is executed by dragging the chunk modifier. These modifier are visible when the chunk is focused and not dragged. The limits of the modifier are, besides the limits of the chunk dragging limits, the original start and end of the referenced material. This additional limit prevents chunk offsets less than 0 and ends greater then the original content length.

Initially, it was planed to move chunks on the timeline and reordering the chunks by dragging them. However, to avoid accidential reordering while moving chunks and to increase the UX, the reordering was eventually implemented by menu action buttons. Focused chunks can swapped in order with their previous or their next chunk by reordering the element position within the timelineList array.

Another timeline editing option requiring chunk focus is the cutting of chunks. This feature allows the user to cut one chunk at a specified quantized time position into two seperate chunks, one ends and one starts at the cutted position. A modified chunk clone will be created additionally on the timelineList right after the position of the cut chunk. The clone and the cut chunk values are then set anew. The chunk that was cut starts at its original position and end at the specified cut position. The new added chunk starts from there and end at the cut chunk's original end position. The offset of the new chunk is then recalculated with regards to the original offset by adding the difference of the cut position.

If the user makes wrong actions on the timeline, the undo button can be used to nullify executed step. There are up to ten chronologically ordered states of the timeline stored in the *savedSteps* array. Its states are converted strings of the JSON object timelineList pushed into savedSteps after certain timeline action (panEnd, swap etc.). The JSON string convertion is needed, because merely adding the timelineList object into the array will only reference the timelineList. So each added state would reference the current timelineList in use instead of its chronological state changes. To keep savedSteps at ten states, adding an eleventh state will remove the first state of the array. Undo allows the user to jump back up to ten steps while the redo button will jump forward up to the current state. The *savedStepPointer* tracks the current position in the chronological array while moving for- and backward.

D. Preview Player

The preview player emulates the stitching result of the service backend without actually sending the necessary requests. In doing so, imperfections in the stitching configurations can be detected and corrected via real-time feedback. Users are not impelled to wait for the actual stitching response, thus saving them time.

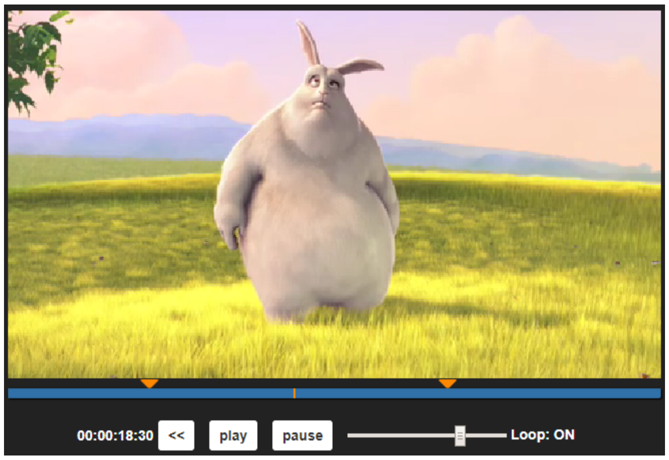

Fig. 3. Look and controls of preview player.

1) Preview Player - Features: The implemented preview player supports all common controls of a usual browser media player. These include: a "play/pause"-button, a "time-position"-slider and a "volume"-slider. On top of that, a "restart"-button, a "loop-play"-toggle and a "play-in-range"-slider were added. For the latter the *noUiSlider*[18] library is utilized. Functionally, the "play-in-range"-slider marks a time range in which the "time-position"-slider is contraint. Almost all of the listed controls can be operated via shortkeys[19] making the web UI behave more like a desktop application.

2) Preview Playback - Logic: For audio and image preview playback it is sufficient to only have one HTML audio, respectively image, element and change its sources in real-time while playing. However, this method does not work for all video sources, e.g. cloud provider URLs generally take more time for the initial loading which leads to stuttering, especially right after swapping video sources. Therefore, per video chunk the implemented player creates one dedicated HTML video element with preloaded source. While playing, it then only manipulates the z-indexes of the video elements and stops or starts them accordingly (see Algorithm 1) resulting in a smoother playback and minimized buffering time.

As optimization, whenever a new video chunk is added to the timeline area, the preview player is signalled to inspect whether there already exists an HTML video element referencing the same source. If there is none, then one will be created. Analogously, if a video chunk is removed from the timeline, then its corresponding HTML video element is deleted if there is no other chunk requiring the same source.

[18]https://refreshless.com/nouislider/
[19]see README of GitHub repository

Algorithm 1 Preview playback loop, simplified

```
1:  cc ← current chunk
2:  pc ← previous chunk
3:  if cc ≠ pc and (pc.type = video or pc.type = audio) then
4:      pcHTML ← corresponding HTML video/audio element
5:      pause pcHTML
6:  end if
7:  if time = timeline-end or time = range-end then
8:      if loop play active then
9:          go to range-start
10:     else
11:         pause player
12:         return
13:     end if
14: end if
15: bring current video/image to the front via z-index
16: if cc.type = video or cc.type = audio then
17:     ccHTML ← corresponding HTML video/audio element
18:     calculate ccHTML offset position
19:     mute ccHTML if necessary
20:     play ccHTML
21: end if
22: pc ← cc
23: time ← time counter + 100 ms
24: diff ← (real-time-passed - time)
25: repeat Algorithm 1 in (100 ms - diff)
```

Due to the nature of Javascript, Algorithm 1 had to be seperated into two functions code-wise or else the parallel updated time counter would start one loop ahead of the preview player resulting in both units not being in sync. To keep the time counter even more precise, a self-adjustment calculation was introduced. By tracking the real-world time passed and matching that with the internal player time, it can be ensured that the preview playback loop is almost always repeated accurately every 100 milliseconds.

E. SQS

Once the user is satisfied with the constructed timeline chunks, the whole stitching procedure can be initiated by pressing the "stitching"-button. The stitching process consists of two phases: 1.) A segmentation phase and 2.) a stitching phase.

For communicating, the stitching backend operates over two seperate SQS queues. From the web client perspective, one queue is for sending requests, the other is needed for receiving results. Both are used for segmentation as well as for stitching requests. The request type is determined through the message body by the service backend. Further, each message body possesses a user defined *jobID* which helps users to identify their messages in the queue for responses. As with other libraries utilized in the developed user interface, functionalities of Amazon's SQS were made accessible by incorporating the official AWS SDK[20] for JavaScript via Bower.

[20]https://github.com/aws/aws-sdk-js#using-bower

```
var sqs = new AWS.SQS({
  "accessKeyId": sqsAccessKeyId,
  "secretAccessKey": sqsSecretAccessKey,
  "region": sqsRegion
});
```

Fig. 4. Setting up an SQS instance in AngularJS.

At the beginning, the condition of the timeline and content area at process starting time are temporarily preserved. By working on those data, accidental user changes and edits in the meantime don't affect the stitching procedure. To avoid further problems, it is not possible to send another stitching request until the whole process is completed.

1) Segmentation Phase: In this context, segmentation means creating an MPD file based on an audio, image or video file, requiring only a publicly accessible resource URL. Theoretically, since fragmentation is applied to files in full length and not single chunks, this step could be executed as soon as a source is added to the content area or when a content is added to the timeline. However, the user has to wait for the stitching process either way and not every resource might be used in the end which would then lead to spamming and wasting storage of the FOKUS backend. Also worth mentioning is that for segmenting images a length must be specified priorly which is only possible after actually working on the timeline.

During phase 1, chunks in the timeline are fragmented in chronological order. A segmentation request message to a chunk is created and put on the request queue (see figure 5). While images need an extra *loop* attribute compared to videos to indicate their length, audio files require an *onlyaudio* attribute, although the fragmentation of latter are currently not supported by the backend.

```
var msg = {
  jobID: segmentationId,
  S3URL: chunkUrl,
  encodingprofile: "default",
  requestEnqueueTime: +new Date()
};

var params = {
  MessageBody: JSON.stringify(msg),
  QueueUrl: requestQueueURL,
  MessageGroupId: 'myGroupId'
};

sqs.sendMessage(params, function(err, data) {
  if (err) {
    console.log('ERROR: ', err);
  } else {
    console.log("send request successfully");
  }
});
```

Fig. 5. Example request for segmenting a video.

After the segmentation request was send, the result queue will be polled for via a recursive receiving function. The reason for this lies in the AWS SDK, which only allows the client to receive at most the 10 oldest messages on the result queue. This in turn means that if 10 or more messages of other users are occupying the queue first, then the client has to wait until those are removed by their target user.

The implemented approach polls for the maximum number of 10 messages at a time and only considers those targeting the user via identifying the *jobID*. Are among those messages no result messages, a progress per chunk fragmentation is shown and the next 10 are polled for. In case one segmentation is done, a message contains an MPD URL, which is saved temporarily and the next chunk will be segmented, but only if its corresponding source URL has not been segmented yet. Either way, those messages are then removed from the queue by its proper recipient so that the next responses in the FIFO queue can be received.

By default, short polling is used in SQS. Long polling though yields several benefits and is enabled by setting the *WaitTimeSeconds* parameter when receiving messages from a queue. The developed algorithm uses the maximum value of 20 seconds. This way, the probability of getting empty replies is disposed of by waiting until messages are available before responding to the client. Nevertheless, messages are still returned as soon as they become obtainable. Long polling further queries all Amazon SQS servers instead of a mere subset so that false empty responses are avoided, too.

2) Stitching Phase: Are all chunks segmented successfully, the stitching phase starts. By sewing MPDs together according to a specified configuration, a new video MPD is generated. Except the message to be transmitted, sending and receiving a stitching request are analog to that of a segmentation request.

```
var configStitching = {
  "LOG_LEVEL": "info",
  "server": {
    "url": "http://localhost:8090",
    "staticFolder": "adstitcher-srv/public/
        mpds",
    "dashEndpoint": "/Users/fr/Documents/
        adstitcher-srv/public/mpds/"
  },
  "content": [
    {
      "type":"video",
      "begin":0.8,
      "end":5.5,
      "offset":0,
      "mute":false,
      "hide":false,
      "url":"https://s3.eu-central-1.amazonaws
          .com/wesualizesegment/exampleVideo1/
          Manifest.mpd"
    },
    {
      "type":"video",
      "begin":6.3,
      "end":10.0,
      "offset":3.7,
      "mute":false,
      "hide":false,
      "url":"https://s3.eu-central-1.amazonaws
          .com/wesualizesegment/exampleVideo2/
          Manifest.mpd"
    }
  ]
}

var msg = {
  jobID: stitchingId,
  config: configStitching,
  requestEnqueueTime: +new Date()
};
```

Fig. 6. Example message of a stitching request

While the segmentation works perfectly, the stitching request results in an error (*errorCode 11: "something wrong with the dashEndpoint in the config. Please check the input config."*). Since the *dashEndpoint* information was provided to us and there was no documentation of the stitching backend as well as the developers not always being contactable, this issue could not be resolved in this work.

3) Problems when receiving messages: During tests it became apparent, that not every polling call resulted in the client receiving messages. Despite that, when only one client is polling for the FIFO result queue at a time, it works perfectly. As soon as more users attempt to receive messages, the whole process is slowed down massively even though long polling is used. It might be that the concurrent pollings from different clients are interferring with each other so that the probability of receiving empty responses is increased rapidly. This way, the FIFO queue can not make any or only very slow progresses. On the other hand it might also be that the SQS queues are deployed on budget servers that are not able to actually handle multiple requests simultaneously.

Even worse, when many clients are polling for results, and some of these abort this process, then that might lead to stagnation of the result queue, whereby no active user can receive any more messages.

In this prototype version of the UI, a "purge&restart"-button was implemented as a safety measure for now. This button will remove all messages on the queue and restart the FOKUS backend server, although this is no longterm solution to menttioned problems.

V. Discussion

In itself, the implemented user interface works without causing any problems. The intuitive UI and the vast amount of implemented features, mostly inspired by professional video tools, offer a great variety of editing possibilities. Most work related to SQS however, is not impeccable. The stitching phase error aside, the approach that was taken to communicate with the FOKUS stitching backend still remains immature. Whenever two or more clients are simultaneously waiting for responses from the queue, it takes much longer until results are transmitted. First and foremost, it's due to Amazon only allowing to receive not more than 10 messages per query, favouring congestion of the result queue. Above all, concurrent polling by many clients on the same queue severely increases the probability of receiving empty responses for whatever reasons. Since this is the case, it makes sense to debate whether Amazon's Simple Queue Service is suited as interface between the implemented frontend and the existing backend. We believe that the already stated potential benefits of using SQS outweigh the problems we encountered during this work. Theoretically, if every long polling delivered non-empty responses, the implemented approach would work flawlessly because each client would quickly remove its own messages from the queue as soon as they are available.

One possible solution to this might be a dedicated intermediary backend (IB) managing the sending of requests and receiving of results. Instead of direct interaction with the queues, each client will then have to use an API provided by the IB. The intermediary backend remembers clients sending requests via its API and forwards these requests to the SQS request queue. Then the IB alone extracts messages from the SQS result queue and forwards these messages to their actual targets. Messages that can not be forwarded immediately are stored so they can be send again later. After several retries and a timeout, that message will be deleted assuming its target client is either inactive or aborted the stitching process. This way, the SQS result queue only needs to handle one polling instance. On the downside, the intermediary backend would be required to cope with all the clients though. How exactly this new issue is to be resolved, can not be decided right now. Nevertheless, there should exist countless ways to implement such a backend.

VI. Conclusion

In this work a prototype of a user interface offering many features and options for video and image stitching was implemented. Video, audio and image content resources can be added to the application by simply using a publicly accessible URL. On a timeline, chunks can be constructed to a new video by utilizing different editing possibilities. Real-time feedback is given via a dedicated preview player emulating the stitching results without doing actual video cutting or processing. By serializing whole working sessions, its state can be shared or recalled conveniently at a later time. One of the main purposes though, i.e. to make the UI interact with the existing stitching backend by Fraunhofer FOKUS, could be satisfied only partially. Having one sole client communicate with the provided FIFO queues is working perfectly fine. As soon as more users concurrently attempt to poll for answers on the result queue, that queue becomes a bottleneck even though a more beneficial long polling method is used. The reason for that might be in the implemented receiving algorithm or in the load handling capability of the server deploying the queues. Possible solutions involve approaches using an intermediary backend to take care of sending and receiving messages from the SQS queues. Another issue we encountered is that while the segmentation phase delivers the expected MPDs, the stitching phase results in an error. Since there was no documentation of the FOKUS backend and with its developers not always being contactable, this problem could not be resolved in our work as of now. In spite of mentioned deficiencies, the implemented prototype still qualifies as a fully functional user interface and forms the basis for a frontend of the existing stitching backend by Fraunhofer FOKUS.

Future Work: While the realized prototype provides many user interaction possibilities, there is still room for further valuable features and improvements. One of the most handy additions would be to control the preview player time and play-in-range positions by utilizing the currently only indicating position markers on the timeline. Also, the un-/redo features merely apply for timeline changes at the moment. When executed, “active-element”-updates on content resources through adding and deleting chunks won’t be taken into account. It is desired to monitor these changes and other operations done on the content area. Editing multiple chunks at once via focussing more than one chunk would additionally increase the UX. Furthermore, dragging new chunks to the timeline, moving or shortening chunks would be more appealing if tooltips are displayed which show time stamps during these actions to increase the user experience. An option to set the quantization value in the timeline would be another useful feature, although very small values might need too much processing power. Moreover, the URL acceptance check and resource type whitelist matching algorithms would greatly benefit from enhancements, too. Loading saved session files might need correctness checks in a future revision as well, so that this point can not be exploited, providing more robustness and consistency.

Regarding the communication with SQS, besides fixing the stitching phase error, it is suggested to implement a dedicated intermediary backend to take care of sending and receiving messages (as mentioned in **V. Discussion**). Otherwise, strategies must be devised, so that concurrently receiving clients won’t result in a bottleneck.

References

[1] Fox, A., et al. (2009). “Above the clouds: A berkeley view of cloud computing”. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13), 2009.

[2] Sodagar, I. (2011). “The mpeg-dash standard for multimedia streaming over the internet”. *IEEE MultiMedia*, (4), 62-67.

[3] Varia, J., & Mathew, S. (2014). “Overview of amazon web services”. *Amazon Web Services*.